Fully Dynamic Algorithm for Constrained Submodular Optimization

Silvio Lattanzi[*]
Google Research
silviol@google.com

Slobodan Mitrović[*]
MIT
slobo@mit.edu

Ashkan Norouzi-Fard[*]
Google Research
ashkannorouzi@google.com

Jakub Tarnawski[*]
Microsoft Research
jatarnaw@microsoft.com

Morteza Zadimoghaddam[*]
Google Research
zadim@google.com

Abstract

The task of maximizing a monotone submodular function under a cardinality constraint is at the core of many machine learning and data mining applications, including data summarization, sparse regression and coverage problems. We study this classic problem in the fully dynamic setting, where elements can be both inserted and removed. Our main result is a randomized algorithm that maintains an efficient data structure with a poly-logarithmic amortized update time and yields a $(1/2 - \epsilon)$-approximate solution. We complement our theoretical analysis with an empirical study of the performance of our algorithm.

1 Introduction

Thanks to the ubiquitous nature of "diminishing returns" functions, submodular optimization has established itself as a central topic in machine learning, with a myriad of applications ranging from active learning [GK11] to sparse reconstruction [Bac10, DDK12, DK11], video analysis [ZJCP14] and data summarization [BIRB15]. In this field, the problem of maximizing a monotone submodular function under a cardinality constraint is perhaps the most central. Despite its generality, the problem can be (approximately) solved using a simple and efficient greedy algorithm [NWF78].

However, this classic algorithm is inefficient when applied on modern large datasets. To overcome this limitation, in recent years there has been much interest in designing efficient streaming [BMKK14, CK14, BFS15, FKK18, NTM$^+$18] and distributed algorithms [MZ15, MKBK15, BENW16, ENV19] for submodular maximization.

Although those algorithms have found numerous applications, they are not well-suited for the common applications where data is highly dynamic. In fact, real-world systems often need to handle evolving datasets, where elements are added and deleted continuously. For example, in a recent study [DJR12], Dey et al. crawled two snapshots of 1.4 million New York City Facebook users several months apart and reported that 52% of them had changed their profile privacy settings significantly during this period. Similarly, Snapchat processes several million picture uploads and deletions daily; Twitter processes several million tweet uploads and deletions daily. As one must still be able to run basic machine learning tasks, such as sparse recovery or data summarization, in such highly dynamic settings, we need fully *dynamic algorithms*: ones able to *efficiently* handle a stream containing not only insertions, but also an arbitrary number of deletions, with small processing time per update.

[*]Equal contribution.

The general dynamic setting is classic and a staple of algorithm design, with many applications in machine learning systems. However, for many problems it is notoriously difficult to obtain efficient algorithms in this model. In the case of submodular maximization, algorithms have been proposed for the specialized settings of sliding windows [CNZ16, ELVZ17] and robustness [MBN$^+$17, KZK18]. However, as we discuss below, these solutions cannot handle the full generality of the described real-world scenarios.

Our contribution. In this paper we design an efficient fully dynamic algorithm for submodular maximization under a cardinality constraint. Our algorithm:

- takes as input a sequence of arbitrarily interleaved insertions and deletions,
- after each such update, it continuously maintains a solution whose value is in expectation at least $(1/2 - \epsilon)$ times the optimum of the underlying dataset at the current time,
- has amortized time per update that is poly-logarithmic in the length of the stream.

This result settles the status of submodular maximization as *tractable* in the dynamic setting. We also empirically validate the efficiency of our algorithm in several applications.

Related work. The question of computing a concise summary of a stream of n data points on the fly was first addressed by *streaming* algorithms. This line of work focuses on using small space, independent of (or only poly-logarithmically dependent on) n. The SIEVESTREAMING algorithm [BMKK14] achieves a $(1/2 - \epsilon)$-approximation in this model, which is tight [FNFSZ20]. The main thresholding idea of SIEVESTREAMING has had a large influence on recent submodular works, including ours. However, streaming algorithms do not support deletions. In fact, the low-memory requirement is fundamentally at odds with the dynamic setting, as any approximation algorithm for the latter must store all stream elements.[2] A natural idea is to adapt streaming algorithms to deletions by storing the stream and recomputing the solution when it loses elements. However, this takes $\Omega(n)$ time per deletion, and is also shown to be inefficient in our experimental evaluations.

A notable related problem is that of maintaining a summary that focuses only on recent data (e.g., the most recent one million data points). This task is captured by the *sliding window model*. In particular, [CNZ16, ELVZ17] give algorithms that optimize a monotone submodular function under the additional constraint that only the last W elements of the stream can be part of the solution. Unfortunately this setting, while crucial for the data freshness objective, is unrealistic for real-world dynamic systems, where it is impossible to assume that data points are deleted in such structured order. In particular, emerging privacy concerns and data protection regulations require data processing platforms to respond rapidly to users' data removal requests. This means that the arrival and removal of data points follows an arbitrary and non-homogeneous pattern.

Another important task is that of generating a summary that is robust to a specific number D of adversarial deletions. This setting is the inspiration for the *two-stage deletion-robust model*. In the first stage, elements are inserted, and the algorithm must retain an intermediate summary of limited size. In the second stage, an adversary removes a set of up to D items. The algorithm then needs to find a final solution from the intermediate summary while excluding the removed items. Despite the generality of the deleted items being arbitrary, this framework assumes that all deletions occur after all items have been introduced to the system, which is often unrealistic and incompatible with privacy objectives. Furthermore, in the known algorithms for this setting [MBN$^+$17, KZK18], the time needed to compute a single solution depends linearly on D, which could be as large as the size n of the entire dataset. Therefore a straightforward use of these methods in fully dynamic settings would result in $\Omega(n)$ per-update time, which is prohibitively expensive.

Finally, a closely related area is that of low-adaptivity complexity. In particular, [FMZ19] is closely related to our work; we build upon the batch insertion idea of the Threshold Sampling algorithm introduced there.

[2] If even one element is not stored by an algorithm, an adversary could delete all other elements, bringing the approximation ratio down to 0.

2 Preliminaries

We consider a (potentially large) collection V of items, also called the *ground set*. We study the problem of maximizing a *non-negative monotone submodular function* $f : 2^V \to \mathbb{R}_{\geq 0}$. Given two sets $X, Y \subseteq V$, the *marginal gain* of X with respect to Y is defined as

$$f(X \mid Y) = f(X \cup Y) - f(Y),$$

which quantifies the increase in value when adding X to Y. We say that f is *monotone* if for any element $e \in V$ and any set $Y \subseteq V$ it holds that $f(e \mid Y) \geq 0$. The function f is *submodular* if for any two sets X and Y such that $X \subseteq Y \subseteq V$ and any element $e \in V \setminus Y$ we have

$$f(e \mid X) \geq f(e \mid Y).$$

Throughout the paper, we assume that f is *normalized*, i.e., $f(\emptyset) = 0$. We also assume that f is given in terms of a value oracle that computes $f(S)$ for given $S \subseteq V$. As usual in the field, when we talk about running time, we are counting the number of oracle calls/queries, each of which we treat as a unit operation. The number of non-oracle-call operations we perform is within a polylog factor of the number of oracle calls.

Submodularity under a cardinality constraint. The problem of maximizing a function f under a *cardinality constraint* k is defined as selecting a set $S \subseteq V$ with $|S| \leq k$ so as to maximize $f(S)$. We will use OPT to refer to such a maximum value of f.

Notation for dynamic streams. Consider a stream of insertions and deletions. Denote by V_i the set of all elements that have been inserted and not deleted up to the i-th operation. Let $\mathcal{O}_i$ be an optimum solution for V_i; denote $\mathrm{OPT}_i = f(\mathcal{O}_i)$.

In our dynamic algorithm we are interested in updating our data structure efficiently. We say that an algorithm has amortized update time t if its total running time to process a worst-case sequence of n insertions and deletions is in expectation at most nt.

3 Overview of our approach and intuitive analysis

In this section we provide an overview of the main techniques and ideas used in our algorithm. To that end we skip some details of the algorithm and present the arguments intuitively, while formal arguments are provided in Section 4. We start by noting that previous approaches either do not support deletions altogether, or support only a limited (and small) number of deletions (with linear running time per deletion) and so they do not capture many real-world scenarios. In this work, we overcome this barrier by designing a novel fully dynamic data structure that has only poly-logarithmic amortized update time.

We start with a few useful observations. For a moment, ignore the values of elements in the ground set V. Consider a set X of k elements sampled uniformly at *random* from V.[3] The set X is very robust against deletions (which, as a reminder, we asssume to be chosen *independently* of the choice of X). Namely, in order to delete an ϵ-fraction of the elements in X, one needs (in expectation) to delete an ϵ-fraction of the elements in V. This property suggests the following fast algorithm, that we refer to by ALG-SIMPLE, for maintaining a set of at most k elements: sample k elements uniformly at random from the ground set, and call that set X; after an ϵ-fraction of the elements in X is deleted, sample another X from scratch. The current set X represents an output after an update. ALG-SIMPLE has expected running time $O(1/\epsilon)$ per deletion, and can also be extended to support insertions in $\mathrm{polylog}(n)$ time. The main issue with this approach is the lack of guarantees on the quality of the output solution after an update, i.e., the approach is oblivious to the values of the elements in V. For instance, the ground set might contain many *useless* elements, hence selecting k of them uniformly at random would not lead to a set of high utility. The main idea in our paper is to partition the ground set into groups (that we call *buckets*) so that applying ALG-SIMPLE within each bucket outputs a robust set of high utility. Moreover, the union of these sampled elements provides close to optimal utility.

[3]Hence, each element from V is sampled with probability k/n.

Our data structure, which we refer to by A, divides the elements into $T = \log n$ *levels*, with each level subdivided into $R = \log k$ buckets. Informally speaking, each bucket is designed in such a way that selecting elements from it by ALG-SIMPLE results in sets that are both robust and high-quality. Our algorithm maintains a set S that represents the output solution at every point; it is constructed by applying ALG-SIMPLE over distinct buckets. Different buckets might contribute different numbers of elements to S.

The structure of each level of A is essentially the same. The main difference is that different levels maintain different numbers of elements, i.e., level ℓ maintains $O(\frac{n}{2^\ell} \cdot \mathrm{polylog}(n))$ many elements. Intuitively, and informally, levels with small ℓ are recomputed/changed only after many updates, while levels with large ℓ, such as $\ell = T$, are sensitive to updates and recomputed more frequently. In particular, if we insert an extremely valuable element, then the level $\ell = T$ will guarantee that this newly added valuable element will appear in S. We now discuss the structure of A in more detail.

We use $A_{i,\ell}$ to refer to the i-th bucket in level ℓ. Each level is associated with a maximum bucket-size, with level 0 corresponding to the largest bucket-sizes. More precisely, we will maintain the invariant

$$|A_{i,\ell}| \leq \frac{n}{2^\ell} \cdot \mathrm{polylog}(n)$$

for all $1 \leq i \leq R$. Organizing levels to correspond to exponentially decreasing bucket-sizes is one of the main ingredients that enables us to obtain a poly-logarithmic update time.

Buckets within each level are ordered so as to contain elements of exponentially decreasing marginal values with respect to the elements chosen so far. To illustrate this partitioning, consider the first bucket of level 0. Let S be the set of elements representing our (partial) output so far; initially, $S = \emptyset$. Then, we define

$$A_{1,0} = \{e \in V \mid \tau_1 \leq f(e \mid S) \leq \tau_0\},$$

where $\tau_i \approx (1-\epsilon)^i\, \mathrm{OPT}$.[4] It is clear that the construction of $A_{1,0}$ takes $\widetilde{O}(n)$ time. After constructing $A_{1,0}$, our goal is to augment S by some of the elements from $A_{1,0}$ so that the marginal gain of each element added to S is in expectation at least τ_1. After augmenting S, we also refine $A_{1,0}$. This is achieved by repeatedly performing the following steps:
From $A_{1,0}$ we randomly select a subset (of size at most $k - |S|$) of elements whose average marginal gain with respect to to S is at least τ_1. In Appendix C we explain how to obtain such a set efficiently. Then we add this set to S. Now, refine $A_{1,1}$ by removing from it all elements whose marginal gain with respect to S is less than τ_1. If $|A_{1,0}| \geq n/2$ and $|S| < k$, we repeat these steps.

Let us now analyze the robustness of $S \cap A_{1,0}$. The way we selected the elements added to S enables us to perform a similar reasoning to the one we performed to analyze the robustness of ALG-SIMPLE. Namely, when an element $e \in A_{1,0}$ is added to S, it is always chosen *uniformly at random* from $A_{1,0}$. Also, the process of adding elements to S from $A_{1,0}$ is repeated while $|A_{1,0}| \geq n/2$. In other words, e is chosen from a large pool of elements, much larger than k. Hence, an adversary has to remove many elements, $\varepsilon|A_{i,\ell}| \geq \varepsilon n/2$ in expectation, to remove an ε-fraction of elements added from $A_{i,\ell}$ to S. Combining this observation with the fact that the construction of $A_{1,0}$ takes $\widetilde{O}(n)$ time is key to obtaining to the desired update time[5].

Note that so far we have assumed that a good solution can be constructed looking only at elements with marginal value larger than τ_1. Unfortunately this is not always the case and so we need to extend our construction. To construct the remaining buckets $A_{i,\ell}$, we proceed in the same fashion as for $A_{1,0}$ in the increasing order of i. The only difference is that we consider decreasing thresholds:

$$A_{i,\ell} = \{e \in V \mid \tau_i \leq f(e \mid S) \leq \tau_{i-1}\},$$

where S is always the set of elements chosen so far. Once all the buckets in level 0 are processed, we proceed to level 1. The main difference between different layers is that for level ℓ we iterate while $|A_{i,\ell}| \geq n/2^\ell$ and $|S| < k$. So, in every level we explore more of our ground set. Importantly, we can show that on every level we consider a ground set that decreases in size significantly.

At first, it might be surprising that from bucket to bucket of level ℓ we consider elements in decreasing order of their marginal gain, and then in level $\ell + 1$ we again begin by considering elements of the

[4]As a reminder, OPT denotes the maximum value of $f(S)$ over all $S \subseteq V$ such that $|S| \leq k$.

[5]Note that actually achieving the desired running time without any assumption requires further adjustments to the algorithm and more involved techniques that we introduce in further sections.

largest gain. Perhaps it would be more natural to first exhaust all the elements of the largest marginal gain, and only then consider those of lower gain. However, we remark that the smallest value of τ_i that we consider is at least $\Theta(\mathrm{OPT}/k)$. Hence, selecting for S any k elements whose marginal contribution is at least τ_i already leads to a good approximation.

Handling Deletions. Assume that an adversary deletes an element e. If $e \notin S$, we remove e only from the buckets it belongs to, without any extra recomputation. If $e \in S$, let $A_{i,\ell}$ be the bucket from which e is added to S. To update S, we reconstruct A from $A_{1,\ell}$. We now informally bound the running time needed for this reconstruction. The probability that an element from $A_{1,\ell}$ belongs to S is $\frac{t}{n/2^\ell}$, where t is the number of elements selected to S from $A_{i,\ell}$. Saying it differently, an adversary has to (in expectation) remove $\frac{n/2^\ell}{t}$ elements from $A_{i,\ell}$ before it removes an element from $S \cap A_{i,\ell}$. Moreover the running time of a reconstruction of $A_{1,\ell}$ is $\widetilde{O}({}^{n}\!/_{2^\ell})$. Putting these two together, we get that expected running time of reconstruction per deletion is $O(t) \cdot \mathrm{polylog}(n)$. To reduce the update time to $\mathrm{polylog}\, n$, we reconstruct S only if, since its last reconstruction, its value has dropped by a factor ε. Since the elements in $A_{i,\ell}$ have similar marginal gain, an adversary would need to remove roughly εt elements from $S \cap A_{i,\ell}$ to invoke a recomputation of $A_{1,\ell}$, leading to an amortized update time of $\mathrm{polylog}(n)$. Unfortunately, formalizing this intuition is somewhat subtle, as elements are removed from multiple buckets and each removal decreases the value of S.

Handling Insertions. Along with A, we maintain buffer sets $B_1, \ldots, B_T$. Roughly speaking, our algorithm postpones processing insertions into level ℓ until there are $n/2^\ell$ many of them; this enables us to obtain efficient amortized update time of the structure on level ℓ. The buffer set B_ℓ is used to store these insertions until they are processed.

More precisely, when an element is inserted, the algorithm adds it to all the sets B_ℓ. When, for any ℓ, the size of B_ℓ becomes $\frac{n}{2^\ell}$, we add the elements of B_ℓ to $A_{1,\ell}$, reconstruct the data structure from the ℓ-th level, and also empty B_ℓ. This approach handles insertions *lazily*. Notice that lazy updates should be done carefully, since if the newly inserted element has very high utility, we need to add it to the solution immediately. During the execution of the algorithm, B_ℓ essentially represents those elements that we have not considered in the construction of buckets in $A_{i,\ell}$ for $0 \leq \ell \leq T$. The property that the running time of constructing $A_{i,\ell}$ is $\widetilde{O}(\frac{n}{2^\ell})$ implies that the amortized running time per insertion is also $\mathrm{polylog}(n)$. Also observe that we add B_T to $A_{1,T}$ after any element is inserted, which enables us to maintain a good approximate solution at all times. In particular, if an element e of very large marginal gain given S is inserted, e.g., $f(e \mid S) > \mathrm{OPT}/2$, then it will be processed via B_T and added to S. In general, if there are 2^j inserted elements that collectively have very large gain given S, then they will be processed via B_{T-j} and potentially used to update S.

4 The algorithm

We are now ready to describe our algorithm. For the sake of simplicity, we present an algorithm that is parametrized by γ: a guess for the value OPT. Moreover we assume that we know the maximum number of elements available at any given time ($\max_{1 \leq t \leq m} |V_t|$), which is upper-bounded by n. Later we show how to remove these assumptions.

Our algorithm maintains a data structure that uses three families of element sets: A and S indexed by pairs (i, ℓ) and B indexed by ℓ. For an integer R that we will set later, the algorithm also maintains a sequence of thresholds $\tau_0 > \ldots > \tau_R$ (indexed by i), where we think that $\tau_0 \approx \gamma$ and $\tau_R \approx \gamma/(2k)$. We use $S_{j,\ell}$ to refer to the elements chosen to S from bucket j of level ℓ. Let $S_{\mathrm{pred}(i,\ell)}$ be the following union of sets:

$$S_{\mathrm{pred}(i,\ell)} \stackrel{\mathrm{def}}{=} \bigcup_{1 \leq j \leq R, 0 \leq r < \ell} S_{j,r} \cup \bigcup_{1 \leq j \leq i} S_{j,\ell}.$$

In words, a set $S_{\mathrm{pred}(i,\ell)}$, where "pred" refers to "predecessors", defines the subset of elements of S chosen from the buckets that precede bucket i of level ℓ, including that bucket itself. At level ℓ and for index i, we define $A_{i,\ell}$ to be the set of items with marginal value with respect to the set $S_{\mathrm{pred}(i,\ell)}$ in the range $[\tau_i, \tau_{i-1}]$. While $A_{i,\ell}$ has at least $2^{T-\ell}$ items, we use a procedure called

PEELING[6] to select a random subset of $A_{i,\ell}$ to be included into the solution set $S_{i,\ell}$. This can be done in multiple iterations; each time, a randomly chosen batch of items will be inserted into $S_{i,\ell}$. This batch insertion logic is named BUCKET-CONSTRUCT and summarized as Algorithm 2. The solution that our algorithm returns is $S_{\text{pred}(R,T)}$, i.e., the union of all sets $S_{i,\ell}$, and we denote by Sol_t this set after the t-th operation.

In order to implement our algorithm efficiently, we need to be able to select a high-quality random subset of $A_{i,\ell}$ quickly. Our data structure enables us to do this using the PEELING procedure from [FMZ19] (whose full description and a precise statement and proofs of its guarantees are provided in Appendix C).[7] This procedure takes as input a set N and identifies a number t and selects a set S of size t uniformly at random such that: i) the average contribution of each element in S is almost τ, ii) a large fraction of elements in N have contribution less than τ, conditioned on adding S to the solution, iii) it uses only a logarithmic number of oracle queries.

To maintain the above batch insertion logic with every insertion, the algorithm may need to recompute many of the A-sets, which blows up the update time. To get around this problem, we introduce buffer sets B_ℓ for each level $0 \leq \ell \leq T$. Each buffer set B_ℓ has a capacity of at most $2^{T-\ell} - 1$ items. When a new item x arrives, instead of recomputing all A-sets, we insert x into all buffer sets. If some buffer sets exceed their capacity, we pick the first one (with the smallest ℓ^*) and reconstruct all sets in levels beginning from ℓ^*. We call this reconstruction process LEVEL-CONSTRUCT. It is presented as Algorithm 5. The insertion process in summarized as Algorithm 3.

When deleting an element x, our data structure is not affected if the deleted item x does not belong to any set $S_{i,\ell}$. But if it is deleted from some $S_{i,\ell}$, we need to recompute the data structure starting from $S_{i,\ell}$. To optimize the update time, we perform this update operation in a lazy manner as well. We recompute only if an ε-fraction of items in $S_{i,\ell}$ have been deleted since the last time it was constructed. To simplify the algorithm, we reconstruct the entire level ℓ and also the next levels $\ell + 1, ...$ in this case. The deletion logic is summarized as Algorithm 4.

We initialize all sets as empty. The sequence of thresholds τ is set up as a geometric series parametrized by a constant $\epsilon_1 > 0$.

5 Analysis of the algorithm

We now state two technical theorems, and in Appendix C.1 we show how to combine them in the main result. Here $\epsilon_1, \epsilon_p > 0$ are parameters of our algorithm; they affect both approximation ratio and oracle complexity. Intuitively, they should be thought of as small constants. (As a reminder, our approach consists of five methods INITIALIZATION, BUCKET-CONSTRUCT, INSERTION, DELETION and LEVEL-CONSTRUCT that are given as Algorithm 1 through Algorithm 5.)

Theorem 5.1 *Let Sol_i be the solution of our algorithm and OPT_i be the optimal solution after i updates. Moreover, assume that γ in Algorithm 1 is such that $(1 + \epsilon_p)\,\text{OPT}_i \geq \gamma \geq \text{OPT}_i$. Then for any $1 \leq i \leq n$ we have $\mathbb{E}[f(Sol_i)] \geq (1 - \epsilon_p - \epsilon(1 + \epsilon_1))\frac{\text{OPT}_i}{2}$.*

Theorem 5.2 *The amortized expected number of oracle queries per update is $O\left(\frac{R^5 \log^2(n)}{\epsilon_p^2 \cdot \varepsilon}\right)$, where R equals $\log_{1+\epsilon_1}(2k)$ (see Algorithm 1).*

Theorems 5.1 and 5.2 are proved in Appendices A and B, respectively. Furthermore, we combine these ingredients with certain well-known techniques to achieve the following result. Its proof is provided in Appendix C.1.

Theorem 5.3 *Our algorithm maintains a $(1 - 2\epsilon_p - \epsilon(1 + \epsilon_1))/2$-approximate solution after each operation. The amortized expected number of oracle queries per update of this algorithm is $O\left(\frac{\log^6_{1+\epsilon_1}(k) \log^2(n)}{\epsilon_p^4 \cdot \varepsilon}\right)$.*

[6] Algorithm PEELING is an implementation of the ideas behind ALG-SIMPLE described in Section 3.

[7] We invoke PEELING on the function $f'(e) = f(e \mid S_{\text{pred}(i,\ell)})$, which is monotone submodular.

Algorithm 1 INITIALIZATION

```
1: $R \leftarrow \log_{1+\epsilon_1}(2k)$
2: $\tau_i \leftarrow \gamma(1+\epsilon_1)^{-i} \quad \forall 0 \le i \le R$
3: $T \leftarrow \log n$
4: $A_{i,\ell} \leftarrow \emptyset \quad \forall 1 \le i \le R \quad 0 \le \ell \le T$
5: $S_{i,\ell} \leftarrow \emptyset \quad \forall 1 \le i \le R \quad 0 \le \ell \le T$
6: $B_\ell \leftarrow \emptyset \quad \forall 0 \le \ell \le T$
```

Algorithm 2 BUCKET-CONSTRUCT(i, ℓ)

```
1: repeat
2:     $A_{i,\ell} = \{e \in A_{i,\ell} \mid \tau_i \le f(e \mid S_{\text{pred}(i,\ell)}) \le \tau_{i-1}\}$
3:     if $|A_{i,\ell}| \ge 2^{T-\ell}$ and $|S_{\text{pred}(R,T)}| < k$ then
4:         $S_{i,\ell} \leftarrow S_{i,\ell} \cup$ PEELING$(A_{i,\ell}, \tau_i, f')$
5:     end if
6: until $|A_{i,\ell}| < 2^{T-\ell}$ or $|S_{\text{pred}(R,T)}| \ge k$
```

Algorithm 3 INSERTION(e)

```
1: $B_\ell \leftarrow B_\ell \cup \{e\} \quad \forall 0 \le \ell \le T$
2: $V \leftarrow V \cup \{e\}$
3: if there exists an index $\ell$ such that $|B_\ell| \ge 2^{T-\ell}$ then
4:     Let $\ell^\star$ be the smallest such index
5:     $S_{i',\ell'} \leftarrow \emptyset \quad \forall \ell^\star \le \ell' \le T \quad \forall 1 \le i' \le R$
6:     $B_\ell \leftarrow \emptyset \quad \forall \ell^\star \le \ell' \le T$
7:     LEVEL-CONSTRUCT$(\ell^\star)$
8: end if
```

Algorithm 4 DELETION(e)

```
1: $A_{i,\ell} \leftarrow A_{i,\ell} \setminus \{e\} \quad \forall 1 \le i \le R \quad 0 \le \ell \le T$
2: $B_\ell \leftarrow B_\ell \setminus \{e\} \quad \forall 0 \le \ell \le T$
3: $V \leftarrow V \setminus \{e\}$
4: if $e \in S_{\text{pred}(R,T)}$ then
5:     Let $S_{i,\ell}$ be the set containing $e$
6:     Remove $e$ from $S_{i,\ell}$
7:     if the size of $S_{i,\ell}$ has reduced by $\varepsilon$ fraction since it was constructed then
8:         $S_{i',\ell'} \leftarrow \emptyset \quad \forall \ell \le \ell' \le T \quad \forall 0 \le i' \le R$
9:         LEVEL-CONSTRUCT$(\ell)$
10:    end if
11: end if
```

Algorithm 5 LEVEL-CONSTRUCT(ℓ)

```
1: $B_\ell \leftarrow \emptyset$
2: for $i \leftarrow 1 \dots R$ do
3:     if $\ell > 0$ then
4:         $A_{i,\ell} \leftarrow B_{\ell-1} \cup \bigcup_{j=0}^{R} A_{j,\ell-1}$
5:     else
6:         $A_{i,\ell} \leftarrow V$
7:     end if
8:     BUCKET-CONSTRUCT$(i, \ell)$
9: end for
10: if $|S_{\text{pred}(R,T)}| \ge k$ then
11:     $A_{i,\ell'} \leftarrow \emptyset \quad \forall \ell < \ell' \le T \quad \forall 1 \le i \le R$

12: end if
13: if $\ell < T$ and $|S_{\text{pred}(R,T)}| < k$ then
14:     LEVEL-CONSTRUCT$(\ell + 1)$
15: end if
```

6 Empirical evaluation

In this section we empirically evaluate our algorithm. We perform experiments using a slightly simplified variant of our algorithm; see Appendix E for more details. We note that this variant also maintains an almost $1/2$-approximate solution after each operation; it differs in the bound on the expected number of oracle queries per update that we can obtain, which is $\tilde{O}(k)$. A proof of these guarantees can also be found in Appendix E.

The code of our implementations can be found at `https://github.com/google-research/google-research/tree/master/fully_dynamic_submodular_maximization`. All experiments in this paper are run on commodity hardware.

We focus on the number of oracle calls performed during the computation and on the quality of returned solutions. More specifically, we perform a sequence of insertions and removals of elements, and after each operation i we output a high-value set S_i of cardinality at most k. For a given sequence of n operations, we plot:

- **Total number of oracle calls** our algorithm performs for each of the n operations.
- **Quality** of the average output set, i.e., $\sum_{i=1}^{n} f(S_i)/n$.

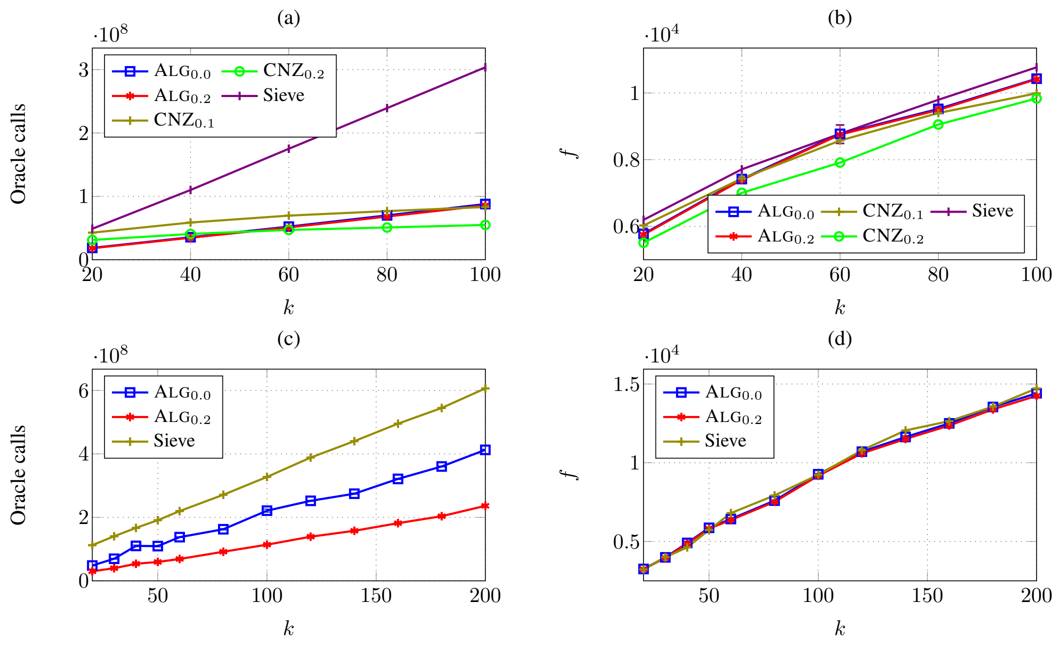

Figure 1: The plots in this figure are obtained for f being the graph coverage function. Plots (a) and (b) show the results on the Enron dataset. We fix an arbitrary order of the Enron email addresses and process them sequentially over windows of size $30,000$. We first insert all elements, and then delete them in the same order. Plots (c) and (d) depict the results for the ego-Twitter dataset. In this experiment the insertions are performed in a random order, while deletions are performed starting from highest-degree nodes.

Dominating sets. In our evaluation we use the dominating set objective function. Namely, given a graph $G = (V, E)$, for a subset of nodes $Z \subseteq V$ we define $f(Z) = |N(Z) \cup Z|$, where $N(Z)$ is the node-neighborhood of Z. This function is monotone and submodular.

Datasets and their processing. We perform evaluations on the Enron ($|V| = 36,692, |E| = 183,831$), the ego-Twitter ($|V| = 81,306, |E| = 1,768,149$), and the Pokec ($|V| = 1,632,803, |E| = 30,622,564$) graph datasets from SNAP Large Networks Data Collection [LK15].

We run two types of experiments on the abovementioned datasets.

1. We consider **a sliding window** of size ℓ over an arbitrary order of the nodes of the graph. When the window reaches a node, we add that node to the stream. Similarly, after ℓ insertions, i.e., when a node leaves the window, we delete it. This provides us with a stream of interspersed insertions and deletions. Moreover, setting ℓ to the number of nodes in the graph is equivalent to inserting all the nodes in an arbitrary order and then deleting them in the same order.
2. We insert all the nodes of the graph in **arbitrary order**. Afterward, we delete them node-by-node by choosing a node in the current solution that has the largest neighborhood. Intuitively, we delete the elements that contribute the most to the optimum solution; this potentially results in many changes to Sol$_i$. We observe that even for this stream, our algorithm is efficient and makes a small number of oracle calls on average.

Due to space constraints, we present the results of only two experiments, one for each of the types. For the first type, we present the results on the Enron dataset for a window of size $\ell = 30,000$. For the second type, we present the results on the ego-Twitter dataset. Further results on other datasets and different values of ℓ are included in Appendix D.

The baselines. We consider the performance of our algorithm for $\epsilon = 0.0$ and $\epsilon = 0.2$, and denote those versions by ALG$_{0.0}$ and ALG$_{0.2}$, respectively. Recall that in our algorithm, if an ϵ-fraction of elements is deleted from the solution on some level, we reconstruct the solution beginning from that level. We cannot compare against the true optimum or the greedy solution, as computing them is intractable for data of this size. We compare our approach with the following baselines:

1. The algorithms of [CNZ16] and [ELVZ17] (developed concurrently and very similar). This method is designed for the sliding window setting and can only be used if elements are deleted in the same order as they were inserted. It is parametrized by ε and we consider values of $\varepsilon = 0.1$ and $\varepsilon = 0.2$, and use $\text{CNZ}_{0.1}$ and $\text{CNZ}_{0.2}$ to denote these two variants.
2. SIEVESTREAMING [BMKK14], which is a streaming algorithm that only supports inserting elements. For any insertion, we simply have SIEVESTREAMING insert the element. For any deletion that deletes an element in the solution of SIEVESTREAMING, we restart SIEVESTREAMING on the set of currently available elements.[8]
3. RND algorithm, which maintains a uniformly random subset of k elements. RND outputs solutions of significantly lower quality than other baselines, so due to space constraints we report its objective value results only in the appendix.

Results. The results of our evaluation are presented in Fig. 1. As shown in plots (b) and (d), our approach (even for different values of ε) is qualitatively almost the same as SIEVESTREAMING. However, compared to SIEVESTREAMING, our approach has a smoother increase in the number of oracle calls with respect to the increase in k. As a result, starting from small values of k, e.g., $k = 40$, our approach $\text{ALG}_{0.2}$ requires at least $2\times$ fewer oracle calls than SIEVESTREAMING to output sets of the same quality for both Enron and ego-Twitter. The behavior of our algorithm for $\varepsilon = 0.0$ is closest to SIEVESTREAMING in the sense that, as soon as a deletion from the current solution occurs, it performs a recomputation (see Line 7 of DELETION). For larger ε our approach performs a recomputation only after a number of deletions from the current solution. As a result, for $\varepsilon = 0.2$, on some datasets our approach requires almost $3\times$ fewer oracle calls to obtain a solution of the same quality as SIEVESTREAMING (see Fig. 1(c) and (d)).

Compared to $\text{CNZ}_{0.1}$ and $\text{CNZ}_{0.2}$ in the context of sliding-window experiments (plots (a) and (b) in Fig. 1), our approach shows very similar performance in both quality and the number of oracle calls. $\text{CNZ}_{0.2}$ is somewhat faster than our approach (plot (a)), but it also reports a lower-quality solution (plot (b)). We point out that CNZ fundamentally requires that insertions and deletions are performed in the same order. Hence, we could not run CNZ for plots (c) and (d), where the experiment does not have that special structure. Since our approach is randomized, we repeat each of the experiments 5 times using fresh randomness; plots show the mean values. The standard deviation of reported values for $\text{ALG}_{0.0}$ and $\text{ALG}_{0.2}$, less than 5%, is plotted in Fig. 8.

7 Conclusion and future work

We present the first efficient algorithm for cardinality-constrained dynamic submodular maximization, with only poly-logarithmic amortized update time. We complement our theoretical results with an extensive experimental analysis showing the practical performance of our solution. Our algorithm achieves an almost $1/2$-approximation. This approximation ratio is tight in the (low-memory) streaming setting [FNFSZ20], but not necessarily in the dynamic setting; a natural question is whether it can be improved, even for insertion-only streams. Another compelling direction for future work is to extend the current result to more general constraints such as matroids.

Broader impact

This work does not present any foreseeable societal consequence.

Acknowledgments and Disclosure of Funding

Slobodan Mitrović was supported by the Swiss NSF grant No. P400P2_191122/1, MIT-IBM Watson AI Lab and Research Collaboration Agreement No. W1771646, and FinTech@CSAIL.

[8]Like our algorithm, SIEVESTREAMING operates parallel copies of the algorithm for different guesses of OPT. We restart only those copies whose solution contains the removed element.

References

[Bac10] Francis R. Bach. Structured sparsity-inducing norms through submodular functions. In *Advances in Neural Information Processing Systems 23: 24th Annual Conference on Neural Information Processing Systems 2010. Proceedings of a meeting held 6-9 December 2010, Vancouver, British Columbia, Canada.*, pages 118–126, 2010.

[BENW16] Rafael da Ponte Barbosa, Alina Ene, Huy L Nguyen, and Justin Ward. A new framework for distributed submodular maximization. In *2016 IEEE 57th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 645–654. Ieee, 2016.

[BFS15] Niv Buchbinder, Moran Feldman, and Roy Schwartz. Online submodular maximization with preemption. In *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015, San Diego, CA, USA, January 4-6, 2015*, pages 1202–1216, 2015.

[BIRB15] Ramakrishna Bairi, Rishabh K. Iyer, Ganesh Ramakrishnan, and Jeff A. Bilmes. Summarization of multi-document topic hierarchies using submodular mixtures. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing of the Asian Federation of Natural Language Processing, ACL 2015, July 26-31, 2015, Beijing, China, Volume 1: Long Papers*, pages 553–563, 2015.

[BMKK14] Ashwinkumar Badanidiyuru, Baharan Mirzasoleiman, Amin Karbasi, and Andreas Krause. Streaming submodular maximization: Massive data summarization on the fly. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 671–680. ACM, 2014.

[BS06] Nikhil Bansal and Maxim Sviridenko. The santa claus problem. In *Proceedings of the Thirty-Eighth Annual ACM Symposium on Theory of Computing*, pages 31–40. ACM, 2006.

[CK14] Amit Chakrabarti and Sagar Kale. Submodular maximization meets streaming: Matchings, matroids, and more. In Jon Lee and Jens Vygen, editors, *Integer Programming and Combinatorial Optimization*, pages 210–221, Cham, 2014. Springer International Publishing.

[CNZ16] Jiecao Chen, Huy L. Nguyen, and Qin Zhang. Submodular maximization over sliding windows. *CoRR*, abs/1611.00129, 2016.

[DDK12] Abhimanyu Das, Anirban Dasgupta, and Ravi Kumar. Selecting diverse features via spectral regularization. In *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012. Proceedings of a meeting held December 3-6, 2012, Lake Tahoe, Nevada, United States.*, pages 1592–1600, 2012.

[DJR12] R. Dey, Z. Jelveh, and K. Ross. Facebook users have become much more private: A large-scale study. In *2012 IEEE International Conference on Pervasive Computing and Communications Workshops*, pages 346–352, 2012.

[DK11] Abhimanyu Das and David Kempe. Submodular meets spectral: Greedy algorithms for subset selection, sparse approximation and dictionary selection. In *Proceedings of the 28th International Conference on Machine Learning, ICML 2011, Bellevue, Washington, USA, June 28 - July 2, 2011*, pages 1057–1064, 2011.

[ELVZ17] Alessandro Epasto, Silvio Lattanzi, Sergei Vassilvitskii, and Morteza Zadimoghaddam. Submodular optimization over sliding windows. In *Proceedings of the 26th International Conference on World Wide Web, WWW 2017, Perth, Australia, April 3-7, 2017*, pages 421–430, 2017.

[ENV19] Alina Ene, Huy L Nguyen, and Adrian Vladu. Submodular maximization with matroid and packing constraints in parallel. In *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing*, pages 90–101, 2019.

[FKK18] Moran Feldman, Amin Karbasi, and Ehsan Kazemi. Do less, get more: Streaming submodular maximization with subsampling. *CoRR*, abs/1802.07098, 2018.

[FMZ19] Matthew Fahrbach, Vahab S. Mirrokni, and Morteza Zadimoghaddam. Submodular maximization with nearly optimal approximation, adaptivity and query complexity. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2019, San Diego, California, USA, January 6-9, 2019*, pages 255–273, 2019.

[FNFSZ20] Moran Feldman, Ashkan Norouzi-Fard, Ola Svensson, and Rico Zenklusen. The one-way communication complexity of submodular maximization with applications to streaming and robustness. *STOC*, 2020.

[GK11] Daniel Golovin and Andreas Krause. Adaptive submodularity: Theory and applications in active learning and stochastic optimization. *J. Artif. Intell. Res.*, 42:427–486, 2011.

[KZK18] Ehsan Kazemi, Morteza Zadimoghaddam, and Amin Karbasi. Scalable deletion-robust submodular maximization: Data summarization with privacy and fairness constraints. In *Proceedings of the*

35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018, pages 2549–2558, 2018.

[LK15] Jure Leskovec and Andrej Krevl. {SNAP Datasets}:{Stanford} large network dataset collection. 2015.

[MBN$^+$17] Slobodan Mitrović, Ilija Bogunovic, Ashkan Norouzi-Fard, Jakub Tarnawski, and Volkan Cevher. Streaming robust submodular maximization: A partitioned thresholding approach. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, 4-9 December 2017, Long Beach, CA, USA*, pages 4560–4569, 2017.

[MKBK15] Baharan Mirzasoleiman, Amin Karbasi, Ashwinkumar Badanidiyuru, and Andreas Krause. Distributed submodular cover: Succinctly summarizing massive data. In *Advances in Neural Information Processing Systems*, pages 2881–2889, 2015.

[MZ15] Vahab Mirrokni and Morteza Zadimoghaddam. Randomized composable core-sets for distributed submodular maximization. In *Proceedings of the forty-seventh annual ACM symposium on Theory of computing*, pages 153–162, 2015.

[NTM$^+$18] Ashkan Norouzi-Fard, Jakub Tarnawski, Slobodan Mitrovic, Amir Zandieh, Aidasadat Mousavifar, and Ola Svensson. Beyond 1/2-approximation for submodular maximization on massive data streams. In *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, pages 3826–3835, 2018.

[NWF78] George L Nemhauser, Laurence A Wolsey, and Marshall L Fisher. An analysis of approximations for maximizing submodular set functions—i. *Mathematical Programming*, 14(1):265–294, 1978.

[ZJCP14] Jingjing Zheng, Zhuolin Jiang, Rama Chellappa, and P. Jonathon Phillips. Submodular attribute selection for action recognition in video. In *Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems 2014, December 8-13 2014, Montreal, Quebec, Canada*, pages 1341–1349, 2014.

A Query complexity of our algorithm

We now state two invariants that are maintained by our algorithms. We will then use these invariants to analyze the oracle-query complexity of our algorithms.

Invariant 1 *For any ℓ, it holds that $|A_\ell| \le (R+1) \cdot 2^{T-\ell}$.*[9] *Here we overload notation A_ℓ to denote $\cup_{i=1}^{R} A_{i,\ell}$.*

Invariant 2 *For any ℓ, it holds that $|B_\ell| \le 2^{T-\ell}$.*

Lemma A.1 *If Invariants 1 and 2 hold before invoking* LEVEL-CONSTRUCT *(Algorithm 5), then they also hold after executing Line 13 of* LEVEL-CONSTRUCT. *Consequently, the invariants hold after* LEVEL-CONSTRUCT *terminates.*

Proof. First, notice that the invocation of LEVEL-CONSTRUCT(ℓ') never adds new elements to any B_ℓ, hence the claim holds for Invariant 2. Now we prove the claim for Invariant 1.

Notice that LEVEL-CONSTRUCT(ℓ') only potentially increases the elements of level ℓ', and the only change for the rest is a reset to the empty set. Therefore we only focus on showing the invariant for on $A_{i,\ell'}$. When LEVEL-CONSTRUCT(ℓ') is invoked, it iterates over all $i = 0 \ldots R$, and for each of them invokes BUCKET-CONSTRUCT on Line 8. By the definition, BUCKET-CONSTRUCT(i', ℓ') increments $S_{i',\ell'}$ and reduces $A_{i',\ell'}$ until the size of $A_{i',\ell'}$ becomes less than $2^{T-\ell'}$ or until $|S_{\mathrm{pred}(R,T)}| \ge k$ (see Line 6 of BUCKET-CONSTRUCT). After this invocation of BUCKET-CONSTRUCT the set $A_{i',\ell'}$ is not changed anymore. There are now two cases, depending on which of the two conditions on Line 6 of BUCKET-CONSTRUCT is false.

Case $|S_{\mathrm{pred}(R,T)}| \ge k$. In this case, each $A_{i,\ell'}$ is set to be the empty set (Line 11 of LEVEL-CONSTRUCT), and hence the claim follows directly.

Case $|A_{i,\ell'}| < 2^{T-\ell'}$. In this case, the size of $A_{i,\ell'}$ remains at most $2^{T-\ell'}$ throughout the rest of the execution. Since this holds for each $j = 0 \ldots R$ for which $|A_{j,\ell'}| < 2^{T-\ell'}$, after the loop on Line 2 terminates we have that $|A_{\ell'}| \le (R+1)2^{T-\ell'}$. □

Lemma A.2 *If Invariants 1 and 2 hold before invoking* INSERTION *(Algorithm 3), then they also hold after* INSERTION *terminates.*

Proof. Observe that Line 1 of INSERTION changes only sets B_ℓ. Hence, if Line 3 evaluates to false, then the two invariants still hold. Before we analyze the case when Line 3 evaluates to true, we first show that there does not exist ℓ' such that $|B_{\ell'}| > 2^{T-\ell'}$. Observe that INSERTION is the only function that adds elements to $B_{\ell'}$.

Towards a contradiction, assume that there is an invocation of INSERTION where for some ℓ' it holds that $|B_{\ell'}| > 2^{T-\ell'} \ge 1$. Let that be the c-th invocation of INSERTION. Since in each invocation of INSERTION the size of $B_{\ell'}$ increases by at most 1, it means that in the $(c-1)$-st invocation of INSERTION it holds that $|B_{\ell'}| \ge 2^{T-\ell'}$. Hence, Line 3 evaluates to true in that invocation. So, by the choice of $\ell^\star$ (see Line 4 of INSERTION) it holds that $\ell^\star \le \ell'$. But this now implies that Line 6 of INSERTION sets $B_{\ell'}$ to be the empty set. Hence, after Line 3 of INSERTION evaluates to true in the $(c-1)$-st invocation, the size of $B_{\ell'}$ in the c-th invocation is at most 1. This contradicts our assumption.

This now implies that when LEVEL-CONSTRUCT is invoked, the two invariants hold. Hence, by Lemma A.1 these two invariants also hold after the execution of LEVEL-CONSTRUCT invoked on Line 7 of INSERTION, and consequently hold after the execution of INSERTION. □

Lemma A.3 *If Invariants 1 and 2 hold before invoking* DELETION *(Algorithm 4), then they also hold after* DELETION *terminates.*

[9]Recall that $T = \log n$ (see Line 3 of INITIALIZATION.)

Proof. On Lines 1 and 2 DELETION removes some elements from $A_{i,\ell}$ and B_ℓ. So, these steps maintain Invariants 1 and 2. The rest of the changes of the sets $A_{i,\ell}$ and B_ℓ is done through invocation of LEVEL-CONSTRUCT on Line 9. The proof now follows by Lemma A.1. □

A.1 Oracle-query Complexity

Lemma A.4 (LEVEL-CONSTRUCT Complexity) LEVEL-CONSTRUCT(ℓ) *performs* $O(R^4 \cdot 2^{T-\ell}/\epsilon_p)$ *oracle queries in expectation.*

Proof. Note that all the oracle queries performed by LEVEL-CONSTRUCT are via invocations of BUCKET-CONSTRUCT. Hence, we first analyze the oracle-query complexity of BUCKET-CONSTRUCT. We begin by bounding $|A_{i,\ell}|$ during an execution of LEVEL-CONSTRUCT(ℓ).

By the invariants, from Lines 4 and 6 of LEVEL-CONSTRUCT we have that for each i and if $R \geq 2$ it holds that

$$\begin{aligned} |A_{i,\ell}| &\leq |B_{\ell-1}| + \sum_{j=0}^{R} |A_{j,\ell-1}| \\ &\leq 2^{T-\ell+1} + (R+1) \cdot 2^{T-\ell+1} \\ &\leq 4 \cdot R \cdot 2^{T-\ell}. \end{aligned} \tag{1}$$

Oracle queries of BUCKET-CONSTRUCT(i, ℓ). For each i, LEVEL-CONSTRUCT invokes BUCKET-CONSTRUCT on Line 8. Line 2 of BUCKET-CONSTRUCT performs at most $|A_{i,\ell}|$ oracle queries. By Lemma C.4, PEELING invoked on Line 4 requires at most $c \cdot \log^2 k$ oracle queries, for some absolute constant c. Furthermore, from Item 4 of Lemma C.4 and our bound (1), BUCKET-CONSTRUCT in expectation executes at most $c_1 \cdot \frac{\log R}{\epsilon_p}$ iterations, for some absolute constant c_1, until $|A_{i,\ell}| < 2^{T-\ell}$. This altogether implies that in expectation BUCKET-CONSTRUCT(i, ℓ) performs at most

$$c_1 \cdot \tfrac{\log R}{\epsilon_p} R \cdot (|A_{i,\ell}| + c \cdot \log^2 k) \tag{2}$$

oracle queries.

Total number of oracle queries. Let C be the expected number of oracle queries performed by LEVEL-CONSTRUCT(ℓ). Given that LEVEL-CONSTRUCT(ℓ) might recursively invoke LEVEL-CONSTRUCT(ℓ') for all $\ell < \ell' \leq T$, from our bounds above we have

$$\begin{aligned} C &\overset{\text{by (2)}}{\leq} \sum_{\ell'=\ell}^{T} \sum_{i=0}^{R} c_1 \cdot \tfrac{\log R}{\epsilon_p} \cdot (|A_{i,\ell}| + c \cdot \log^2 k) \\ &\overset{\text{by (1)}}{\leq} c_1 (R+1) \tfrac{\log R}{\epsilon_p} \cdot \sum_{\ell'=\ell}^{T} (4 \cdot R \cdot 2^{T-\ell} + c \cdot \log^2 k). \end{aligned}$$

Using that $\sum_{\ell'=\ell}^{T} 2^{T-\ell} \leq 2 \cdot 2^{T-\ell}$, that $R \geq \log k$, and also that $(T-\ell+1) \cdot c \cdot \log^2 k \leq 2^{T-\ell} \cdot c \cdot \log^2 k$, from the last chain of inequalities we further derive

$$C = O(R^4 \cdot 2^{T-\ell}/\epsilon_p),$$

as desired. □

Lemma A.5 (Amortized complexity per deletion) *The amortized number of oracle queries per deletion is* $O\left(\frac{R^5 \log^2(n)}{\epsilon_p^2 \cdot \varepsilon}\right)$ *in expectation.*

We first give an intuition of why this lemma holds, and then provide a formal proof. Let us concentrate on a single bucket (i, ℓ). By construction, any element added to $S_{i,\ell}$ is added from a set $A_{i,\ell}$ of size at least $2^{T-\ell}$. This happens in every iteration inside BUCKET-CONSTRUCT, of which there are $O(\log(n)/\epsilon_p)$ many with high probability because of Property 4 of Lemma C.4. Therefore, any given element to be removed is not very likely to have been in the solution set $S_{i,\ell}$. Since a recomputation is triggered once an ε fraction of that set is removed (see Line 7), it is required to remove a large $(\Omega(\epsilon_p\varepsilon/\log(n)))$ fraction of elements of $A_{i,\ell}$ (which has size at least $2^{T-\ell}$) in expectation before this happens. Once recomputation is triggered, it costs $O(2^{T-\ell}\cdot R^4/\epsilon_p)$ oracle queries by Lemma A.4. So, the overall amortized oracle-query complexity per deletion is $O\left(\frac{R^4\log(n)}{\epsilon_p^2\cdot\varepsilon}\right)$. We conclude by summing up these contribution over all buckets (i,ℓ).

Proof. We now formalize these arguments. Fix (i,ℓ). We will analyze the way $S_{i,\ell}$ is constructed by BUCKET-CONSTRUCT(i,ℓ) from $A_{i,\ell}$. This is done iteratively within the loop on Line 1. Let I be the number of iterations performed by BUCKET-CONSTRUCT. Denote by $A^t_{i,\ell}$ the set $A_{i,\ell}$ and by $S^t_{i,\ell}$ the set $S_{i,\ell}$ at the end of the t-th iteration, for $t=1,2,...,I$. Let $S^0_{i,\ell}\overset{\text{def}}{=}\emptyset$. Assume that $|A^1_{i,\ell}|\geq 2^{T-\ell}$, as otherwise $S_{i,\ell}$ is empty and hence no deletion affects $S_{i,\ell}$.

First we argue that $I\leq 32\log(n)/\epsilon_p$ with high probability. Let F_t be the fraction of elements not filtered away in the t-th iteration (if $t>I$, set $F_t=0$). Property 4 of Lemma C.4 implies that F_t is at most $1-\epsilon_p/8$ in expectation (regardless of the state before the t-th iteration). Thus we have

$$\mathbb{E}\left[F_1\cdot F_2\cdot\ldots\cdot F_{32\log(n)/\epsilon_p}\right]\leq(1-\epsilon_p/8)^{4\log(n)\cdot 8/\epsilon_p}\approx(1-\epsilon_p/8)^{4\log_{1-\epsilon_p/8}(n)}=n^{-4}$$

and by Markov's inequality, the probability that $I>32\log(n)/\epsilon_p$, for which it is necessary that $F_1\cdot F_2\cdot\ldots\cdot F_{32\log(n)/\epsilon_p}\geq 1/|A^1_{i,\ell}|$, is at most $|A^1_{i,\ell}|\cdot n^{-4}\leq n^{-3}$.

In the rest of the proof, we will show that in expectation it is needed to remove $\frac{\epsilon_p\varepsilon 2^{T-\ell}}{132\log(n)}$ elements from $A^1_{i,\ell}$ in order for Line 7 of DELETION to become true.

Define $D^t\overset{\text{def}}{=}S^t_{i,\ell}\setminus S^{t-1}_{i,\ell}$, for each $1\leq i\leq I$. The set D^t is obtained on Line 4 of BUCKET-CONSTRUCT by invoking PEELING on $A^t_{i,\ell}$. By Lemma C.4, this set is a subset of $A^t_{i,\ell}$ of cardinality $|D^t|$ chosen uniformly at random. Observe that $A^t_{i,\ell}$ depends on the choice of $S^{t-1}_{i,\ell}$; in particular, $A^t_{i,\ell}\cap S^{t-1}_{i,\ell}=\emptyset$ as long as $\tau_i>0$. Nevertheless, the randomness used by PEELING to obtain D^t from $A^t_{i,\ell}$ does not depend on the choice of $S^{t-1}_{i,\ell}$. For $r=1,2,...$, let X_r be the chronologically first r elements removed from $A^1_{i,\ell}$. Then

$$\mathbb{E}\left[\frac{|D^t\cap X_r|}{|D^t|}\;\middle|\;A^t_{i,\ell}\right]=\frac{|A^t_{i,\ell}\cap X_r|}{|A^t_{i,\ell}|}\leq\frac{r}{2^{T-\ell}}$$

and thus

$$\mathbb{E}\left[\frac{|D^t\cap X_r|}{|D^t|}\right]\leq\frac{r}{2^{T-\ell}}\,.$$

We write

$$\mathbb{E}\left[\frac{|X_r\cap S_{i,\ell}|}{|S_{i,\ell}|}\right]=\mathbb{E}\left[\frac{|X_r\cap S_{i,\ell}|}{|S_{i,\ell}|}\;\middle|\;I\leq\frac{32\log(n)}{\epsilon_p}\right]\cdot\mathbb{P}\left[I\leq\frac{32\log(n)}{\epsilon_p}\right]$$
$$+\underbrace{\mathbb{E}\left[\frac{|X_r\cap S_{i,\ell}|}{|S_{i,\ell}|}\;\middle|\;I>\frac{32\log(n)}{\epsilon_p}\right]}_{\leq 1}\cdot\underbrace{\mathbb{P}\left[I>\frac{32\log(n)}{\epsilon_p}\right]}_{\leq n^{-3}}\,.$$

Let us adopt the convention that $D^t = \emptyset$ and $\frac{|D^t \cap X_r|}{|D^t|} = 0$ for $t > I$. Then

$$\begin{aligned}
\mathbb{E}\left[\frac{|X_r \cap S_{i,\ell}|}{|S_{i,\ell}|} \;\middle|\; I \le \frac{32\log(n)}{\epsilon_p}\right] &= \mathbb{E}\left[\sum_{t=1}^{I} \frac{|X_r \cap D^t|}{|S_{i,\ell}|} \;\middle|\; I \le \frac{32\log(n)}{\epsilon_p}\right] \\
&\le \sum_{t=1}^{32\log(n)/\epsilon_p} \mathbb{E}\left[\frac{|X_r \cap D^t|}{|D^t|} \;\middle|\; I \le \frac{32\log(n)}{\epsilon_p}\right] \\
&\le \sum_{t=1}^{32\log(n)/\epsilon_p} \mathbb{E}\left[\frac{|X_r \cap D^t|}{|D^t|}\right] \cdot \mathbb{P}\left[I \le \frac{32\log(n)}{\epsilon_p}\right]^{-1} \\
&\le \frac{32\log(n)}{\epsilon_p} \cdot \frac{r}{2^{T-\ell}} \cdot \mathbb{P}\left[I \le \frac{32\log(n)}{\epsilon_p}\right]^{-1}
\end{aligned}$$

(for the second inequality we used the simple fact that $\mathbb{E}[X \mid A] \le \mathbb{E}[X] / \mathbb{P}[A]$ for $X \ge 0$). In the end we get

$$\mathbb{E}\left[\frac{|X_r \cap S_{i,\ell}|}{|S_{i,\ell}|}\right] \le \frac{32\log(n)}{\epsilon_p} \cdot \frac{r}{2^{T-\ell}} + n^{-3} .$$

Let D denote the number of elements removed before Line 7 of DELETION evaluates to true. We have

$$\begin{aligned}
\mathbb{E}[D] &= \sum_{r=0}^{\infty} \mathbb{P}[D > r] \\
&\ge \sum_{r=1}^{\frac{\varepsilon 2^{T-\ell}}{66\log(n)/\epsilon_p}} \mathbb{P}\left[\frac{|X_r \cap S_{i,\ell}|}{|S_{i,\ell}|} < \varepsilon\right] \\
&\ge \sum_{r=1}^{\frac{\varepsilon 2^{T-\ell}}{66\log(n)/\epsilon_p}} \underbrace{\left(1 - \frac{32\log(n)/\epsilon_p \cdot \frac{r}{2^{T-\ell}} + n^{-3}}{\varepsilon}\right)}_{\ge 1 - \frac{33\log(n)/\epsilon_p \cdot \frac{r}{2^{T-\ell}}}{\varepsilon} \ge 1/2} \\
&\ge \frac{\epsilon_p \varepsilon 2^{T-\ell}}{132\log(n)} ,
\end{aligned}$$

where the second-last inequality follows from Markov's inequality. Finally, a recomputation costs $O(2^{T-\ell} \cdot R^4/\epsilon_p)$ oracle queries by Lemma A.4. This is in expectation over the randomness used in the recomputation, which is independent from the randomness used to determine D (so we can compute the expectation of the ratio using the ratio of the expectations). Thus the expected amortized cost per deleted element is $O\left(\frac{R^4\log(n)}{\epsilon_p^2\varepsilon}\right)$.

We obtain the final bound by summing up the contributions of all RT buckets to this amortized expected recomputation cost. □

Lemma A.6 (Amortized complexity per insertion) *The amortized number of oracle queries per insertion is $O(T \cdot R^4/\epsilon_p)$ in expectation.*

Proof. When an element e is inserted, it is added to B_ℓ for all $0 \le \ell \le T$ (see Line 1 of INSERTION). Assume that after adding e some sets B_ℓ become "too large", i.e., $|B_\ell| \ge 2^{T-\ell}$ (see Line 3). Let $\ell^\star$ be the smallest such ℓ. Then, INSERTION invokes LEVEL-CONSTRUCT($\ell^\star$) on Line 7. By Lemma A.4, this invocation requires $O(2^{T-\ell^\star} \cdot R^4/\epsilon_p)$ oracle queries in expectation. Also, during this invocation the set $B_{\ell^\star}$ is set to be the empty set (see Line 1 of LEVEL-CONSTRUCT($\ell^\star$)). Moreover, at the beginning of the algorithm $B_{\ell^\star}$ was empty and is augmented only by INSERTION. This altogether means that $2^{T-\ell^\star}$ elements have to be added to $B_{\ell^\star}$ in order for INSERTION to invoke this execution of LEVEL-CONSTRUCT. This implies that per one element added to $B_{\ell^\star}$, INSERTION uses $O(R^4/\epsilon_p)$

oracle queries. Moreover, an element is added to T different sets B_ℓ. Therefore, across all ℓ, INSERTION in expectation spends $O(T \cdot R^4/\epsilon_p)$ oracle queries per one inserted element. □

Together, Lemmas A.5 and A.6 imply the following result.

Theorem 5.2 *The amortized expected number of oracle queries per update is* $O\left(\frac{R^5 \log^2(n)}{\epsilon_p^2 \cdot \varepsilon}\right)$, *where* R *equals* $\log_{1+\epsilon_1}(2k)$ *(see Algorithm 1).*

B Correctness of our algorithm

Let us start by introducing a key property of our algorithm. Throughout this section, in case $\ell = 0$, by $S_{\text{pred}(r,\ell-1)}$ we denote $S_{\text{pred}(r-1,T)}$ and we define $S_{\text{pred}(-1,T)} = \emptyset$.

Observation B.1 *The only time when elements are added to one of the sets* $S_{i,\ell}$ *is Line 4 of Algorithm 2. Moreover, all the sets* $S_{\cdot,\cdot}$ *after* $S_{i,\ell}$ *are empty, i.e.,*

$$\bigcup_{0\le j\le R, \ell<r\le T} S_{j,r} \cup \bigcup_{i+1\le j\le R} S_{j,\ell} = \emptyset.$$

Proof. This follows from the fact that we empty all the abovementioned sets in Line 5 of Algorithm 3 and Line 8 of Algorithm 4 before calling LEVEL-CONSTRUCT, and those are the only lines that BUCKET-CONSTRUCT is called from. □

Theorem 5.1 *Let* Sol_i *be the solution of our algorithm and* OPT_i *be the optimal solution after* i *updates. Moreover, assume that* γ *in Algorithm 1 is such that* $(1+\epsilon_p)\,\mathrm{OPT}_i \ge \gamma \ge \mathrm{OPT}_i$. *Then for any* $1 \le i \le n$ *we have* $\mathbb{E}[f(Sol_i)] \ge (1-\epsilon_p-\epsilon(1+\epsilon_1))\frac{\mathrm{OPT}_i}{2}$.

Proof. Consider the last time j when LEVEL-CONSTRUCT(ℓ) is called for some value ℓ. Let us start by analysing $f(\mathrm{Sol}_j)$. Consider the following two cases depending on the size of $S_{\text{pred}(R,T)}$ at this moment, i.e., after the j-th operation:

- If $|S_{\text{pred}(R,T)}| = k$: then by Item 3 of Lemma C.4 any set of elements that are added to one of the sets S gives marginal contribution of τ_i per element to the previous elements of S

  $$\mathbb{E}[f(S_{i,\ell}|S_{\text{pred}(i,\ell-1)}) \ge (1-\epsilon_p)\tau_i|S_{i,\ell}| \ge (1-\epsilon_p)\frac{\gamma}{2k}|S_{i,\ell}|.$$

  Therefore since in this case $|\mathrm{Sol}_j| = k$, and by Observation B.1, by linearity of expectation we get:

  $$\mathbb{E}[f(\mathrm{Sol}_j)] \ge k\cdot(1-\epsilon_p)\frac{\gamma}{2k} \ge (1-\epsilon_p)\frac{\mathrm{OPT}_i}{2}. \tag{3}$$

- If $|S_{\text{pred}(R,T)}| < k$: The goal in this case is to show that for any element $e \in O_j$, $f(e|S_{\text{pred}(R,T)}) < \frac{\gamma}{2k}$. To that end, consider any element $e \in O_j$. Consider the LEVEL-CONSTRUCT(ℓ') with the lowest ℓ' that is called after inserting this element[10]. In Line 4 or Line 6 of the LEVEL-CONSTRUCT algorithm, e will be added to some sets in A. Moreover, in this case, LEVEL-CONSTRUCT(T) must be called, since this is the only way how the condition in Line 13 can be false. This also results in calling BUCKET-CONSTRUCT(r, T) for all $1 \le r \le R$ in Line 2 of Algorithm 5. Notice that BUCKET-CONSTRUCT(r, T) stops only if $|A_{r,T}| = 0$ (in Line 6). Therefore all the elements have been removed from the sets A at some point. Also, the only time when we remove elements from A is in Line 2, which combined with submodularity shows that for any element $e \in O_j$ we have

  $$f(e|S_{\text{pred}(R,T)}) \le \tau_R = \frac{\gamma}{2k}.$$

[10]This might not happen immediately after the insertion.

By definition, $\mathrm{Sol}_j = S_{\mathrm{pred}(R,T)}$, which results in

$$f(e|\mathrm{Sol}_j) \leq \frac{\gamma}{2k}.$$

Applying the above inequality for all the elements $e \in O_j$ along with submodularity, we get that

$$f(O_j|\mathrm{Sol}_j) \leq k \cdot \frac{\gamma}{2k} \leq \frac{\gamma}{2}.$$

Moreover, by submodularity and monotonicity, we have that

$$f(O_j) \leq f(\mathrm{Sol}_j) + f(O_j|\mathrm{Sol}_j).$$

Combining the above two inequalities we get that

$$f(\mathrm{Sol}_j) \geq \mathrm{OPT}_j - \frac{\gamma}{2} \geq \mathrm{OPT}_j - \frac{1+\epsilon_p}{2}\,\mathrm{OPT}_i \geq \frac{1-\epsilon_p}{2}\,\mathrm{OPT}_i\,, \tag{4}$$

where the last two inequalities follow by the theorem's assumption and the fact that $\mathrm{OPT}_j \geq \mathrm{OPT}_i$, respectively. Notice that $\mathrm{OPT}_j \geq \mathrm{OPT}_i$ since there are no insertions after the last call to LEVEL-CONSTRUCT.

By (3) and (4) we get that

$$\mathbb{E}[f(\mathrm{Sol}_j)] \geq (1-\epsilon_p)\frac{\mathrm{OPT}_i}{2}. \tag{5}$$

Let us now complete the proof by showing that $f(\mathrm{Sol}_i) \geq (1 - \frac{\epsilon(1+\epsilon_1)}{1-\epsilon_p})f(\mathrm{Sol}_j)$ for $i > j$. Notice that by the assumption that $\gamma \geq \mathrm{OPT}$ there is no element e with $f(e) > \gamma$. Therefore all the elements on any layer will belong to one of the buckets. Consider any $S_{r,\ell}$ ($1 \leq r \leq R$, $0 \leq \ell \leq T$); we know that when LEVEL-CONSTRUCT(ℓ) has been called, we have $E[f(S_{r,\ell}|S_{\mathrm{pred}(r,\ell-1)})] \geq (1-\epsilon_p)\tau_r|S_{r,\ell}|$.[11] Moreover, at most an ϵ-fraction of its elements can be removed. By $S'_{r,\ell}$ we denote this set after these deletions. We know that the marginal contribution of each element in $S_{r,\ell}$ with respect to $S_{\mathrm{pred}(r,\ell-1)}$ is at most $\tau_{i-1} = (1+\epsilon_1)\tau_i$. By submodularity we get that

$$\begin{aligned} f(S'_{r,\ell}|S_{\mathrm{pred}(r,\ell-1)}) &\geq f(S_{r,\ell}|S_{\mathrm{pred}(r,\ell-1)}) - |S_{r,\ell}|\epsilon(1+\epsilon_1)\tau_i f(S'_{r,\ell}|S_{\mathrm{pred}(r,\ell-1)}) \\ &\geq f(S_{r,\ell}|S_{\mathrm{pred}(r,\ell-1)})(1 - \frac{\epsilon(1+\epsilon_1)}{1-\epsilon_p}). \end{aligned}$$

Now for all r and ℓ, let $S'_{\mathrm{pred}(r,\ell)}$ denote the set $S_{\mathrm{pred}(r,\ell)}$ after the i-th operation, i.e., after applying the deletions.

Considering that f is submodular, and $S'_{\mathrm{pred}(r,\ell)} \subseteq S_{\mathrm{pred}(r,\ell)}$, we get

$$f(S'_{r,\ell}|S'_{\mathrm{pred}(r,\ell-1)}) \geq f(S_{r,\ell}|S_{\mathrm{pred}(r,\ell-1)})(1 - \frac{\epsilon(1+\epsilon_1)}{1-\epsilon_p}).$$

By adding up the above marginal values over r and ℓ, we get

$$f(S'_{\mathrm{pred}(R,T)}) \geq (1 - \frac{\epsilon(1+\epsilon_1)}{1-\epsilon_p})f(S_{\mathrm{pred}(R,T)}).$$

So $f(\mathrm{Sol}_i) \geq (1 - (1 - \frac{\epsilon(1+\epsilon_1)}{1-\epsilon_p}))f(\mathrm{Sol}_j)$. This along with Eq. (5) concludes the proof:

$$f(\mathrm{Sol}_i) \geq (1 - \frac{\epsilon(1+\epsilon_1)}{1-\epsilon_p})f(\mathrm{Sol}_j),$$

$$\mathbb{E}[f(\mathrm{Sol}_i)] \geq (1 - \epsilon_p - \epsilon(1+\epsilon_1))\frac{\mathrm{OPT}_i}{2}.$$

□

[11] Recall that, in case $\ell = 0$, by $S_{\mathrm{pred}(r,\ell-1)}$ we denote $S_{\mathrm{pred}(r-1,T)}$. Moreover, we let $S_{\mathrm{pred}(-1,T)} = \emptyset$.

C Peeling algorithm

For completeness, in this section we present the routine PEELING, which is part of the algorithm THRESHOLD-SAMPLING in [FMZ19]. All the lemmas and algorithms in this section have been introduced in [FMZ19] and we provide them here for completeness. Before presenting PEELING, we need to define a distribution. Let $N \subseteq V$ be some set of elements.

Definition C.1 *Conditioned on the current state of the algorithm, consider the process where first a set $S \sim \mathcal{U}(N, s)$ (size-s subset of N) and then an element $x \sim N \setminus S$ are drawn uniformly at random. Let $\mathcal{D}_s$ denote the probability distribution over the indicator random variable*

$$I_s = \mathbb{1}[f(x \mid S) \geq \tau].$$

Algorithm 6 PEELING

```
Input: Subset of items $N \subseteq V$, function $f : 2^N \to R^{\geq 0}$, constraint $k$, threshold $\tau$, error $\epsilon$
1: Set smaller error $\hat{\epsilon} \leftarrow \epsilon/4$
2: Set $m \leftarrow \lceil \log(k)/\hat{\epsilon} \rceil$
3: for $i = 0$ to $m$ do
4:     Set $s \leftarrow \min\{\lfloor (1+\hat{\epsilon})^i \rfloor, |N|\}$
5:     if REDUCED-MEAN($\mathcal{D}_s, \hat{\epsilon}$) then
6:         break
7:     end if
8: end for
9: Sample $S \sim \mathcal{U}(N, \min\{s, k\})$
10: return $S$
```

We briefly remark that the REDUCED-MEAN subroutine is a standard unbiased estimator for the mean of a Bernoulli distribution. Since $\mathcal{D}_t$ is a uniform distribution over indicator random variables, it is in fact a Bernoulli distribution. The guarantees of in Lemma C.2 are consequences of Chernoff bounds [BS06].

Algorithm 7 REDUCED-MEAN

```
Input: access to a Bernoulli distribution $\mathcal{D}$, error $\hat{\epsilon}$
1: Set failure probability $\delta \leftarrow 2\hat{\epsilon}^2/(k\log(k))$
2: Set number of samples $m \leftarrow 16\lceil \log(2/\delta)/\hat{\epsilon}^2 \rceil$
3: Sample $X_1, X_2, \ldots, X_m \sim \mathcal{D}$
4: Set $\overline{\mu} \leftarrow \frac{1}{m}\sum_{i=1}^{m} X_i$
5: if $\overline{\mu} \leq 1 - 1.5\hat{\epsilon}$ then
6:     return true
7: end if
8: return false
```

Lemma C.2 *For any Bernoulli distribution $\mathcal{D}$,* REDUCED-MEAN *uses $O(\log(\delta^{-1})/\hat{\epsilon}^2)$ samples to correctly report one of the following properties with probability at least $1-\delta$:*

1. *If the output is* `true`*, then the mean of $\mathcal{D}$ is $\mu \leq 1 - \hat{\epsilon}$.*
2. *If the output is* `false`*, then the mean of $\mathcal{D}$ is $\mu \geq 1 - 2\hat{\epsilon}$.*

Here δ is set to be $2\hat{\epsilon}^2/(k\log(k))$ in the algorithm REDUCED-MEAN.

Proof. By construction, the number of samples is $m = 16\lceil \log(2/\delta)/\hat{\epsilon}^2 \rceil$. To show the correctness of REDUCED-MEAN, it suffices to prove that $\Pr(|\overline{\mu} - \mu| \geq \hat{\epsilon}/2) \leq \delta$. Letting $X = \sum_{i=1}^{m} X_i$, this is equivalent to

$$\Pr\left(|X - m\mu| \geq \frac{\hat{\epsilon}m}{2}\right) \leq \delta.$$

Using the Chernoff bounds in Lemma C.3 and a union bound, for any $a > 0$ we have

$$\Pr(|X - m\mu| \geq a) \leq e^{-\frac{a^2}{2m\mu}} + e^{-a\min\left(\frac{1}{5}, \frac{a}{4m\mu}\right)}.$$

Let $a = \hat{\epsilon}m/2$ and consider the exponents of the two terms separately. Since $\mu \leq 1$, we bound the left term by

$$\frac{a^2}{2m\mu} = \frac{\hat{\epsilon}^2 m^2}{8m\mu} \geq \frac{\hat{\epsilon}^2}{8\mu} \cdot \frac{16\log(2/\delta)}{\hat{\epsilon}^2} \geq \log(2/\delta).$$

For the second term, first consider the case when $1/5 \leq a/(4m\mu)$. For any $\hat{\epsilon} \leq 1$, it follows that

$$a\min\left(\frac{1}{5}, \frac{a}{4m\mu}\right) = \frac{1}{5} \geq \frac{\hat{\epsilon}}{10} \cdot \frac{16\log(2/\delta)}{\hat{\epsilon}^2} \geq \log(2/\delta).$$

Otherwise, we have $a/(4m\mu) \leq 1/5$, and by the previous analysis we have $a^2/(4m\mu) \geq \log(2\delta)$. Therefore, in all cases we have

$$\Pr\left(|X - m\mu| \geq \frac{\hat{\epsilon}m}{2}\right) \leq 2e^{-\log(2/\delta)} = \delta,$$

which completes the proof. □

Lemma C.3 (Chernoff bounds, [BS06]) *Suppose $X_1, \ldots, X_n$ are binary random variables such that $\Pr(X_i = 1) = p_i$. Let $\mu = \sum_{i=1}^n p_i$ and $X = \sum_{i=1}^n X_i$. Then for any $a > 0$, we have*

$$\Pr(X - \mu \geq a) \leq e^{-a\min\left(\frac{1}{5}, \frac{a}{4\mu}\right)}.$$

Moreover, for any $a > 0$, we have

$$\Pr(X - \mu \leq -a) \leq e^{-\frac{a^2}{2\mu}}.$$

Using the guarantees for REDUCED-MEAN, we can prove:

Lemma C.4 *Let N be a set of elements such that for each $e \in N$ we have $f(e) \geq \tau$. The algorithm* PEELING *outputs a set $S \subseteq N$ with $|S| \leq k$ such that the following properties hold:*

1. *There are $O(\log^2(k))$ oracle queries.*
2. PEELING *finds a size X and returns a uniformly random subset of size X from its input items.*
3. *The expected average marginal satisfies $\mathbb{E}[f(S)/|S|] \geq (1 - \epsilon_p)\tau$.*
4. *If $|S| < k$, then the expected number of items $x \in N$ with $\Delta(x, S) < \tau$ is at least $\epsilon_p|N|/8$.*

Proof.

We prove the upper bound on the query complexity of PEELING. There are $m = O(\log(k))$ runs of REDUCED-MEAN, each of which makes $O(\delta^{-1}) = O(\log(k))$ queries. Therefore the total query complexity of PEELING is $O(\log^2(k))$.

Next, we show the lower bound on the average value of selected items, the set S. PEELING starts by calling REDUCED-MEAN with $s = 1$. We note that in our case the input of PEELING always consists of items with marginal value at least τ. Therefore the first run of REDUCED-MEAN returns false. We call REDUCED-MEAN with different values of s. Let s' be the first time REDUCED-MEAN returns true and s'' be the previous value that we called REDUCED-MEAN with. If REDUCED-MEAN always returns false, we let s'' be the maximum value of s.

We call REDUCED-MEAN $\lceil \log(k)/\hat{\epsilon} \rceil \leq 2\log(k)/\hat{\epsilon}$ times. Using Lemma C.2 and a union bound, we know that with probability at least $1 - 2\delta\log(k)/\hat{\epsilon} \geq 1 - \hat{\epsilon}/k$ for all calls of REDUCED-MEAN we have the two properties of Lemma C.2.

For s'', REDUCED-MEAN returns false, therefore the mean of random variable $\mathcal{D}_{s''}$ is at least $1-2\hat{\epsilon}$. So picking s'' random items yields at least $(1-2\hat{\epsilon})\tau s''$ value. Here we use linearity of expectation, and also use that the expected marginal gain of a randomly chosen element with respect to a random subset $Z \subseteq V$ does not increase with the increase of size of Z. (We refer a reader to Lemma 3.4 and Lemma 3.2 of [FKK18] for a formal proof of this argument.) PEELING returns a random subset of size $X = \min(s,k)$. By definition of s'', we have $X \leq (1+\hat{\epsilon})s''$. Therefore the expected value of the solution set S is at least $\frac{1-2\hat{\epsilon}}{1+\hat{\epsilon}}|S|\tau$. The above statements hold only if we are in the case where REDUCED-MEAN does not fail in any of the calls. Since the failure probability is upper-bounded by $\hat{\epsilon}/k$ and even in the failure case we do not pick more than k items, we can still say that the expected value of the solution is at least $(1-\hat{\epsilon})\frac{1-2\hat{\epsilon}}{1+\hat{\epsilon}}|S|\tau \geq (1-\epsilon)\tau|S|$. The inequality holds because of the way the parameter $\hat{\epsilon}$ is set. This proves the lower bound on the expected value of the solution.

To prove the last property of the lemma, we note that if the solution set S consists of fewer than k items, we know that REDUCED-MEAN has returned true at least once. Denote by s' the first time it does so. In this case, PEELING returns a random set of size s'. We know that REDUCED-MEAN did not fail in any of the calls with probability at least $1-\hat{\epsilon}/k$. Focusing on the case that REDUCED-MEAN does not fail, we know that the mean of $\mathcal{D}_{s'}$ is at most $1-\hat{\epsilon}$. Therefore after picking solution S (e.g., s' random items), the expected number of items with marginal value below τ is at least an $\hat{\epsilon}$-fraction of all input items. Noting that the failure probability is at most $1-\hat{\epsilon}/k \leq 1/2$ proves the last claim of the lemma.

□

C.1 Combining the ingredients

We now recall well-known techniques that can be used to remove the assumptions we made while designing our algorithm. First, we assumed that we have a tight estimate of OPT, e.g., the value of γ in Theorem 5.1. This assumption can be removed by considering geometrically increasing guesses $\gamma = (1+\epsilon_p)^i$ of OPT, and for each of the guesses executing a separate instance of our algorithm. Even though OPT potentially changes from operation to operation, at each point one of the guesses is correct up to a small multiplicative factor. A similar approach was employed in several prior works. After every operation, we return the maximum-value solution over all γ's. Theorem 5.1 shows that, for the γ value that is close to the true optimum value at that time, the instance parametrized by γ returns a solution of high value. Moreover, the number of oracle calls is independent of the value of γ. This results in losing a factor $\log(k\Delta/(\delta\epsilon_p))$ in the number of oracle calls, where Δ, δ denote the value of the elements of maximum and minimum value in the universe, respectively. We do not need to know these two values in advance; we simply compute them on the fly and run parallel copies of the algorithm for the currently relevant guesses of OPT. Moreover, we can again use a simple technique to remove the dependency on $\log(\Delta/\delta)$. Namely, it suffices to add an element e to those copies of the algorithm where $f(e) \leq \gamma \leq \frac{k}{\epsilon_p} f(e)$ since: (i) while e is not deleted it holds $\text{OPT} \geq f(e)$, therefore we do not need to consider copies with $\gamma < f(e)$; and, (ii) all elements with $\gamma \geq \frac{k}{\epsilon_p} f(e)$ contribute at most only $\epsilon_p \gamma$ to the solution of this copy. Therefore, this increases the number of oracle calls by a factor of $\log k/\epsilon_p$, while decreasing the approximation guarantee by $1-\epsilon_p$.

Second, we assumed that we know the length n of the stream, which is used to upper-bound $|V_t|$. We remove this assumption as follows. We maintain an upper-bound $\tilde{n}$ on n. The algorithm is initiated by $\tilde{n} = 1$. If at some point n equals $\tilde{n}$, we restart the algorithm by doubling $\tilde{n}$, i.e, by letting $\tilde{n} \leftarrow 2 \cdot \tilde{n}$, and defining $T = \log \tilde{n}$ in Algorithm 1. This affects the number of oracle calls only by a constant factor, and has no effect on the approximation guarantee.

Theorem 5.3 *Our algorithm maintains a $(1-2\epsilon_p-\epsilon(1+\epsilon_1))/2$-approximate solution after each operation. The amortized expected number of oracle queries per update of this algorithm is* $O\left(\frac{\log^6_{1+\epsilon_1}(k)\log^2(n)}{\epsilon_p^4 \cdot \varepsilon}\right)$.

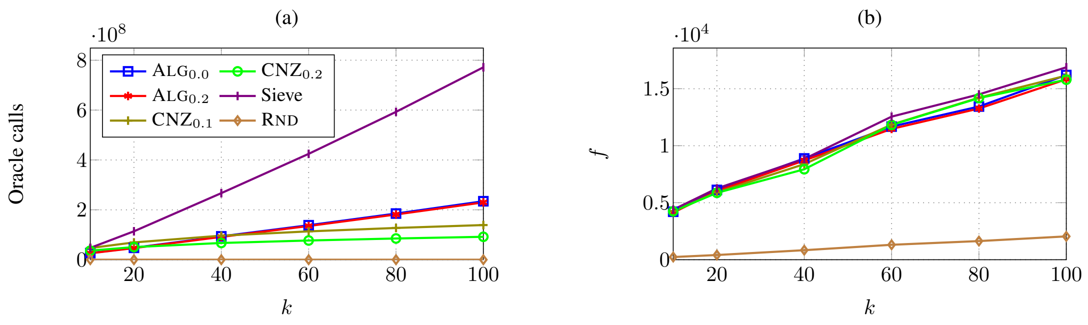

Figure 2: These plots depict results of our experiments run on Twitter network, with the elements presented as an arbitrary ordered stream. The optimization is performed over windows of size $70,000$.

D Additional experiments

In this section, we provide additional experiments to those presented in Fig. 1. The setup we use in this section is the same as in Section 6. We refer a reader to Section 6 for the details of this setup (e.g., definition of f and the type of plots we present).

We perform two types of experiments:

- plotting the number of oracle calls and the values of f with respect to k, while varying k (as the plots in Fig. 1), and
- plotting the number of oracle calls and the values of f for blocks of queries (insertions and deletions) for a *fixed* k (as the plots in Fig. 9).

Plots for varying k. We perform the following experiments:

- **ego-Twitter:** In Fig. 2 we depict the result of our experiments performed on the ego-Twitter network where the nodes are processed as arbitrary ordered stream. We perform optimization over windows of size $70,000$. Recall that for this dataset $|V| = 81,306$.
- **Pokec:** The plots in Figs. 3 and 4 are obtained on the dataset Pokec. In this case, we run two experiments: experiments over window of size $1,200,000$ (see Fig. 3); and an experiment where all the nodes are inserted first and then those with largest neighborhoods are deleted (see Fig. 4). Recall that for this dataset $|V| = 1,632,803$.

Results. In each of the experiments the quality of outputs of our algorithm matches those of SIEVESTREAMING. In term of the number of oracle calls, our algorithm performs at most as many as SIEVESTREAMING, while most often our method performs significantly fewer. It is interesting to note that in Fig. 4 ALG$_{0.2}$ performs significantly fewer oracle calls than ALG$_{0.0}$, while in the same time not losing on the quality of output.

In the sliding-window setting, where CNZ is applicable, we observe that the quality of the output of our approach it similar the one of CNZ. In Fig. 3 the baseline CNZ$_{0.2}$ performs almost 50% fewer oracle calls than our approach, at the expenses of outputting solutions of around 5% worse quality compared to ALG$_{0.2}$ and ALG$_{0.0}$.

Plots for fixed k. In this type of experiments we fix the value of k and split all the operations (insertions/deletions) performed in a given experiment into 400 blocks, all blocks representing the same number of operations. Then for each block we either plot the number of oracle calls or plot the average value of f. All the plots are obtained for Enron dataset.

Results. In Fig. 5, we give results Enron for $k = 20$ for window of size $30,000$, which is a more detailed experiment of the one in Fig. 1 (a) for $k = 20$. We plot the number of oracle calls performed in each block (Fig. 5 (a)), and also the cumulative number of oracle calls performed up to each block

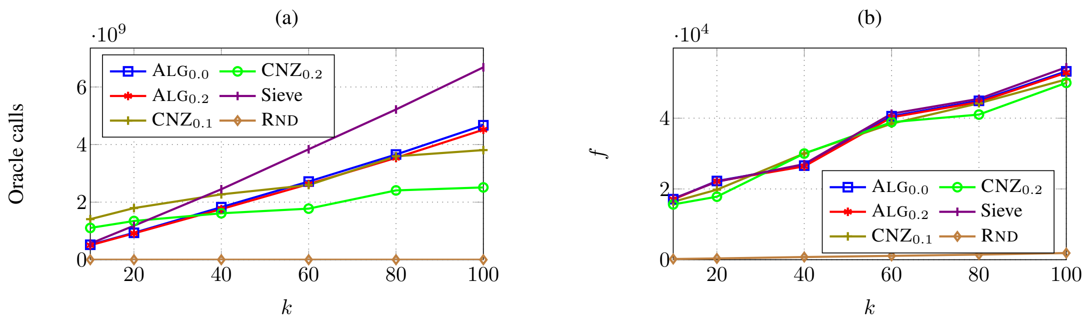

Figure 3: These plots depict results of our experiments run on Pokec network, with the elements presented as an arbitrary ordered stream. The optimization is performed over windows of size $1,200,000$.

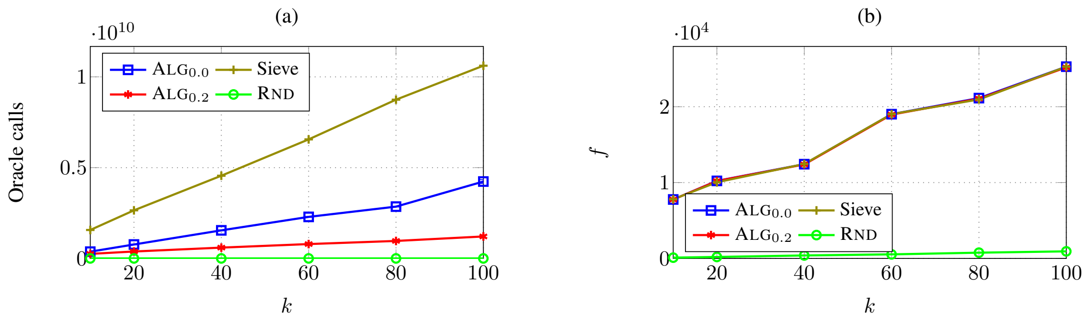

Figure 4: These plots represent the execution of our algorithm on the Pokec network. The nodes of this network are first inserted in arbitrary order, and then all the nodes are deleted by deleting first those having largest neighborhood.

(Fig. 5 (b)). The same type of experiment is performed for f as well and depicted in Fig. 9. Similar results for $k = 50$ and $k = 70$ are given in Fig. 6.

For the results on number of oracle calls, we observe that there exists a small number of operations over which our algorithm performs a significant number of oracle calls. This suggest that indeed, as we state in our main theorem, our algorithm has only amortized poly-logarithmic update time. This also leads to an interesting open question to design an algorithm that requires poly-logarithmic worst case update time. Despite this, our algorithm in total performs fewer oracle calls than $\text{CNZ}_{0.1}$, $\text{CNZ}_{0.2}$ and SIEVESTREAMING, while significantly outperforming SIEVESTREAMING.

In terms of f, the plots in Fig. 6 show the same behavior as the plot in Fig. 9.

In Fig. 7 we show results of experiments on Enron where the nodes of this network are first inserted in arbitrary order, and then all the nodes are deleted by deleting first those having largest neighborhood. It is interesting to note that for these experiments there are no operations where ALG has significant increase in the number of calls as it has for window-experiments.

D.1 Value of f after each operation.

Fig. 1 shows the average value of f for different values of k over all operations (insertions/deletions). In Fig. 9 we present a more detailed view of the experiment of Fig. 1 (b) in the following sense. We fix $k = 20$ and split the operations into 400 equal-sized blocks. For each block, we compute the average value of f. We plot those values for all the baselines. This experiment allows us to compare our approach and the baselines over the course of the entire execution. We can see that ALG is very similar to $\text{CNZ}_{0.1}$ in each block, while $\text{CNZ}_{0.2}$ shows around 10% worse performance in the blocks where f has the highest value. In those blocks, SIEVESTREAMING has around 5% better performance than ALG and $\text{CNZ}_{0.1}$.

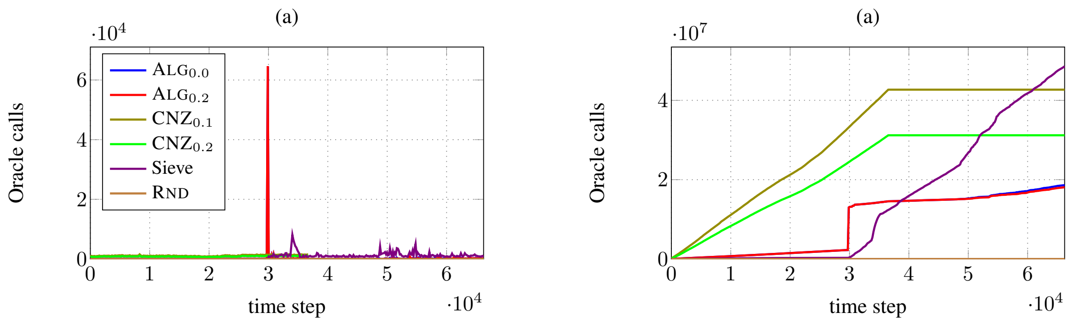

Figure 5: This plot present more detailed analysis of Fig. 1 (a) for $k = 20$ which, as a reminder, depicts a window-experiment results performed on Enron. The whole execution (i.e., the performed operations) used to obtain the point in Fig. 1 (a) for $k = 20$ is divided into 400 blocks. Plot (a) shows count of oracle calls in each block separately, while plot (b) shows cumulative number of oracle calls. Legend is the same for both plots in this figure.

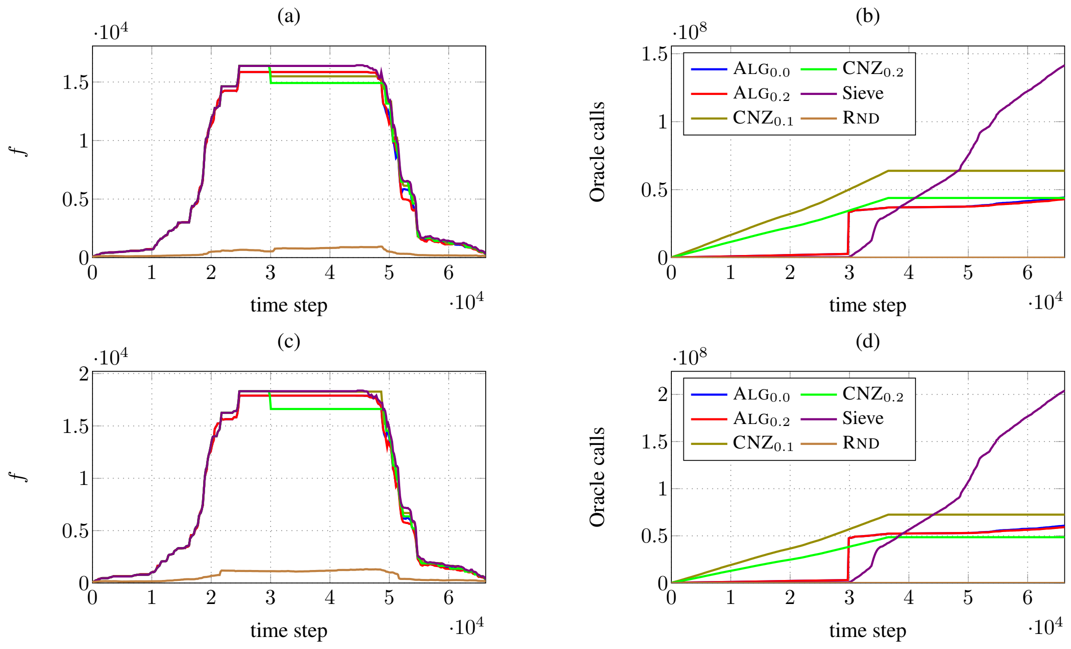

Figure 6: The setup in this figure is the same as the one for Figs. 5 and 9, and extends the result from Fig. 1 (a) and (b) for $k = 50$ (plots (a) and (b)) and $k = 70$ (plots (c) and (d)). Legend is the same for all plots in this figure.

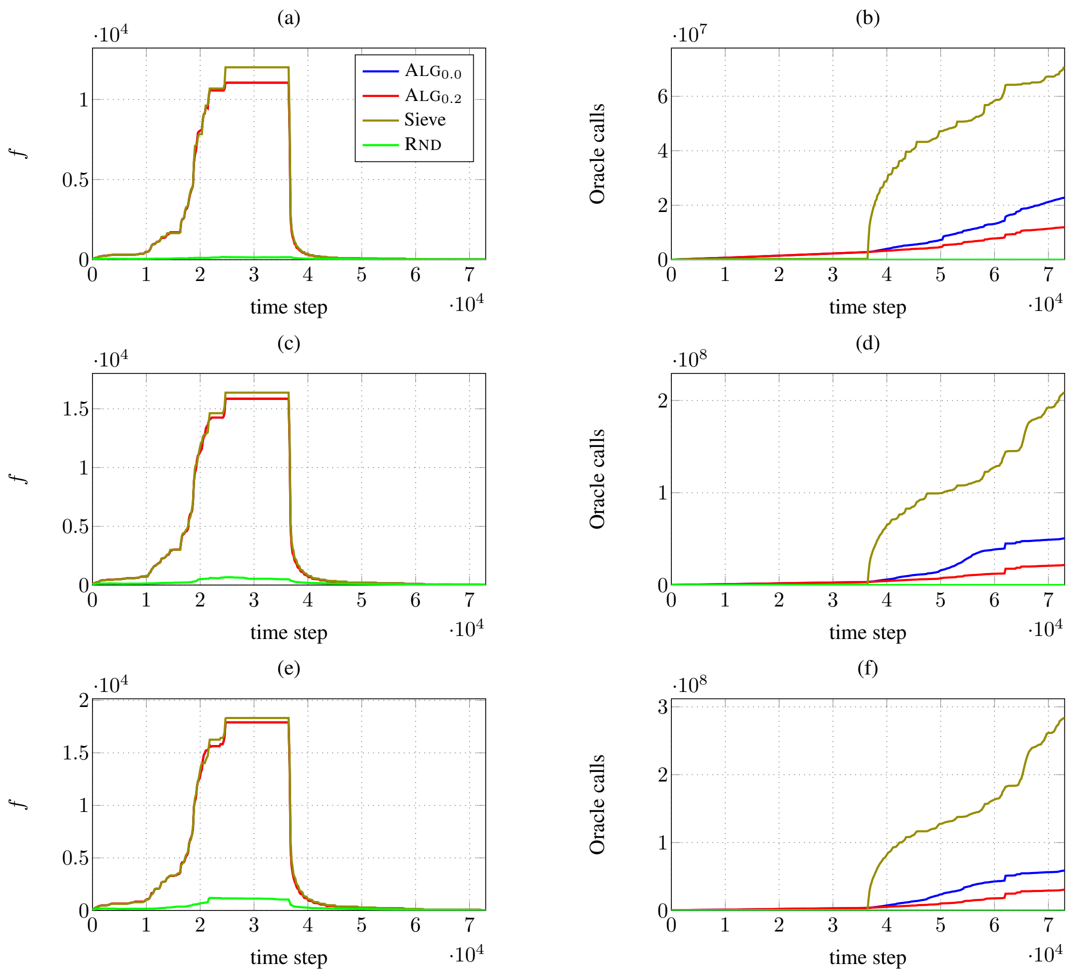

Figure 7: These plots represent the execution of our algorithm on Enron. The nodes of this network are first inserted in arbitrary order, and then all the nodes are deleted by deleting first those having largest neighborhood. Plots (a) and (b) correspond to $k = 20$, plots (c) and (d) correspond to $k = 50$, and plots (e) and (f) correspond to $k = 70$. The whole execution (i.e., the performed operations) for each experiment is divided into 400 blocks. Plots (a), (c) and (e) show average f in each block separately, while plots (b), (d) and (f) show cumulative number of oracle calls. Legend is the same for all plots in this figure.

E Details of the implemented algorithm

In this section we explain a simpler version of our algorithm, which we use for the implementation and the experiments. We do this as we believe that the bucketing idea is not crucial in real-world applications, even though it is needed to achieve the theoretical guarantee. Namely, given the random structure of layers described in our algorithm, it is very hard for an adversary to delete good elements in a layer without deleting many elements, as the notion of "good" (i.e., contribution of the element with respect to previously chosen elements) heavily depends on the random elements chosen in the previous layers. We believe this situation does not happen often in practice, and so we can simply treat all the elements in a layer in the same way. The only difference is that in this algorithm we do not partition the elements with respect to their contribution and only consider one bucket in each layer. In what follows, we present the algorithm in details for the sake of completeness. One can also read this section without reading the main algorithm.

Similar to before, in this section we assume we know OPT and first present an algorithm for the case when deletions appear only after all the insertions have been performed. Later, we explain a slight

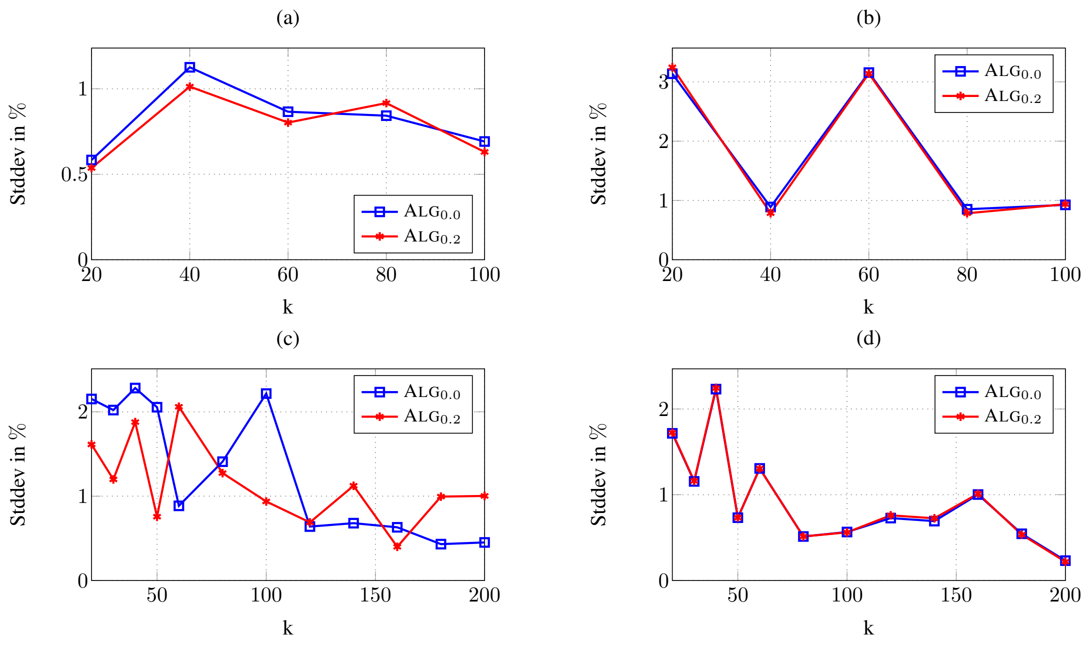

Figure 8: Plots (a), (b), (c) and (d) in this figure depict standard deviation in percentage of the values ploted in Fig. 1 (a), (b), (c) and (d), respectively.

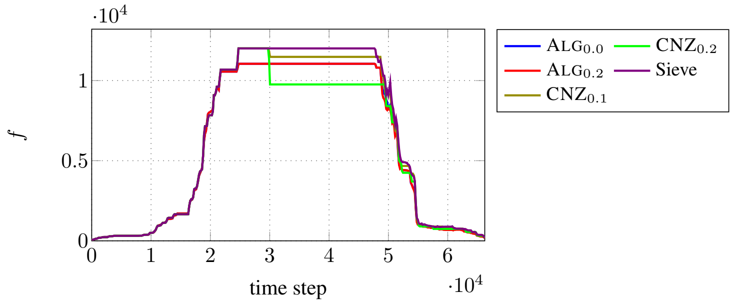

Figure 9: This plot presents a more detailed analysis of Fig. 1 (b) for $k = 20$; recall that this corresponds to a window-experiment results performed on Enron. The entire execution (i.e., the performed operations) used to obtain the point in Fig. 1 (b) for $k = 20$ is divided into 400 blocks. This plot shows the average value of f within each block.

modification to our algorithm to support fully dynamic insertions and deletions. In the following we refer to the set of inserted elements as V.

We will construct a hierarchy of sets $H_1, H_2, \ldots$. Let H_1 be all the elements $e \in V$ such that $f(e) \geq \frac{\mathrm{OPT}}{2k}$, where OPT is the value of the optimum solution of the set V. We first define

$$H_1 \leftarrow \left\{ e \in V \;\middle|\; f(e) \geq \frac{\mathrm{OPT}}{2k} \right\}.$$

Afterwards, we select an element e_1 from H_1 uniformly at random and we define H_2 as

$$H_2 \leftarrow \left\{ e \in H_1 \;\middle|\; f(e|\{e_1\}) \geq \frac{\mathrm{OPT}}{2k} \right\}.$$

We repeat this procedure until either we have chosen k elements or for some $1 \leq \ell \leq k$, the set H_ℓ has become empty. More precisely, for any $2 \leq \ell \leq k$, we define

$$H_\ell \leftarrow \left\{ e \in H_{\ell-1} \;\middle|\; f(e|\{e_1, \ldots, e_{\ell-1}\}) \geq \frac{\mathrm{OPT}}{2k} \right\}.$$

Then we let e_ℓ be a random element from H_ℓ. We call this simple procedure *a round of peeling*. It is easy to see that the running time of this simple procedure is $O(k|V|)$. This means that the average running time is $O(k)$ per element. Let m be the number of elements we have picked this way. Now we can observe that the solution $E = \{e_1, \ldots, e_m\}$ is a $1/2$-approximation:

- If $|E| = k$, then simply by adding the marginal contributions we get that:
$$f(E) = \sum_{1 \le \ell \le k} f(e_\ell | e_1 \ldots e_{\ell-1}) \ge k \frac{\mathrm{OPT}}{2k} \ge \frac{\mathrm{OPT}}{2}.$$
- If $|E| < k$, the contribution of any element $e \in V$ to a subset of E is at most $\frac{\mathrm{OPT}}{2k}$ which by submodularity of function f shows that $f(e|E) \le \frac{\mathrm{OPT}}{2k}$. By summing up this inequality for all the elements e in some optimal solution O, from submodularity we get that
$$f(O|E) \le \frac{\mathrm{OPT}}{2}.$$
Now, by monotonicity, it follows that $\mathrm{OPT} \le f(O|E) + f(E)$ and hence
$$f(E) \ge \frac{\mathrm{OPT}}{2}.$$

Now consider the deletion of an element e. If $e \notin E$, then we do nothing since the current solution has the desired guarantees; we simply remove e from all the sets in our hierarchy. Assume instead that an element $e_\ell \in E$ is deleted. Then we need to recompute the hierarchy beginning from H_ℓ. Fortunately, this does not change the expected oracle-query complexity much as, intuitively, one needs to delete roughly $O(|H_\ell|)$ elements to delete a fixed element e_ℓ with large probability (since it is a randomly chosen element of H_ℓ). Moreover, the time needed for this re-computation is $O(k \cdot |H_\ell|)$, which results in an amortized running time of $O(k)$. We introduce some additional optimizations.

Improving the insertion algorithm We introduce two modifications to the previously described algorithm.

- **Capping the number of elements in H.** We ensure that the number of elements in H_ℓ is at most $2^{\log|V|-\ell}$, which also guarantees that the height of the hierarchy is at most $\log|V| + 1$. We achieve this by running more than one round of peeling. More precisely, when constructing H_ℓ, we keep running rounds of peeling until either its size becomes below the desired threshold (i.e., $|H_\ell| \le 2^{\log|V|-\ell}$), or we have selected k elements (i.e., $|E| = k$). This does not hurt the running time nor the approximation guarantee.
- We maintain buffer sets $B_1, ..., B_T$, where $T = \log n + 1$ is the maximum possible height of the hierarchy. When an element is inserted, we add it to all the sets B_ℓ. When, for any ℓ, the sizes of B_ℓ and H_ℓ are equal we add the elements of B_ℓ to H_ℓ (and empty the set B_ℓ). The main goal of this procedure is to handle updates lazily. During the execution of the algorithm, B_ℓ essentially represents those elements that we have not considered in the construction of H_ℓ. Note that, as described before, the running time for constructing H_ℓ is at most $O(k|H_\ell|)$. This guarantees that the amortized running time for insertion is also $O(k)$. Also observe that we merge B_T whenever its size is one. This enables us to maintain a good approximate solution at all times.[12]

Improving the deletion algorithm Deletions are also handled in a lazy manner: we update our solution only when an ϵ-fraction of elements in a set is deleted. In the next section we explain this idea in more details. Intuitively, we maintain a partitioned version of H_ℓ into sets $A_{i,\ell}$ consisting of elements with similar marginal contributions. When an ϵ-fraction of elements in one set $A_{i,\ell}$ is deleted, we trigger a recomputation. Interestingly, same at the proofs presented before we can show that this significantly reduces the number of re-computations while giving a slightly weaker approximation guarantee, i.e., almost $1/2 - \epsilon$.

[12]Here we do not provide a formal proof, since in the previous sections we present a more efficient algorithm that obtains better running times for both insertions and deletions.