Fully Dynamic Submodular Maximization over Matroids

Paul Dütting [*1] **Federico Fusco** [*2] **Silvio Lattanzi** [*1] **Ashkan Norouzi-Fard** [*1] **Morteza Zadimoghaddam** [*1]

Abstract

Maximizing monotone submodular functions under a matroid constraint is a classic algorithmic problem with multiple applications in data mining and machine learning. We study this classic problem in the fully dynamic setting, where elements can be both inserted and deleted in real-time. Our main result is a randomized algorithm that maintains an efficient data structure with an $\tilde{O}(k^2)$ amortized update time (in the number of additions and deletions) and yields a 4-approximate solution, where k is the rank of the matroid.

1. Introduction

Thanks to the ubiquitous nature of "diminishing returns" functions, submodular maximization is a central problem in unsupervised learning with multiple applications in different fields, including video analysis (Zheng et al., 2014), data summarization (Lin & Bilmes, 2011; Bairi et al., 2015), sparse reconstruction (Bach, 2010; Das & Kempe, 2011), and active learning (Golovin & Krause, 2011; Amanatidis et al., 2022).

Given a submodular function f, a universe of elements V, and a family $\mathcal{F} \subseteq 2^V$ of subsets of V the submodular maximization problem consists in finding a set $S \in \mathcal{F}$ that maximizes $f(S)$. A classic choice for $\mathcal{F}$ are the capacity constraints (a.k.a. k-uniform matroid constraints) where every subset S of cardinality at most k is feasible. Another common restriction that generalizes capacity constraints and comes up in many real-world scenarios are matroid constraints. Submodular maximization under matroid constraints is NP-hard, although efficient approximation algorithms exist for this task in both the centralized and streaming setting (Fisher et al., 1978; Călinescu et al., 2011; Chakrabarti & Kale, 2015; Ene & Nguyen, 2019).

[*]Equal contribution [1]Google Research [2]Department of Computer, Control and Management Engineering "Antonio Ruberti", Sapienza University of Rome, Rome, Italy. Correspondence to: Federico Fusco <fuscof@diag.uniroma1.it>.

Proceedings of the 40th International Conference on Machine Learning, Honolulu, Hawaii, USA. PMLR 202, 2023. Copyright 2023 by the author(s).

One fundamental limitation of these algorithms is that they are not well-suited to handle highly dynamic datasets, where elements are added and deleted continuously. Many real-world applications exhibit such dynamic behaviour; for example, Dey et al. (2012) crawled two snapshots of 1.4 million New York City Facebook users several months apart and reported that 52% of the users changed their profile privacy settings during this period. Similarly, TikTok processes millions of video uploads and deletions each day, while also Snapchat processes millions of message uploads and deletions daily. In such settings, it is essential to quickly perform basic machine learning tasks, such as active learning or data summarization, so it is crucial to design *fully dynamic* algorithms that can *efficiently* process streams containing not only insertions but also an arbitrary number of deletions, with small processing time per update.

For these reasons, many problems have been studied in the dynamic setting, even if it is notoriously difficult to obtain efficient algorithms in this model. For monotone submodular maximization with a cardinality constraint, a $(2+\varepsilon)$-approximation algorithm with poly-logarithmic amortized update time (with respect to the length of the stream) was designed by Lattanzi et al. (2020); subsequently, this result has been proved to be tight by Chen & Peng (2022). In the case of submodular maximization with matroid constraints, algorithms have been proposed only for specialized dynamic settings, namely sliding windows (Chen et al., 2016; Epasto et al., 2017) and deletion robustness (Dütting et al., 2022; Mirzasoleiman et al., 2017; Zhang et al., 2022b).

Our contribution. In this paper we propose the first fully dynamic algorithm for submodular maximization under a matroid constraint with amortized running time that is sublinear in the length of the stream. Our randomized algorithm processes a stream of arbitrarily interleaved insertions and deletions with an (expected) amortized time per update that is $\tilde{O}(k^2)$*. Crucially, it also continuously maintains a solution whose value is (deterministically), after each update, at least $\frac{1}{4}$ of the optimum on the available elements.

Technical challenges. While many algorithms are known to handle insertions-only streams, it is challenging to efficiently handle deletions: removing one element from a

*In this work, $\tilde{O}$ hides factors poly-logarithmic in n (the number of elements in the stream) and k (the rank of the matroid).

candidate solution may make necessary to recompute a new solution from scratch using *all* the elements arrived in previous insertions. This is the reason why well-known techniques for the centralized or streaming framework cannot be applied directly in the dynamic setting without suffering a linear amortized update time $\Omega(n)$ (see Appendix C for further discussion). The fully-dynamic algorithm for cardinality constraint (Lattanzi et al., 2020) addresses this phenomenon via a two-dimensional bucketing data structure that allows to efficiently recover elements with large enough contribution to the current solution (and can be used to quickly recompose a good solution after a deletion). Unfortunately, that approach crucially depends on the nature of the constraint and does not extend to more structured constraints as matroids. The key difficulty is that when an element of an independent set in a matroid gets deleted, only a subset of the elements can replace it, according to the matroid constraint. This is a crucial difference with cardinality constraints, where all elements are interchangeable.

Our techniques. In this paper, we also design and analyze a data structure that is organized in levels, each one providing robustness at different scales. In addition, we carefully design an update rule that simulates in real-time the behavior of the classic SWAPPING algorithm for submodular maximization under matroid constraint (Chakrabarti & Kale, 2015). A key insight of our approach is that one can reorder and delay the addition or swapping of the elements with lower robustness without losing the simplicity and effectiveness of the SWAPPING algorithm. Interestingly, our construction simplifies substantially that of Lattanzi et al. (2020) as it removes one of the two dimensions of the dynamic data structure. Finally, we highlight a speed-up to the swapping algorithm that reduces the number of matroid independence queries by a factor $(k/\log k)$. This result may be of independent interest.

Additional related works. Independently and in parallel from Lattanzi et al. (2020), Monemizadeh (2020) achieved the same approximation guarantee with $\tilde{O}(k^2)$ amortized update time were k is the cardinality constraint. An area of research that is very close to the fully dynamic setting is robust submodular optimization (Orlin et al., 2018; Bogunovic et al., 2017; Mirzasoleiman et al., 2017; Mitrovic et al., 2017; Kazemi et al., 2018; Avdiukhin et al., 2019; Zhang et al., 2022a). In this setting, the goal is to select a summary of the whole dataset that is robust to d adversarial deletions; crucially the number d of deletions is known to the algorithm and typically all the deletions happen after the insertion in the stream. The results in this line of research do not apply to our dynamic setting where the number of deletions is arbitrary and deletions are interleaved with insertions.

2. Preliminaries

We consider a set function $f : 2^V \to \mathbb{R}_{\geq 0}$ on a (potentially large) ground set V. Given two sets $X, Y \subseteq V$, the *marginal gain* of X with respect to Y, $f(X \mid Y)$, quantifies the change in value of adding X to Y and is defined as

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

When X consists of a singleton x, we use the shorthand $f(x \mid Y)$ instead of $f(\{x\} \mid Y)$. Function f is called *monotone* if $f(e \mid X) \geq 0$ for each set $X \subseteq V$ and element $e \in V$, and *submodular* if for any two sets $X \subseteq Y \subseteq V$ 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 monotone and that it is *normalized*, i.e., $f(\emptyset) = 0$. We model access to the submodular function f via a value oracle that computes $f(S)$ for given $S \subseteq V$.

Submodularity under a matroid constraint. A non-empty family of sets $\mathcal{M} \subseteq 2^V$ is called a *matroid* if it satisfies the following properties:

- *Downward-closure* if $A \subseteq B$ and $B \in \mathcal{M}$, then $A \in \mathcal{M}$
- *Augmentation* if $A, B \in \mathcal{M}$ with $|A| < |B|$, then there exists $e \in B$ such that $A + e \in \mathcal{M}$.

For the sake of brevity, in this paper we slightly abuse the notation and for a set X and an element e, use $X + e$ to denote $X \cup \{e\}$ and $X - e$ for $X \setminus \{e\}$. We call a set $A \subseteq 2^V$ *independent*, if $A \in \mathcal{M}$, and *dependent* otherwise. An independent set that is maximal with respect to inclusion is called a *base*; all the bases of a matroid share the same cardinality k, which is referred to as the *rank* of the matroid. The problem of maximizing a function f under a *matroid constraint* $\mathcal{M}$ is defined as selecting a set $S \subseteq V$ with $S \in \mathcal{M}$ that maximizes $f(S)$. Similarly to what is done for the submodular function, we assume access to an independence oracle that takes in input $S \subseteq V$ and outputs whether S is independent with respect to the matroid or not.

Fully dynamic model. Consider a stream of exactly n insertion and n deletion operations chosen by an oblivious adversary. Denote by V_i the set of all elements inserted and not deleted up to the i-th operation. Let O_i be an optimum solution for V_i and denote $\mathrm{OPT}_i = f(O_i)$. Our goal is to design a dynamic data structure with two key properties. On the one hand, we want the data structure to maintain, at the end of each operation i, a good feasible solution $S^i \subseteq V_i$. In particular, we say that an algorithm is an α-approximation of the best (dynamic) solution if $\mathrm{OPT}_i \leq \alpha f(S^i)$, for all $i = 1, \ldots, 2n$. On the other hand,

Algorithm 1 SWAPPING

1: **Environment:** stream π, function f and matroid $\mathcal{M}$
2: $S \leftarrow \emptyset$, $S' \leftarrow \emptyset$
3: **for** each new arriving element e from π **do**
4: $\quad w(e) \leftarrow f(e \mid S')$
5: $\quad$ **if** $S + e \in \mathcal{M}$ **then**
6: $\quad\quad S \leftarrow S + e$, $S' \leftarrow S' + e$
7: $\quad$ **else**
8: $\quad\quad s_e \leftarrow \operatorname{argmin}\{w(y) \mid y \in S,\ e + S - y \in \mathcal{M}\}$
9: $\quad\quad$ **if** $2w(s_e) < w(e)$ **then**
10: $\quad\quad\quad S \leftarrow S - s_e + e$, $S' \leftarrow S' + e$
11: **Return** S

we are interested in updating our data structure efficiently. We measure efficiency in terms of the amortized running time, i.e., the average per-operation computation: we say that an algorithm has amortized running time t if its expected total running time to process any stream of $2n$ insertions and deletions is at most $2nt$.[†] Throughout this paper, we refer to running time as the total number of submodular function evaluations (value oracle) and independent set evaluations with respect to the matroid (independence oracle). This is a standard practice in submodular optimization as these two oracles typically dominates the running time of optimization algorithms.

Insertion-only streams. The fully dynamic model can be considered — to some extent — a generalization of the insertion-only streaming model. There, an arbitrary sequence of sole insertions is passed to the algorithm that is tasked with retaining a good solution (with respect to the offline optimum), while using only little "online" memory. A key ingredient in our analysis is the SWAPPING algorithm by Chakrabarti & Kale (2015) that is a simple yet powerful routine for submodular maximization with matroid constraint in the streaming setting. SWAPPING maintains a feasible solution and, for each new arriving element, it adds it to the solution if either it does not violate the matroid constraint or it is possible to swap it with some low-value element[‡]. We use a slightly modified version of the original algorithm (see pseudocode for details); namely, the weight of a new element is computed as its marginal value with respect to the set S' of all the elements that *at some point* were in the solution. We refer to Appendix A for a formal proof of the fact that our modified version of SWAPPING still retains the approximation guarantees we want:

Theorem 2.1. *For any (possibly adaptive) stream of elements in V,* SWAPPING *outputs a deterministic* 4*-approximation to the best (offline) independent set in V.*

[†]We are interested in the asymptotic behaviour of the amortized running time, therefore we can safely assume that the sequence contains exactly n deletions.

[‡]With ties in line 8 solved in any consistent way.

3. The Algorithm

In the main body of the paper we present and analyze a simplified version of our data structure whose amortized running time depends poly-logarithmically on a parameter Δ of the function f defined as:

$$\Delta = \frac{\max_{x \in V} f(x)}{\min_{T \subseteq V, x \notin T_0} f(x \mid T)},$$

where with T_0 we denote the set of all the elements with 0 marginal contribution with respect to T. In Appendix B, we show how to replace this dependence in Δ with a $O(k/\varepsilon)$ term for any chosen precision parameter ε that influences the approximation factor in an additive way. To further simplify the presentation, we also assume without loss of generality that our algorithm knows the number n of insertions and deletions in advance, and that n is a power of 2. We show in Section 6 a simple way to avoid this assumption without affecting the approximation guarantee.

We are ready to introduce our algorithm. At a high level, it carefully maintains the stream of elements in a data structure characterized by a small amortized update time and that mimics the behavior of SWAPPING, at each insertion or deletion operation. Our data structure contains $L + 1$ levels, with $L = \log n$. Each one of these levels is characterized by four sets of elements: a partial solution S_ℓ, an auxiliary set S'_ℓ that contains S_ℓ and some elements that used to belong to S_ℓ but were later swapped out from it, a set A_ℓ of candidate elements, that meet certain criteria and are considered *good* addition to the solution, and a buffer B_ℓ of still not processed elements. Moreover, the invariants that $|A_\ell|$ and $|B_\ell|$ are smaller than $n/2^\ell$ are enforced. We claim that the solution of the last level, i.e., S_L (that plays the role of S^i), is consistently a constant factor approximation of OPT_i, at the end of each operation i. We describe how the data structure is maintained; this clarifies the details of our approach.

Initialization. At the beginning of the stream, the routine INITIALIZATION is called. It takes in input n and initializes $\Theta(\log n)$ empty sets; the ones described above: S_ℓ, S'_ℓ, A_ℓ and B_ℓ for all $\ell = 0, 1, \ldots, L$.

Algorithm 2 INITIALIZATION

1: **Input:** n
2: $L \leftarrow \log n$
3: Initialize empty sets $A_\ell, S_\ell, S'_\ell, B_\ell \quad \forall\, 0 \le \ell \le L$

Handling insertions. When a new element e is inserted, it gets immediately added to all the buffers (line 1 of INSERTION). This addition induces the call of another routine, LEVEL-CONSTRUCT, on the level ℓ with smallest index such that the buffer B_ℓ exceeds a certain cardinality (namely

when $|B_\ell| \geq n/2^\ell$, see line 2 of INSERTION). Note that such a level always exists by our choice of L.

Algorithm 3 INSERTION(e)

$B_\ell \leftarrow B_\ell + e \quad \forall\, 0 \leq \ell \leq L$
if there exists an index ℓ such that $|B_\ell| \geq \frac{n}{2^\ell}$ **then**
  Let $\ell^\star$ be such ℓ with lowest value
  Call LEVEL-CONSTRUCT($\ell^\star$)

Handling deletions. When an element e is deleted from the stream, then the data structure is updated according to DELETION. Element e is removed from all the candidate elements sets A_ℓ and buffers B_ℓ (lines 1 and 2) and causes a call of LEVEL-CONSTRUCT on the smallest-index level such that $e \in S_\ell$ (line 5). While INSERTION always induces a call of LEVEL-CONSTRUCT, DELETION only causes it if the deleted element belongs to some partial solution S_ℓ.

Algorithm 4 DELETION(e)

1: $A_\ell \leftarrow A_\ell - e \quad \forall\, 0 \leq \ell \leq L$
2: $B_\ell \leftarrow B_\ell - e \quad \forall\, 0 \leq \ell \leq L$
3: **if** $e \in S_\ell$ for some ℓ **then**
4: Let ℓ be the smallest index such that $e \in S_\ell$
5: Call LEVEL-CONSTRUCT(ℓ)

LEVEL-CONSTRUCT. We now describe the main routine of our data structure: LEVEL-CONSTRUCT. A call to this routine at level ℓ triggers some operations relevant to sets at level ℓ, and it then recursively runs LEVEL-CONSTRUCT at level $\ell+1$. Therefore LEVEL-CONSTRUCT(ℓ) is essentially responsible for reprocessing the whole data structure at all levels $\ell, \ell+1, \cdots, L$. When it is called on some level ℓ, all the sets associated to that level (S_ℓ, S'_ℓ, A_ℓ and B_ℓ) are reinitialized: the candidate elements set A_ℓ is initialized with the elements in $A_{\ell-1}$ and $B_{\ell-1}$ (line 1), the buffer B_ℓ is erased (line 2), while S_ℓ and S'_ℓ are copied from the previous level (lines 3 and 4). Then, the following iterative procedure is repeated, until the cardinality of A_ℓ becomes smaller or equal to $n/2^\ell$: first, all the elements in A_ℓ that would not be added to S_ℓ by SWAPPING are filtered out (lines 9 to 11), then, if the cardinality of A_ℓ is still large enough (i.e., $|A_\ell| \geq n/2^\ell$, see line 12) an element e from it is drawn uniformly at random and is added to the solution and to S'_ℓ (lines 14 and 15); note that if $S_\ell + e \notin \mathcal{M}$, then e needs to be swapped with some element s_e in the solution (see line 8). Two important implementation details are worth mentioning here: (i) every time an element e is added to a partial solution S_ℓ, also the information about the weight $w(e)$ it has at the moment is stored in S_ℓ; (ii) the partial solutions S_ℓ are maintained sorted in increasing order of weight. Note that these two points do not entail any call of the value or independence oracles.

Algorithm 5 LEVEL-CONSTRUCT(ℓ)

1: $A_\ell \leftarrow A_{\ell-1} \cup B_{\ell-1}$
2: $B_\ell \leftarrow \emptyset$
3: $S_\ell \leftarrow S_{\ell-1}$
4: $S'_\ell \leftarrow S'_{\ell-1}$
5: **repeat**
6: **for** any element $e \in A_\ell$ **do**
7: $w(e) \leftarrow f(e \mid S'_\ell)$
8: $s_e \leftarrow \operatorname{argmin}\{w(y) \mid y \in S_\ell \wedge S_\ell - y + e \in \mathcal{M}\}$
9: $E_\ell \leftarrow \{e \in A_\ell \mid S_\ell + e \in \mathcal{M}\}$
10: $F_\ell \leftarrow \{e \in A_\ell \setminus E_\ell \mid w(e) > 2\, w(s_e)\}$
11: $A_\ell \leftarrow E_\ell \cup F_\ell$
12: **if** $|A_\ell| \geq \frac{n}{2^\ell}$ **then**
13: Pop e from A_ℓ uniformly at random
14: $S_\ell \leftarrow S_\ell + e - s_e$
15: $S'_\ell \leftarrow S'_\ell + e$
16: **until** $|A_\ell| < \frac{n}{2^\ell}$
17: **if** $\ell < L$, call LEVEL-CONSTRUCT($\ell + 1$).

4. Approximation Guarantee

Fix any operation i, we want to show that the solution S_L maintained in the data structure at the end of the computation relative to operation i is a good approximation of the best independent set O_i (of value OPT_i) in V_i. To not overload the notation, we omit the index i when it is clear from the context, so that V stands for the elements that were inserted but not deleted from the stream up to operation i (included) and D stands for the set of elements deleted from the stream up to operation i (included). Clearly the set of all the elements arrived up to operation i is exactly $V \cup D$. We want to show that $f(S_L)$ is a (deterministic) 4-approximation of the independent set in V with largest value. In the following, we actually we prove that $f(S_L)$ is a 4-approximation of something that is at least OPT.

Up to operation i the content of the data structure has changed multiple times, but for the sake of the analysis it is enough to consider a subset of all these modifications. Formally, for each level ℓ, denote with i_ℓ the last operation that triggered a call of LEVEL-CONSTRUCT (ℓ) before or at operation i. This may have happened either directly via an insertion or a deletion at level ℓ or indirectly because something was inserted or removed at some level with smaller index. Denote with X_ℓ the elements in V_{i_ℓ} that were added to S_ℓ during the computation and with Y_ℓ the elements that were filtered out due failure in the "swapping" test. Formally, $X_\ell = S_\ell \setminus S_{\ell-1}$ at the end of the computation relative to operation i_ℓ, while Y_ℓ is made up of all the elements that were in $A_\ell = B_{\ell-1} \cup A_{\ell-1}$ at the beginning of that call of LEVEL-CONSTRUCT (ℓ), but that were neither passed to the following level nor added to the solution during the repeat loop of that call of LEVEL-CONSTRUCT. Note that,

by definition of i_ℓ, nothing happens in level ℓ between operations i_ℓ and i besides possibly some basic addition induced by INSERTION (line 1) and deletions induced by DELETION (lines 1 and 2), thus sets X_ℓ, Y_ℓ and S_ℓ do not change. We start proving that all the X_ℓ and Y_ℓ are pair-wise disjoint.

Lemma 4.1. *All the $2L+2$ sets X_ℓ and Y_ℓ are pair-wise disjoint.*

Proof. By definitions, it is immediate to see that $X_\ell \cap Y_\ell = \emptyset$ for each level $0 \leq \ell \leq L$. Now, consider any two levels ℓ and r, with $\ell < r$. Since level ℓ has a smaller index, the last operation i_ℓ during whose computation LEVEL-CONSTRUCT(ℓ) was called is precedent (or equal) to i_r, that is the equivalent operation for level r. This means that any element e in $X_\ell \cup Y_\ell$ do not belong to $X_r \cup Y_r$: e was not passed to level $\ell+1$ (and thus to any larger level, r included) during the computation at operation i_ℓ and, by definition of i_ℓ it was not considered (i.e., it did not appear in any call of line 1 of LEVEL-CONSTRUCT) in any level with larger index, for all the computations between operation i_ℓ and operation i (i_r included!). □

Denote with X the (disjoint) union of all the X_ℓ and the Y_ℓ. We prove that X is a superset of V.

Lemma 4.2. *V is contained in X: for any $e \in V$ there exists (exactly) one level ℓ such that $e \in X_\ell$ or $e \in Y_\ell$.*

Proof. When a new element $e \in V$ is inserted in the stream, it is added to all buffers (line 1 of INSERTION) and triggers a call of LEVEL-CONSTRUCT at some level ℓ^* (line 4 of INSERTION). Thus it is either added to the solution or filtered out at some level. However, this is not enough to prove the Lemma, as it is possible that that call of LEVEL-CONSTRUCT is not the last time that element e is considered.

To formally prove the Lemma we introduce two (moving) auxiliary levels u_e and d_e such that the following two invariants hold from the operation in which e is added onwards (up to operation i, included):

a) e belongs to the buffer B_ℓ for all $\ell < d_e$

b) e belongs to A_ℓ for all $d_e \leq \ell < u_e$

c) e belongs to either X_{u_e} or Y_{u_e}, for some $u_e \geq d_e$.

Stated differently, we show that at the end of each operation j that follows the insertion of e (included) up to operation i, included, there exist two levels (possibly different for each operation) such that *a*), *b*) and *c*) hold. For the sake of clarity, we omit the dependence of u_e and d_e from j.

When element e is inserted, it triggers LEVEL-CONSTRUCT at some level (we call d_e such level and note that *a*) is respected) and it is either added or filtered out at some other level (we call u_e such level so note that also *b*) and *c*) are respected). By construction of LEVEL-CONSTRUCT, it holds that $u_e \geq d_e$.

Note that $e \in V$, thus e is not deleted from the stream before operation i. So we only need to show that any further call of LEVEL-CONSTRUCT happening between the insertion of e and operation i (included) does not affect the invariants. We have three cases. For any LEVEL-CONSTRUCT that is called for some $\ell > u_e$, nothing changes, as levels $\ell \leq u_e$ are not touched. If LEVEL-CONSTRUCT is called for some $\ell < d_e$, then element e belongs to the buffer $B_{\ell-1}$ (by the invariant *a*) and it is then added to either X_{u_e} or Y_{u_e} for some (new) $u_e \geq \ell$. We rename d_e the above ℓ, and it is easy to verify that all the invariants still hold. Finally, if LEVEL-CONSTRUCT is called for some $d_e \leq \ell < u_e$, then by invariant *b*, it holds that e belongs to $A_{\ell-1}$, thus it will end up filtered out or added to the solution in some new $u_e \geq \ell$. In this case we do not change d_e, and it is easy to see that the invariants still hold. So the three invariants hold during the execution of the entire stream. To conclude the proof we just note that the invariants imply that e is only contained in either one X_ℓ or one Y_ℓ for some ℓ. □

There is a natural notion of ordering on the elements of X, induced by the order in which they were considered by the algorithm, i.e. in which they were either added to the solution S_ℓ (line 3 of LEVEL-CONSTRUCT) or filtered out by the recomputation of E_ℓ and F_ℓ (line 11 of LEVEL-CONSTRUCT), with ties broken arbitrarily. Call π this ordering. To have a better intuition of π, note that it can be split into contiguous intervals, the first corresponding to the elements considered in the first level $X_0 \cup Y_0$, then in the second $X_1 \cup Y_1$, and so on. In interval ℓ, elements of $X_\ell \cup Y_\ell$ are ordered using the same order in which they have been added or filtered out in the last call of LEVEL-CONSTRUCT (ℓ).

The crucial observation is that the solution S at the end of operation i is *exactly* the output of SWAPPING on π. To see why this is the case, consider the story of each element e arriving in π. There are two cases to consider. If e is in some X_ℓ, then our algorithm has added it to the candidate solution S_ℓ during the operation i_ℓ because e was in either E_ℓ or F_ℓ. Similarly, also SWAPPING would have added e to its solution, with the exact same swap. If e is in Y_ℓ, then it means that the algorithm has filtered it out during operation i_ℓ because it failed to meet the swapping condition: thus it would have been discarded also by SWAPPING. This implies, by Theorem 2.1, that $f(S)$ is a 4-approximation to the best independent set in X, which is an upper bound on OPT (as it is the best independent set on a larger set).

Theorem 4.3. *For any operation i it holds that the solution S^i output by the algorithm at the end of the computation relative to iteration i is a deterministic 4-approximation of* OPT_i.

5. Running Time Analysis

In this section, we analyze the amortized running time of our algorithm. Recall that, throughout this paper, we refer to running time as the total number of submodular function evaluation plus the number of independent set evaluation of the matroid. The computation of our algorithm is mainly carried out by the LEVEL-CONSTRUCT function, which exhibits a recursive structure: when INSERTION or DELETION trigger LEVEL-CONSTRUCT (ℓ), this induces a chain of recursive calls of LEVEL-CONSTRUCT to higher levels. Our proof structure is as follows. First, in Lemma 5.5, we construct a deterministic upper bound on the computation induced by any new chain of LEVEL-CONSTRUCT calls; then, in Lemma 5.6 and Lemma 5.8 we measure the number of times that INSERTION, respectively DELETION, induce new chain of LEVEL-CONSTRUCT calls. Finally, we combine these two steps in Lemma 5.9.

We start by showing some of the basic properties of the A and B sets.

Observation 5.1. For any level $0 \leq \ell \leq L$, at the end of LEVEL-CONSTRUCT(ℓ), $|A_\ell| < \frac{n}{2^\ell}$ and $|B_\ell| = 0$.

Proof. It follows directly from Line 16 in LEVEL-CONSTRUCT that $A_\ell < \frac{n}{2^\ell}$, otherwise the loop does not stop. Moreover, Line 2 in LEVEL-CONSTRUCT is the only place where B_ℓ is changed and it is set to empty set. □

Observation 5.2. For any level $0 \leq \ell \leq L$, during the execution of the algorithm $|B_\ell| \leq \frac{n}{2^\ell}$.

Proof. The only place where the size of B_ℓ increases is in Line 1 of INSERTION, where it increases by at most one. When $|B_\ell| = \frac{n}{2^\ell}$, then LEVEL-CONSTRUCT$(\ell)$ is called directly from Line 4 of INSERTION or indirectly in Line 17 of LEVEL-CONSTRUCT. In both cases $|B_\ell| = 0$ due to Observation 5.1. □

Observation 5.3. For any level $0 \leq \ell \leq L$, during the execution of the algorithm $|A_\ell| \leq \frac{n}{2^{\ell-2}}$.

Proof. For any level ℓ, the cardinality of A_ℓ only varies in two cases: when an element in A_ℓ is removed from the stream (line 1 of DELETION) or during a call of LEVEL-CONSTRUCT on level ℓ. Since the former decreases the cardinality of A_ℓ, we only study the latter. When LEVEL-CONSTRUCT (ℓ) is called, A_ℓ is initialized in line 1 and then its cardinality only decreases (Lines 11 and 13). To conclude the proof is then sufficient to prove that, every time A_ℓ is initialized in a new call of LEVEL-CONSTRUCT (ℓ), its cardinality is at most $\frac{n}{2^{\ell-1}}$. The set A_ℓ is initialized with the elements in $B_{\ell-1}$ and in $A_{\ell-1}$. We know that $|B_{\ell-1}| \leq \frac{n}{2^{\ell-1}}$ at any time of the algorithm (Observation 5.2), while the cardinality of $|A_{\ell-1}|$ did not increase since the end of last call of LEVEL-CONSTRUCT $(\ell-1)$. All in all, using the bound in Observation 5.1 we get the desired bound:

$$|A_\ell| \leq \frac{n}{2^{\ell-1}} + \frac{n}{2^{\ell-1}} = \frac{4n}{2^\ell}. \qquad \square$$

Before moving to LEVEL-CONSTRUCT, we show that it is possible to compute the candidate swaps s_e in line 8 in $O(\log k)$ calls of the independence oracle of the matroid.

Lemma 5.4. *For any element $e \in A_\ell$ it is possible to find the candidate swap s_e in line 8 of* LEVEL-CONSTRUCT *in $O(\log k)$ calls of the independence oracle of the matroid.*

Proof. Consider any iteration of the repeat loop in LEVEL-CONSTRUCT, let S be the solution, A the candidate set (we omit the dependence on ℓ for simplicity) and e any element in it. If $S + e \in \mathcal{M}$ then s_e is set to the empty set and the claim holds. Otherwise, call C the set of all elements in S that can be swapped with e to obtain an independent set:

$$C = \{y \in S \mid S - y + e \in \mathcal{M}\}.$$

It is immediate to see that $s_e \in \operatorname{argmin}_{y \in C} w(y)$. We know that the solution $S = \{x_1, x_2, \ldots, x_j\}$ is maintained in decreasing order of weights (resolving ties arbitrarily) and, by the downward closure property of matroids, we can use binary search to find

$$i^* = \max\{i \mid \{x_1, \ldots, x_{i-1}\} + e \in \mathcal{M}\}.$$

We claim that x_{i^*} is a good choice of s_e, i.e., that $x_{i^*} \in \operatorname{argmin}_{y \in C} w(y)$.

First, note that x_{i^*} belongs to C. To see this, consider the set $R = \{x_1, \ldots x_{i^*-1}\} + e \in \mathcal{M}$, and recursively add element from S to it while keeping R independent. By the augmentation property, we know that this is possible until $|R| = |S|$. A single element remains in $S \setminus R$ and it has to be x_{i^*}, as we know that $\{x_1, \ldots, x_{i^*}\} + e$ is dependent, thus $S - x_{i^*} + e = R \in \mathcal{M}$. Now, we show that no element in C can have smaller weight (i.e. larger index) than x_{i^*}. Assume toward contradiction that this is the case, i.e. that there is an x_j such that $S - x_j + e \in \mathcal{M}$ and $j < i^*$. This implies that $\{x_1, \ldots x_{j-1}\} + e$ is independent, which contradicts the minimality of i^*. □

Lemma 5.5. *For any level $0 \leq \ell \leq L$, the running time of* LEVEL-CONSTRUCT(ℓ) *is $O(\frac{nk \log \Delta \log k}{2^\ell})$.*

Proof. We prove this Lemma in two steps. First, we control the running time of the non-recursive part of LEVEL-CONSTRUCT (ℓ) (i.e., all the algorithm but the recursive call

in line 17), then we study, by induction, how these bounds combine recursively.

We start proving that, for every level ℓ from 0 to L, the running time of the non-recursive part of LEVEL-CONSTRUCT (ℓ) is dominated by $c\frac{nk\log k\log\Delta}{2^\ell}$, for some positive constant c. Focus on any level ℓ and consider any call of LEVEL-CONSTRUCT (ℓ). The main computation is performed in the repeat loop (Lines 5-16), which is repeated at most $k\log\Delta$ times (due to the exponential increase of the weight assigned to the elements, the fact we can at most add k elements without swapping and the definition of Δ). In each iteration of the repeat loop, the running time of finding the swap elements in Lines 6-11 is at most order of $|A_\ell|\log k$, which is in $O(\frac{n\log k}{2^\ell})$ (recall, the cardinality of A_ℓ is bounded in Observation 5.3). This concludes the analysis of the non recursive part of LEVEL-CONSTRUCT: there exists a constant $c>0$ such that its running time is at most $c\frac{nk\log k\log\Delta}{2^\ell}$, for every level ℓ from 0 to L.

We now conclude the proof of the Lemma by induction on ℓ from L to 0. More precisely, we show that the (overall) running time of LEVEL-CONSTRUCT(ℓ) is bounded by $2c\frac{nk\log k\log\Delta}{2^\ell}$ for any level $0\le\ell\le L$. We start considering the base case $\ell=L$. As there is no recursive call, the inequality immediately descends on the bound on the non-recursive running time of LEVEL-CONSTRUCT (L). For the induction step, assume that the running time of LEVEL-CONSTRUCT$(\ell+1)$ is upper bounded by $2c\frac{nk\log k\log\Delta}{2^{\ell+1}}$ and we show that the same holds for level ℓ. This is pretty easy to see: the interval running time of LEVEL-CONSTRUCT (ℓ) is at most $c\frac{nk\log k\log\Delta}{2^\ell}$, while the recursive call to LEVEL-CONSTRUCT $(\ell+1)$ has running time $2c\frac{nk\log k\log\Delta}{2^{\ell+1}}$ by the inductive hypothesis. Summing up these two terms yields the desired result. □

We have just assessed a deterministic upper bound on the computational complexity of each call of LEVEL-CONSTRUCT. We now bound the number of times that LEVEL-CONSTRUCT is called during the stream of insertions and deletions. To do so, we bound separately the number of times LEVEL-CONSTRUCT is directly induced by INSERTION or DELETION. By "directly", we mean that LEVEL-CONSTRUCT (ℓ) is called either in line 4 of INSERTION or in line 5 of DELETION, and not as part of a recursive chain induced by some call of LEVEL-CONSTRUCT (ℓ'), for some $\ell'<\ell$. We start by counting the number of calls induced directly by INSERTION.

Lemma 5.6. *For any level $0\le\ell\le L$, the number of times that the* LEVEL-CONSTRUCT(ℓ) *function is called directly from* INSERTION *is at most 2^ℓ.*

Proof. The only place where the size of set B_ℓ increases is in Line 1 of INSERTION, and it increases by at most one per insertion. Moreover, B_ℓ is set to the empty set in Line 2 of LEVEL-CONSTRUCT(ℓ). Also there are at most n insertions and LEVEL-CONSTRUCT(ℓ) is called when the size of B_ℓ is equal to $\frac{n}{2^\ell}$. Therefore there are at most $n\frac{2^\ell}{n}=2^\ell$ calls to LEVEL-CONSTRUCT(ℓ) from INSERTION. □

We move our attention to the number of calls to LEVEL-CONSTRUCT directly induced by deletions. We call a period between two direct invocations of LEVEL-CONSTRUCT(ℓ) from DELETION an ℓ-epoch, and denote with N_ℓ the (random) number of such epochs. We call an ℓ-epoch short if its length is less than $\frac{n}{k2^{\ell+1}\log\Delta}$ and long otherwise. The next Lemma helps us bound the expected number of short ℓ-epochs.

Lemma 5.7. *For any level $0\le\ell\le L$, and any deletion that induces a direct call to* LEVEL-CONSTRUCT (ℓ)*, the probability that the new ℓ-epoch is short is at most $1/2$.*

Proof. Fix any sequence of insertions and deletions, any level $0\le\ell\le L$ and any history of the algorithm up to the point where deletion operation i_0 directly induces a call of LEVEL-CONSTRUCT (ℓ). Consider now the next $T=\frac{n}{k2^{\ell+1}\log\Delta}$ operations[§]: from $i_1=i_0+1$ to $i_T=i_0+T$ and fix any history of the algorithm on these operations, but only relative to computations happening in levels $\ell'<\ell$. This is well defined because what happens in upper levels $\ell''\ge\ell$ has no influence on what happens in lower levels $\ell'<\ell$. We now focus on the randomness of the algorithm at level ℓ during the operations from i_0 to i_T and prove that the probability that the ℓ-epoch starting in i_0 is short is at most $1/2$. By the law of total probability, the arbitrary choice of the history, and the arbitrary choice of the level ℓ and deletion operation i_0, this is sufficient to prove the Lemma.

The history of the algorithm up to operation i_0, as well as the history at levels $\ell'<\ell$ is fixed. Some of the operations from i_1 to i_T induce calls of LEVEL-CONSTRUCT (ℓ) due to (i) direct insertion at level ℓ or (ii) recursive call of LEVEL-CONSTRUCT starting in some lower level $\ell'<\ell$. The calls in (ii) happen *regardless* of the random choices of the algorithm at level ℓ, while (i) depends on them (because the cardinality of B_ℓ depends on the last LEVEL-CONSTRUCT (ℓ) call which, in turn, depends on whether the ℓ-epoch is still on-going). However, if the ℓ-epoch does not terminate before i_T, then it is possible to deterministically compute these insertion operations. Call the fixed operations of type (i) and (ii) *marked*, and number them i_{t_1}, $i_{t_2},\dots,i_{t_j}$ for some $j\le T$. Focus now on the remaining deletions $i_{d_1},i_{d_2},\dots$ between i_0 and i_T and group them according to the last marked operation before them: D_1 contains all the unmarked deletions between i_1 and $i_{t_1}-1$,

[§]For the sake of simplicity we assume T to be integer; if this is not the case then it is sufficient to take the integer part and modify by a constant factor the multiplicative constants in the rest of the Lemma and in Lemma 5.8.

D_2 all the unmarked deletions between $i_{t_1}+1$ and $i_{t_2}-1$, and so on, up to D_j, containing all the unmarked deletions between $i_{t_j}+1$ and i_T. For each unmarked deletion operation i_{d_h}, let $i_t(h)$ be the corresponding marked operation (so that $i_{d_h} \in D_{t(h)}$).

With this notation there is a clear characterization of deletion operations that may induce a direct call to LEVEL-CONSTRUCT (ℓ): a deletion operation i_{d_h} concludes the ongoing ℓ-epoch started at i_0 if and only if the element deleted has been sampled in the solution S_ℓ during the LEVEL-CONSTRUCT (ℓ) call at operation $i_{t(h)}$. Let $\mathcal{E}_h$ be the event that the element deleted at operation i_{d_h} is in the solution S_ℓ sampled in LEVEL-CONSTRUCT (ℓ) at operation $i_{t(h)}$. We have the following:

$$\mathbb{P}\left(\mathcal{E}_h\right) \leq 2^\ell \frac{k \log \Delta}{n}. \tag{1}$$

The reason is simple: in the LEVEL-CONSTRUCT (ℓ) call at operation $i_{t(h)}$, at most $k \log \Delta$ elements have been sampled and added to S_ℓ (and possibly swapped out of it). Each of these elements is sampled from a candidate pool of at least $\frac{n}{2^\ell}$ elements. Therefore the probability that the element deleted in operation i_{d_h} is from the sampled elements is at most $\frac{k2^\ell \log \Delta}{n}$. Recall, the probability in Equation (1) is conditioned with respect to overall history of the algorithm up to operation i_0, and the history on lower levels up to i_T.

Let $\mathcal{E}$ be the event that the ℓ-epoch starting at i_0 is short, it holds that $\mathcal{E}$ is given by the union of all the $\mathcal{E}_h$ (that are at most T). By union bound on these (at most) T events, by the bound in Equation (1), and the definition of T we have the following:

$$\mathbb{P}\left(\mathcal{E}\right) \leq \sum_j \mathbb{P}\left(\mathcal{E}_j\right) \leq T 2^\ell \frac{k \log \Delta}{n} = \frac{1}{2}. \qquad \square$$

Lemma 5.8. *For any level $0 \leq \ell \leq L$, the expected number of times that the* LEVEL-CONSTRUCT (ℓ) *function is called directly from* DELETION *is at most $2^{\ell+3} k \log \Delta$.*

Proof. Denote with N_ℓ^- and N_ℓ^+ the number of short, respectively long, ℓ-epochs. By Lemma 5.7, we know that the probability of a new epoch being short is at most $1/2$, therefore we have the following:

$$\mathbb{E}\left[N_\ell^-\right] \leq \frac{1}{2}\mathbb{E}\left[N_\ell\right] = \frac{1}{2}\mathbb{E}\left[N_\ell^-\right] + \frac{1}{2}\mathbb{E}\left[N_\ell^+\right]. \tag{2}$$

Equation (2) implies that $\mathbb{E}\left[N_\ell^-\right] \leq \mathbb{E}\left[N_\ell^+\right]$.

We know that the number N_ℓ^+ of long epochs is at most the total number of operations $2n$, divided by the lower bound on the cardinality of each long epoch, $\frac{n}{k2^{\ell+1}\log\Delta}$. All in all, $N_\ell^+ \leq 2^{\ell+2} k \log \Delta$. We are ready to conclude:

$$\begin{aligned}\mathbb{E}\left[N_\ell\right] &= \mathbb{E}\left[N_\ell^-\right] + \mathbb{E}\left[N_\ell^+\right] \leq 2\mathbb{E}\left[N_\ell^+\right] \\ &\leq 2^{\ell+3} k \log \Delta. \qquad \square\end{aligned}$$

Lemma 5.9. *The expected average running time per operation is $O(k^2 \log k \log^2 \Delta \log n)$.*

Proof. The running time when an element is inserted or deleted is $O(L) = O(\log n)$ beside the calls made to LEVEL-CONSTRUCT . In what follows we focus on the total running time spent in LEVEL-CONSTRUCT . There are two places that LEVEL-CONSTRUCT(ℓ) can be called from (beside the recursion in Line 17 of LEVEL-CONSTRUCT): INSERTION and DELETION. By comparing Lemma 5.6 and Lemma 5.8 it results that the number of calls induced by DELETION dominates those induced by INSERTION. So we only focus on the former term. Let c be the constant as in the analysis of Lemma 5.5, we bound the total expected running time by

$$\begin{aligned}&2 \sum_{0 \leq \ell \leq L} k 2^{\ell+3} \log \Delta \cdot \left(2c \cdot \frac{nk \log k \log \Delta}{2^\ell}\right) \\ &\quad = 32c \sum_{0 \leq \ell \leq L} nk^2 \log k \log^2 \Delta \\ &\quad = 32c \cdot \left(nk^2 \log k \log^2 \Delta \log n\right). \qquad \square\end{aligned}$$

6. Putting it Together

The results in the previous sections hold under the assumption that the algorithm designer knows the number of insertion and deletions (denoted by n) in advance. In this section we present a well-known tool that enables our algorithm to run without this assumption. We simply start by $n = 1$, and whenever the number of insertions and deletions reaches to n, we restart algorithm and double the value of n. Therefore, if the total operations is m, then largest value n that we use is the smallest power of 2 bigger than m, which is at most $2m$. Combining with Lemma 5.9 we get that the total running time per operation is (up to multiplicative constants)

$$\begin{aligned}&\sum_{1 \leq n_0 \leq \log n} k^2 \log k \log^2 \Delta n_0 \\ &\quad = k^2 \log k \log^2 \Delta \log n^2 \\ &\quad = k^2 \log k \log^2 \Delta \log^2 m. \end{aligned} \tag{3}$$

Combining Equation (3) and Theorem 4.3, we have the main result of the paper.

Theorem 6.1. *Our algorithm yields a 4-approximation to the fully dynamic monotone submodular maximization problem with matroid constraint and exhibit an $O(k^2 \log k \log^2 \Delta \log^2 n)$ expected amortized running time.*

In Appendix B we also explain how one can avoid the dependency on Δ which requires: (i) designing and analyze a new algorithm that combines swapping and thresholding (presented in Appendix A) and (ii) apply standard techniques

to guess the value of OPT and run multiple copies of the algorithm at the same time.

Corollary 6.2. *For any constant $\varepsilon > 0$, there exists a $(4 + O(\varepsilon))$-approximation to the fully dynamic monotone submodular maximization problem with matroid constraint that exhibits an $O\left(\frac{k^2}{\varepsilon} \log k \log^2 n \log^3 \frac{k}{\varepsilon}\right)$ expected amortized running time.*

7. Conclusions and Future Directions

In this paper we design the first efficient algorithm for fully-dynamic submodular maximization with matroid constraint. An interesting open question stems immediately from our result: is it possible to reduce the amortized running to depend only poly-logarithmically in k (currently it is $\tilde{O}(k^2)$)?

In this paper we focus on the crucial worst-case paradigm, constructing an algorithm whose guarantees are robust to any (oblivious) adversary that generates the stream of insertions and deletions. An interesting open problem of research is to study beyond-worst case analysis, when it is natural to assume some "non-adversarial" structure on the stream, similarly to what has been done, for instance, in the random-order arrival model for insertion-only streams (Norouzi-Fard et al., 2018; Liu et al., 2021; Feldman et al., 2022).

Acknowledgements

The authors would like to thank the anonymous reviewers and Jakub Tarnawski for valuable feedback. The work of Federico Fusco is partially supported by ERC Advanced Grant 788893 AMDROMA "Algorithmic and Mechanism Design Research in Online Markets", PNRR MUR project PE0000013-FAIR", and PNRR MUR project IR0000013-SoBigData.it. Part of this work was done while Federico was an intern at Google Research, hosted by Paul Dütting.

References

Amanatidis, G., Fusco, F., Lazos, P., Leonardi, S., and Reiffenhäuser, R. Fast adaptive non-monotone submodular maximization subject to a knapsack constraint. *J. Artif. Intell. Res.*, 74:661–690, 2022.

Avdiukhin, D., Mitrovic, S., Yaroslavtsev, G., and Zhou, S. Adversarially robust submodular maximization under knapsack constraints. In *KDD*, pp. 148–156, 2019.

Bach, F. R. Structured sparsity-inducing norms through submodular functions. In *NIPS*, pp. 118–126, 2010.

Bairi, R., Iyer, R. K., Ramakrishnan, G., and Bilmes, J. A. Summarization of multi-document topic hierarchies using submodular mixtures. In *ACL*, pp. 553–563, 2015.

Bogunovic, I., Mitrovic, S., Scarlett, J., and Cevher, V. Robust submodular maximization: A non-uniform partitioning approach. In *ICML*, pp. 508–516, 2017.

Călinescu, G., Chekuri, C., Pál, M., and Vondrák, J. Maximizing a monotone submodular function subject to a matroid constraint. *SIAM J. Comput.*, 40(6):1740–1766, 2011.

Chakrabarti, A. and Kale, S. Submodular maximization meets streaming: matchings, matroids, and more. *Math. Program.*, 154(1-2):225–247, 2015.

Chekuri, C., Gupta, S., and Quanrud, K. Streaming algorithms for submodular function maximization. In *ICALP (1)*, volume 9134 of *Lecture Notes in Computer Science*, pp. 318–330. Springer, 2015.

Chen, J., Nguyen, H. L., and Zhang, Q. Submodular maximization over sliding windows. *CoRR*, abs/1611.00129, 2016.

Chen, X. and Peng, B. On the complexity of dynamic submodular maximization. In *STOC*, pp. 1685–1698, 2022.

Das, A. and Kempe, D. Submodular meets spectral: Greedy algorithms for subset selection, sparse approximation and dictionary selection. In *ICML*, pp. 1057–1064, 2011.

Dey, R., Jelveh, Z., and Ross, K. Facebook users have become much more private: A large-scale study. In *PerCom*, pp. 346–352, 2012.

Dütting, P., Fusco, F., Lattanzi, S., Norouzi-Fard, A., and Zadimoghaddam, M. Deletion robust submodular maximization over matroids. In *ICML*, pp. 5671–5693, 2022.

Dütting, P., Fusco, F., Lattanzi, S., Norouzi-Fard, A., and Zadimoghaddam, M. Deletion robust non-monotone submodular maximization over matroids. *CoRR*, abs/2208.07582, 2022.

Ene, A. and Nguyen, H. L. Towards nearly-linear time algorithms for submodular maximization with a matroid constraint. In *ICALP*, pp. 54:1–54:14, 2019.

Epasto, A., Lattanzi, S., Vassilvitskii, S., and Zadimoghaddam, M. Submodular optimization over sliding windows. In *WWW*, pp. 421–430, 2017.

Feldman, M., Karbasi, A., and Kazemi, E. Do less, get more: Streaming submodular maximization with subsampling. In *NeurIPS*, pp. 730–740, 2018.

Feldman, M., Liu, P., Norouzi-Fard, A., Svensson, O., and Zenklusen, R. Streaming submodular maximization under matroid constraints. In *ICALP*, pp. 59:1–59:20, 2022.

Fisher, M. L., Nemhauser, G. L., and Wolsey, L. A. An analysis of approximations for maximizing submodular set functions—ii. In *Polyhedral combinatorics*, pp. 73–87. Springer, 1978.

Golovin, D. and Krause, A. Adaptive submodularity: Theory and applications in active learning and stochastic optimization. *J. Artif. Intell. Res.*, 42:427–486, 2011.

Kazemi, E., Zadimoghaddam, M., and Karbasi, A. Scalable deletion-robust submodular maximization: Data summarization with privacy and fairness constraints. In *ICML*, pp. 2549–2558, 2018.

Lattanzi, S., Mitrovic, S., Norouzi-Fard, A., Tarnawski, J., and Zadimoghaddam, M. Fully dynamic algorithm for constrained submodular optimization. In *NeurIPS*, 2020.

Lin, H. and Bilmes, J. A class of submodular functions for document summarization. In *HLT*, pp. 510–520, 2011.

Liu, P., Rubinstein, A., Vondrák, J., and Zhao, J. Cardinality constrained submodular maximization for random streams. In *NeurIPS*, pp. 6491–6502, 2021.

Mirzasoleiman, B., Karbasi, A., and Krause, A. Deletion-robust submodular maximization: Data summarization with "the right to be forgotten". In *ICML*, pp. 2449–2458, 2017.

Mitrovic, S., Bogunovic, I., Norouzi-Fard, A., Tarnawski, J., and Cevher, V. Streaming robust submodular maximization: A partitioned thresholding approach. In *NIPS*, pp. 4557–4566, 2017.

Monemizadeh, M. Dynamic submodular maximization. In *NeurIPS*, 2020.

Norouzi-Fard, A., Tarnawski, J., Mitrovic, S., Zandieh, A., Mousavifar, A., and Svensson, O. Beyond 1/2-approximation for submodular maximization on massive data streams. In *ICML*, pp. 3826–3835, 2018.

Orlin, J. B., Schulz, A. S., and Udwani, R. Robust monotone submodular function maximization. *Math. Program.*, 172 (1):505–537, 2018.

Zhang, G., Tatti, N., and Gionis, A. Optimal deletion-robust coreset for submodular maximization. *CoRR*, abs/2203.01241, 2022a.

Zhang, G., Tatti, N., and Gionis, A. Coresets remembered and items forgotten: submodular maximization with deletions. In *ICDM*, pp. 676–685. IEEE, 2022b.

Zheng, J., Jiang, Z., Chellappa, R., and Phillips, P. J. Submodular attribute selection for action recognition in video. In *NIPS*, pp. 1341–1349, 2014.

A. SWAPPING algorithm

In this section we formally prove that the version of SWAPPING presented in Section 2 maintains the approximation guarantees of the original algorithm by Chakrabarti & Kale (2015). Actually, we prove a more general result that is instrumental to prove, in Appendix B, that it is possible to remove the dependence in Δ from the amortized running time.

Consider the following version of SWAPPING, that we call THRESHOLD-SWAPPING, where all the elements with weight below a given threshold are simply ignored by the algorithm. The details are given in the pseudocode. Note that when the threshold is set to 0, THRESHOLD-SWAPPING and SWAPPING coincides. We prove now the following Theorem, that clearly implies Theorem 2.1 by setting ε and τ to 0.

Algorithm 6 THRESHOLD-SWAPPING

Input: rank k of the matroid, precision parameter ε and threshold τ
Environment: stream π of elements, function f, matroid $\mathcal{M}$
$S \leftarrow \emptyset$, $S' \leftarrow \emptyset$
for each new arriving element e from π **do**
  $w(e) \leftarrow f(e \mid S')$
  if $w(e) < \frac{\varepsilon}{k}\tau$ **then**
    Ignore e and **continue**
  if $S + e \in \mathcal{M}$ **then**
    $S \leftarrow S + e$, $S' \leftarrow S' + e$
  else
    $s_e \leftarrow \operatorname{argmin}\{w(y) \mid y \in S,\ x + S - y \in \mathcal{M}\}$
    if $2w(s_e) < w(e)$ **then**
      $S \leftarrow S - s_e + e$, $S' \leftarrow e + S'$
Return S

Theorem A.1. *For any $\varepsilon > 0$ and threshold $\tau < \mathrm{OPT}$, it holds that* THRESHOLD-SWAPPING *yields a $(4 + O(\varepsilon))$-approximation to the optimal offline solution on the stream.*

The proof of Theorem A.1 follows a similar analysis to the one of the original algorithm in Chakrabarti & Kale (2015); we precede it here with some Lemmata.

Lemma A.2. *Let $K = S' \setminus S$ be the set of elements that were in the solution and were later swapped out and extend by linearity the function w to sets. Then the following three properties hold true:*

(i) $w(K) \leq w(S)$

(ii) $w(S) \leq f(S)$

(iii) $f(S') = w(S')$

Proof. Crucially, the weight function w is linear and once an element enters S, its weight is fixed forever as the marginal value it contributed when entering S but with respect to S'. During the run of the algorithm, every time an element s_e is removed from S, the weight of S increases by $w(e) - w(s_e)$ by its replacement with some element e. Moreover, $w(s_e) \leq w(e) - w(s_e)$ for every element $s_e \in K$ since $2 \cdot w(s_e) \leq w(e)$. Summing up over all elements in K, it holds that

$$w(K) = \sum_{s_e \in K} w(s_e) \leq \sum_{s_e \in K} [w(e) - w(s_e)] \leq w(S),$$

where the last inequality follows from a telescoping argument (for each chain of swaps we only retain with the positive sign the final elements that remained in S). This establishes (i).

Consider now the second inequality (ii). We denote with S'_e the version of set S' when element e was added, and we denote with $S_e = S'_e \cap S$. We have the following:

$$f(S) = \sum_{e \in S} f(e|S_e) \geq \sum_{e \in S} f(e|S'_e) = \sum_{e \in S} w(e) = w(S),$$

where the crucial inequality is due to submodularity. Finally, the last point (iii) follows by the definition of w. □

In the analysis of the approximation guarantees of THRESHOLD-SWAPPING we need to somehow account for all the elements in the optimum that were initially added to the solution by the algorithm but then later swapped out. To analyze this "chain of swaps" we need a useful combinatorial lemma: Lemma 13 of Feldman et al. (2018) (also Lemma 30 of Chekuri et al. (2015)). It concerns directed acyclic graphs (DAGs) where the nodes are also elements of a matroid. Under some assumption, it guarantees the existence of an injective mapping between elements in an independent set and of the sinks of the DAG. As a convention, we denote with $\delta^+(u)$ the out-neighborhood of any node u and we say that an independent set T spans an element x if $T + x \notin \mathcal{M}$.

Lemma A.3. *Consider an arbitrary directed acyclic graph $G = (V, E)$ whose vertices are elements of some matroid $\mathcal{M}$. If every non-sink vertex u of G is spanned by $\delta^+(u)$ in $\mathcal{M}$, then for every set S of vertices of G which is independent in $\mathcal{M}$ there must exist an injective function ψ such that, for every vertex $u \in S$, $\psi(u)$ is a sink of G which is reachable from u.*

We clarify now how we intend to use the previous result to our problem, similarly to what is done in Dütting et al. (2022).

Lemma A.4. *Let OPT be the optimal offline solution and denote with F the set of elements that failed the threshold test in line 6 of* THRESHOLD-SWAPPING. *Then it holds that $w(\mathrm{OPT} \setminus (S' \cup F)) \leq 2w(S)$.*

Proof. Consider the following procedure to construct a DAG whose nodes are given by the elements of the stream that passed the weight test in line 6. When an element x arrives, call C the circuit[¶] in $S + x$ and y the element in $C - x$ with smaller weight. If y is swapped with x, then we add directed edges from y to all the elements in $C - y$, if x is not added, then add directed edges from x to each element in $C - x$. If x is added without any swap, then its out-neighborhood is empty. Every edge in this graph points from a vertex dropped or swapped out at some time to a vertex that is either never deleted or removed at some time in the future. This time component makes the underlying graph a DAG. We now apply Lemma A.3 on G, then there exists an injective function ψ that associates each element in $\mathrm{OPT} \setminus (S' \cup F)$ to an element in S such that $w(u) \leq 2 \cdot w(\psi(u))$ for all $u \in \mathrm{OPT} \setminus (S' \cup F)$. To see this, consider that in each u-$\psi(u)$ path there is at most one swapping where the weight does not increase, and it has to be the first swap if the new element u of the stream was not added to the solution because of some node in the solution whose weight was possibly smaller, but no more than a factor 2 smaller. This implies that $w(\mathrm{OPT} \setminus (S' \cup F)) \leq 2 \cdot w(S)$. □

We finally have all the ingredients to prove the Theorem.

Proof of Theorem A.1. The proof of the Theorem is just a simple chain of inequalities that uses the Lemmata:

$$
\begin{aligned}
f(\mathrm{OPT}) &\leq f(\mathrm{OPT} \cup S') \\
&= f(S') + f(\mathrm{OPT} \,|S') \\
&\leq w(S) + w(K) + \sum_{e \in \mathrm{OPT} \setminus S'} f(e|S') && \text{(Property } (iii) \text{ and submodularity)} \\
&\leq 2 \cdot f(S) + \sum_{e \in \mathrm{OPT} \setminus (S' \cup F)} w(e) + \sum_{e \in \mathrm{OPT} \cap F} w(e) && \text{(Properties } (ii) + (i) \text{ and submodularity)} \\
&\leq 4 \cdot f(S) + \sum_{e \in \mathrm{OPT} \cap F} w(e) && \text{(Lemma A.4 and Property } (ii)\text{)} \\
&\leq 4 \cdot f(S) + \varepsilon \, \mathrm{OPT} \,. && \text{(Assumption on } \tau\text{)}
\end{aligned}
$$

□

B. Removing the dependence on Δ

In this section, we show how to remove the dependence in Δ of the amortized running time. Similarly to Lattanzi et al. (2020), for any fixed choice of a precision parameter $\varepsilon > 0$, we run multiple instances of our algorithm in parallel, where each instance is parametrized with a threshold $\tau = (1 + \varepsilon)^j$ for various integer values of j. In each one of these instances,

[¶] A circuit is a dependent set that is minimal with respect to inclusion

Algorithm 7 INSERTION(e) corresponding to threshold $\tau = (1+\varepsilon)^j$, for some integer j

if $(1+\varepsilon)\tau > f(e) \geq \frac{\varepsilon}{k}\tau$ **then**
  $B_\ell \leftarrow B_\ell + e \quad \forall\, 0 \leq \ell \leq L$
  if there exists an index ℓ such that $|B_\ell| \geq \frac{n}{2^\ell}$ **then**
    Let $\ell^\star$ be such ℓ with lowest value
    Call LEVEL-CONSTRUCT($\ell^\star$)

all the elements with weight smaller than $\varepsilon \cdot \tau/k$ are simply ignored. We return the best solution among these parallel runs after each operation.

There are two modifications to the algorithm presented in the main body that we implement. One in the INSERTION routine, and one in LEVEL-CONSTRUCT. We present here the pseudocodes and describe the changes.

The INSERTION routine is modified in a simple way: when an element is inserted, it is actually considered for insertion only if its value is within a certain range. We refer to the pseudocode for the details. Given this modification and the geometric construction of the thresholds $\tau = (1+\varepsilon)^j$, we have an immediate bound on the number of parallel rounds of the algorithm that any element is actually inserted into:

Observation B.1. Any element e is inserted in $O(\frac{1}{\varepsilon}\log\frac{k}{\varepsilon})$ copies of the algorithm.

We move our attention towards LEVEL-CONSTRUCT. There, we modify the repeat loop to filter out all the elements whose marginal contribution falls, at any point, below $\frac{\varepsilon}{k}\tau$. More precisely, we add Line 9 in the following pseudocode.

Algorithm 8 LEVEL-CONSTRUCT(ℓ) corresponding to threshold $\tau = (1+\varepsilon)^j$, for some integer j

1: $A_\ell \leftarrow A_{\ell-1} \cup B_{\ell-1}$
2: $B_\ell \leftarrow \emptyset$
3: $S_\ell \leftarrow S_{\ell-1}$
4: $S'_\ell \leftarrow S'_{\ell-1}$
5: **repeat**
6: **for** any element $e \in A_\ell$ **do**
7: $w(e) \leftarrow f(e \mid S'_\ell)$
8: $s_e \leftarrow \operatorname{argmin}\{w(y) \mid y \in S_\ell \wedge S_\ell - y + e \in \mathcal{M}\}$
9: $A_\ell \leftarrow \{e \in A_\ell \mid f(e \mid S_\ell) \geq \frac{\varepsilon}{k}\tau\}$
10: $E_\ell \leftarrow \{e \in A_\ell \mid S_\ell + e \in \mathcal{M}\}$
11: $F_\ell \leftarrow \{e \in A_\ell \setminus E_\ell \mid w(e) > 2\,w(s_e)\}$
12: $A_\ell \leftarrow E_\ell \cup F_\ell$
13: **if** $|A_\ell| \geq \frac{n}{2^\ell}$ **then**
14: Pop e from A_ℓ uniformly at random
15: $S_\ell \leftarrow S_\ell + e - s_e$
16: $S'_\ell \leftarrow S'_\ell + e$
17: **until** $|A_\ell| < \frac{n}{2^\ell}$
18: **if** $\ell < L$, call LEVEL-CONSTRUCT($\ell+1$).

Corollary 6.2. *For any constant $\varepsilon > 0$, there exists a $(4+O(\varepsilon))$-approximation to the fully dynamic monotone submodular maximization problem with matroid constraint that exhibits an $O\left(\frac{k^2}{\varepsilon}\log k \log^2 n \log^3 \frac{k}{\varepsilon}\right)$ expected amortized running time.*

Proof. Let us start by the running time analysis. The crucial place of the analysis in the main body where Δ appears is when we use $\log\Delta$ to bound the length of any chain of swaps in any repeat loop of LEVEL-CONSTRUCT (Lemma 5.5 and Lemma 5.8). This is because each swap increase the weight by at least a factor 2. Let n_j denote the number of elements inserted to the jth copy of the algorithm, corresponding to the threshold $\tau = (1+\varepsilon)^j$. Our claim is that line 9 enables us to bound the length of any chain of swaps by $\log\frac{k}{\varepsilon}$. The argument is as follows: each element inserted into that copy of the algorithm has value (and, by submodularity also weight) at most $(1+\varepsilon)\tau = (1+\varepsilon)^{j+1}$. Conversely, only swaps with weight at least $\frac{\varepsilon}{k}\tau = \frac{\varepsilon}{k}(1+\varepsilon)^j$ can happen (by the filter in line 9 of LEVEL-CONSTRUCT). Putting these two bounds together, we have the desired upper bound on the length of any chain of swaps.

The previous argument allows us to replace $O(\log \Delta)$ with $O(\log \frac{k}{\varepsilon})$ in the analysis of the running time of each copy of the algorithm (Theorem 6.1): there the total running time is then $O(n_j \cdot k^2 \log k \log^2 \frac{k}{\varepsilon} \log^2 n_j)$. Summing up over all the copies, we have that the total running time (up to multiplicative constants) is

$$\sum_{j:n_j>0} n_j k^2 \log k \log^2 \frac{k}{\varepsilon} \log^2 n_j \leq \left(k^2 \log k \log^2 n \log^2 \frac{k}{\varepsilon}\right) \sum_{j:n_j>0} n_j \leq \left(k^2 \log k \log^2 n \log^3 \frac{k}{\varepsilon}\right) \cdot \frac{n}{\varepsilon}$$

where in the first inequality inclusion we used the simple bound $n_i \leq n$ and the second one follows from Observation B.1. Therefore the amortized running time is

$$O\left(\frac{k^2}{\varepsilon} \log k \log^2 n \log^3 \frac{k}{\varepsilon}\right).$$

It remains to show that our new algorithm does not significantly worsen the approximation algorithm of the algorithm we presented in the main body. Fix any operation i, and consider the copy of the algorithm corresponding to a threshold τ that lies in the interval $[\mathrm{OPT}_i/(1+\varepsilon), \mathrm{OPT}_i]$. We first observe that any element $e \in V_i$ (any element currently part of the instance) cannot have $f(e) > (1+\varepsilon)\tau$, since:

$$f(e) \leq \mathrm{OPT}_i \leq (1+\varepsilon)\tau\,.$$

Therefore, ignoring element e such that $(1+\varepsilon)\tau > f(e)$ does not affect the approximation guarantee. The elements whose value falls below $\frac{\varepsilon}{k}\tau \geq \frac{\varepsilon}{k}\,\mathrm{OPT}_i/(1+\varepsilon)$ (and thus not inserted in this copy of the algorithm due to the filter in line 1 of INSERTION), on the other hand, only cause an extra additive $O(\varepsilon)$ error in the approximation and can be ignored (similarly to what has been used in the proof of Theorem A.1). Finally, from Theorem A.1 we know that the filter in line 11 of LEVEL-CONSTRUCT only cause in another additive $O(\varepsilon)$ error in the guarantee of SWAPPING. Using the same argument in Section 4 we can then conclude that our algorithm mutuates the approximation guarantees of THRESHOLD-SWAPPING and thus has a $4 + O(\varepsilon)$-approximation guarantee. □

C. LAZY-GREEDY and SWAPPING fails in the dynamic setting

In this section we show how two well known algorithms for submodular maximization subject to matroid constraints (LAZY-GREEDY and SWAPPING) cannot be directly applied to the dynamic setting without suffering $\Omega(n)$ worst case update time.

We define the two fully-dynamic algorithms: DYNAMIC-SWAPPING and DYNAMIC-GREEDY. They both maintain the set V_i of elements that were inserted but not deleted in the first i operations and a candidate solution. When an element from the candidate solution gets discarded they both recompute from scratch a feasible solution from V_i using their non-fully-dynamic counterpart: DYNAMIC-SWAPPING uses SWAPPING on some ordering of the elements in V_i, while DYNAMIC-GREEDY performs LAZY-GREEDY on V_i. The two algorithms differ on how they handle insertions. When an element is inserted, DYNAMIC-GREEDY recomputes the solution from scratches on V_i as the new element may have changed drastically the greedy structure of the instance. On the contrary, DYNAMIC-SWAPPING simply processes this new element as SWAPPING would do; this is because SWAPPING is a streaming algorithm and thus handles efficiently insertions.

We construct here an instance where both DYNAMIC-SWAPPING and DYNAMIC-GREEDY exhibits an amortized running time that is $\Omega(n)$. Consider a set $V = \{x_1, \ldots, x_n\}$ of n elements and an additive function f on it, with $f(x_i) = 3^i$. The stream is simple: the elements are inserted one after the other in increasing order $x_1, x_2, \ldots$ and are then deleted one after the other in decreasing order $x_n, x_{n-1}, \ldots$ We show that both algorithms have a large per-operation running time on this instance.

Claim C.1. DYNAMIC-SWAPPING has worst case amortized running time that is $\Omega(n)$

Proof. Consider the stream and the function presented above, with a cardinality constraint $k = 1$. We divide the analysis of the running time into two parts. During the first n operations, the algorithm performs at least one value call and one independence call. For each one of the deletion operations, the deleted element is — by construction — in the solution maintained by DYNAMIC-SWAPPING. Let's call x_i this deleted element, the routine SWAPPING is called on the sequence $x_1, \ldots, x_{i-1}$, where it performs $\Omega(i)$ oracle calls. Summing up, the total number of oracle calls (both value and independence calls) is $\Omega(n^2)$, yielding an amortized running time of $\Omega(n)$. □

Claim C.2. DYNAMIC-GREEDY has worst case amortized running time that is $\Omega(n)$

Proof. Consider the stream and the function presented above, with a cardinality constraint $k = 1$.Also here, we divide the analysis of the running time into two parts. During the first n operations, the algorithm performs at least one value call and one independence call (actually way more than that). For each one of the deletion operations, the deleted element is — by construction — in the solution maintained by DYNAMIC-GREEDY. Let's call x_i this deleted element, the routine LAZY-GREEDY is called on the sequence $x_1, \ldots, x_i$, where it performs $\Omega(i)$ oracle calls. All in all, the total number of oracle calls (both value and independence calls) is $\Omega(n^2)$, yielding an amortized running time of $\Omega(n)$. □