Improved Fully Dynamic Submodular Maximization Under Matroid Constraints

Yiwei Gao[1,2], **Jialin Zhang**[1, 2], **Zhjie Zhang**[3*]

[1]State Key Lab of Processors, Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190, China
[2]School of Computer Science and Technology, University of Chinese Academy of Sciences, Beijing 100049, China
[3]Center for Applied Mathematics of Fujian Province, School of Mathematics and Statistics, Fuzhou University
gaoyiwei22@mails.ucas.ac.cn, zhangjialin@ict.ac.cn, zzhang@fzu.edu.cn

Abstract

This paper studies submodular maximization over matroids in the fully dynamic setting, where elements of an underlying ground set undergo sequential insertions and deletions. The goal is to maintain an approximate optimal solution for the current element set with a low amortized update time. For monotone submodular functions. we propose a dynamic algorithm achieving a $(0.3178-\varepsilon)$-approximation using $\tilde{O}_\varepsilon(k^3)$ expected amortized queries, where k is the rank of the matroid contraint. Furthermore, we extend our approach to the non-monotone submodular maximization setting, obtaining a $(0.1921-\varepsilon)$-approximation with the same update complexity. Both algorithms improve upon the best known approximation guarantees, which are $(0.25-\varepsilon)$ for the monotone case and $(0.0932-\varepsilon)$ for the non-monotone case.

Introduction

Submodularity is a fundamental property that is possessed by many objective functions of combinatorial optimization problems. It captures the real-world phenomenon of *diminishing returns*, making it applicable across a range of practical contexts, including data summarization (Bairi et al. 2015; Kumari and Bilmes 2021), influence maximization (Kempe, Kleinberg, and Tardos 2003), sparse reconstruction (Bach 2010; Das and Kempe 2011), feature selection (Khanna et al. 2017; Das and Kempe 2018), information gathering (Radanovic et al. 2018), video analysis (Zheng et al. 2014). Due to their elegant structures, matroids often arise as the constraints of these problems. As a result, submodular maximization subject to a matroid constraint has been one of the central topics in the past decades.

Many excellent results on approximating the optimal solution of the problem have emerged. For monotone submodular maximization, the groundbreaking work of Vondrák (2008) proposed the famous continuous greedy algorithm, which attains $(1-1/e)$ approximation under matroid constraints. This is the best ratio one can achieve with a polynomial number of queries (Nemhauser and Wolsey 1978). For non-monotone submodular maximization, the best-known algorithm achieves 0.401 approximation (Buchbinder and Feldman 2024), while any polynomial algorithm can not achieve an approximation ratio better than 0.478 (Gharan and Vondrák 2011; Qi 2022).

Recently, submodular maximization under the *(fully) dynamic model* has aroused researchers' interest (Lattanzi et al. 2020a; Monemizadeh 2020). The dynamic model assumes that there is a stream of insertions and deletions of elements in the ground set $\mathcal{N}$. Let $\mathcal{N}_t \subseteq \mathcal{N}$ be the set of elements that have been inserted but not deleted subsequently until time t. The goal is to maintain a feasible solution $S_t \in \mathcal{N}_t$ that (approximately) maximizes the submodular objective function with an amortized update time of $O(\mathrm{poly}(k), \mathrm{polylog}(n))$.

Significant progresses has been made on the dynamic monotone submodular maximization for the cardinality constraint, also known as the uniform matroid. A series of works (Lattanzi et al. 2020a; Monemizadeh 2020; Banihashem et al. 2023a, 2024) proposed a variety of dynamic algorithms that maintains $(1/2-\varepsilon)$-approximation using low amortized queries. The ratio is optimal in the sense that any algorithm that achieves a $(1/2+\varepsilon)$-approximation for the problem requires an amortized query complexity of $n^{\tilde{\Omega}(\varepsilon)}/k^3$ (Chen and Peng 2022). For general matroids, however, the best-known dynamic algorithms (Duetting et al. 2023; Banihashem et al. 2024) achieves only $(1/4-\varepsilon)$-approximation, leaving a substantial room for improvement. For non-monotone submodular maximization, the best-known dynamic algorithms maintain $(1/8-\varepsilon)$-approximation for the cardinality constraints (Banihashem et al. 2023b) and $(0.0932-\varepsilon)$-approximation for general matroid constraints (Liu and Yang 2024).

Our Contributions. We propose improved dynamic algorithms for submodular maximization subject to a matroid constraint. Specifically,

1. We are the first to show that a streaming algorithm based on the multilinear extension can be successfully maintained within the dynamic framework of (Banihashem et al. 2024)
2. Building on this, we develop two dynamic algorithms for fully dynamic submodular maximization under a matroid constraint: a $(0.3178-\varepsilon)$-approximation for the monotone setting and a $(0.1921-\varepsilon)$-approximation for the non-monotone setting, both with amortized query complexity $\tilde{O}_\varepsilon(k^3)$.

*Corresponding author

Both of our algorithms achieve substantially better approximation ratios than all existing methods, while maintaining average query complexity that is completely independent of n.

Related Work. Lattanzi et al. (2020a) and Monemizadeh (2020) independently initiated the study of submodular maximization in the dynamic setting. For monotone submodular maximization under the cardinality constraint, Monemizadeh (2020) proposed a dynamic algorithm that attains $(1/2 - \varepsilon)$-approximation with amortized query complexity of $O(\varepsilon^{-3}k^2 \log^5 n)$. Lattanzi et al. (2020a) presented an alternative dynamic algorithm that also achieves $(1/2 - \varepsilon)$-approximation, but has a polylogarithmic update time of $O(\varepsilon^{-11} \log^6 k \log^2 n)$. Later, Banihashem et al. (2023a) pointed out that the analysis of Lattanzi et al. (2020a) has some correctness issues, and then presented another dynamic algorithm with polylogarithmic update time. Lattanzi et al. also remedied their algorithm in the latest arXiv version of their paper (Lattanzi et al. 2020b), achieving an improved update time of $O(\varepsilon^{-4} \log^4 k \log^2 n)$. After that, Banihashem et al. (2024) proposed a novel dynamic algorithm whose amortized query complexity is $O(\varepsilon^{-1}k \log^2 k)$, which is the first fully dynamic algorithm with an update time independent of n. These results match the hardness result of Chen and Peng (2022), who showed that any dynamic algorithm that achieves a $(1/2 + \varepsilon)$-approximation for the problem requires amortized query complexity of $n^{\tilde{\Omega}(\varepsilon)}/k^3$.

For monotone submodular maximization under the matroid constraint, Duetting et al. (2023) presented a dynamic algorithm whose approximation ratio is $1/4 - \varepsilon$ and amortized query complexity is $O(\varepsilon^{-1}k^2 \log^2 n \log^3(\varepsilon^{-1}k))$. Indepenently, Banihashem et al. (2024) provided an improved algorithm with an $O(k \log k \log^3(\varepsilon^{-1}k))$ update time.

For non-monotone submodular maximization, Banihashem et al. (2023b) proposed a dynamic algorithm under the cardinality constraint, which achieves $(1/8 - \varepsilon)$-approximation and requires $O(\varepsilon^{-1}k \log^2 k)$ oracle queries. Liu and Yang (2024) presented an improved algorithm that achieves $(1/6 - \varepsilon)$-approximation and requires $O(\varepsilon^{-1}k^2 \log^2 k)$ amortized oracle queries. Liu and Yang (2024) also presented a dynamic algorithm for the matroid constraint, which attains $(0.0932 - \varepsilon)$-approximation and has an $O(\varepsilon^{-1}k^2 \log k \log^3(\varepsilon^{-1}k))$ update time.

Preliminaries

Submodular Functions. Given a ground set $\mathcal{N}$, a set function $f : 2^{\mathcal{N}} \to \mathbb{R}$ maps every element of the power set of $\mathcal{N}$ to a real number. Submodularity captures the property of diminishing marginal returns in set functions. For clarity, we use the notation $S + u$ to denote $S \cup \{u\}$, and $S - u$ to denote $S \setminus \{u\}$. We define the marginal contribution of a set S with respect to another set T as $f(S \mid T) := f(S \cup T) - f(T)$. In particular, the marginal contribution of a single element u with respect to a set T is given by $f(u \mid T) := f(T \cup \{u\}) - f(T)$. A set function $f : 2^{\mathcal{N}} \to \mathbb{R}$ is called *submodular* if for any two sets $S \subseteq T \subseteq \mathcal{N}$ and an element $u \in \mathcal{N} \setminus T$ we have

$$f(u \mid S) \geq f(u \mid T).$$

In addition, f is called *monotone* if for any sets $S \subseteq T$, we have $f(S) \leq f(T)$. f is called *non-negative* if $f \geq 0$.

Matroids. A *matroid* $\mathcal{M} = (\mathcal{N}, \mathcal{I})$ is a set system where $\mathcal{I} \subseteq 2^{\mathcal{N}}$ is a family of subsets of the ground set $\mathcal{N}$, which satisfies the following properties:

- $\emptyset \in \mathcal{I}$;
- If $A \subseteq B$ and $B \in \mathcal{I}$, then $A \in \mathcal{I}$;
- If $A, B \in \mathcal{I}$ with $|A| < |B|$, there exists $u \in B \setminus A$ such that $A + u \in \mathcal{I}$.

The members of $\mathcal{I}$ are called *independent sets*. The maximal independent sets are called *bases*. It can be proven that all the bases share a common cardinality, called the *rank* of $\mathcal{M}$. For a subset $S \subseteq \mathcal{N}$ (not necessarily independent), its *rank* is defined as the size of its maximum independent subset, i.e. $\mathrm{rank}(S) = \max_{T \subseteq S, T \in \mathcal{I}} |T|$. The *span* of a set $S \subseteq \mathcal{N}$ in a matroid is the set of elements $u \in \mathcal{N}$ such that $\mathrm{rank}(S \cup \{u\}) = \mathrm{rank}(S)$.

Submodular Maximization under Matroid Constraints. The problem we are concerned with can be formulated as $\max\{f(S) : S \in \mathcal{I}\}$, where f is a non-negative submodular function and $\mathcal{M} = (\mathcal{N}, \mathcal{I})$ forms a matroid defined over the ground set $\mathcal{N}$. We use OPT to denote the optimal solution of the problem. We assume that f is accessed via a *value oracle* that returns $f(S)$ when S is queried, and $\mathcal{I}$ is accessed by a *membership oracle* that returns whether $S \in \mathcal{I}$. We use the number of queries to the oracles an algorithm makes to measure the algorithm's complexity. It is well-known that for maximizing a monotone submodular function with matroid constraint, any algorithm that achieves an approximation exceeding $1 - 1/e$ requires exponentially many queries (Nemhauser and Wolsey 1978).

Multilinear Extension. A canonical way to solve submodular maximization is to transform it into a continuous problem. We now introduce the tools used in this approach. For a vector $x \in [0,1]^{\mathcal{N}}$, let $\mathcal{R}(x)$ be a random subset of $\mathcal{N}$ where each element $u \in \mathcal{N}$ is included in $\mathcal{R}(x)$ independently with probability x_u. For a set function $f : 2^{\mathcal{N}} \to \mathbb{R}$, its *multilinear extension* $F : [0,1]^{\mathcal{N}} \to \mathbb{R}$ is defined as the expected value of f over the random set $\mathcal{R}(x)$, that is,

$$\begin{aligned} F(x) &= \mathbb{E}[f(\mathcal{R}(x))] \\ &= \sum_{S \subseteq \mathcal{N}} \left[f(S) \cdot \prod_{u \in S} x_u \cdot \prod_{u \notin S} (1 - x_u) \right]. \end{aligned}$$

Although computing the exact value of $F(x)$ requires exponentially many queries to f, it can be efficiently approximated by Monte Carlo sampling. For any vector $x \in [0,1]^{\mathcal{N}}$, the partial derivative of F at coordinate u satisfies $\partial_u F(x) = F(x + (1 - x_u) \cdot \mathbf{1}_u) - F(x - x_u \cdot \mathbf{1}_u)$, where $\mathbf{1}_u$ is the indicator vector for element u.

The matroid polytope of $\mathcal{M}$ is defined as

$$P_{\mathcal{M}} := \left\{ x \in \mathbb{R}^{\mathcal{N}}_{\geq 0} : \sum_{u \in S} x_u \leq \mathrm{rank}(S), \forall S \subseteq \mathcal{N} \right\}.$$

functions	constraints	approximation ratio	amortized query complexity	reference
monotone	cardinality	$0.5-\varepsilon$	$O(\varepsilon^{-3}k^2\log^5 n)$	Monemizadeh (2020)
		$0.5-\varepsilon$	$O(\varepsilon^{-11}\log^6 k\log^2 n)$	Banihashem et al. (2023a)
		$0.5-\varepsilon$	$O(\varepsilon^{-4}\log^4 k\log^2 n)$	Lattanzi et al. (2020b)
		$0.5-\varepsilon$	$O(\varepsilon^{-1}k\log^2 k)$,	Banihashem et al. (2024)
	matroid	$0.25-\varepsilon$	$O(\varepsilon^{-1}k^2\log^2 n\log^3(\varepsilon^{-1}k))$	Duetting et al. (2023)
		$0.25-\varepsilon$	$O(k\log k\log^3(\varepsilon^{-1}k))$	Banihashem et al. (2024)
		$0.3178-\varepsilon$	$O\left(\varepsilon^{-2}k^3\log^5(k/\varepsilon)\log^2 k\right)$	**this paper**
non-montone	cardinality	$0.125-\varepsilon$	$O(\varepsilon^{-1}k\log^2 k)$	Banihashem et al. (2023b)
		$0.1667-\varepsilon$	$O(\varepsilon^{-1}k^2\log^2 k)$	Liu and Yang (2024)
	matroid	$0.0932-\varepsilon$	$O(\varepsilon^{-1}k^2\log k\log^3(\varepsilon^{-1}k))$	Liu and Yang (2024)
		$0.1921-\varepsilon$	$O\left(\varepsilon^{-2}k^3\log^5(k/\varepsilon)\log^2 k\right)$	**this paper**

Table 1: The approximation ratio and update complexity of both previous and our dynamic algorithms

Given vector $x \in P_{\mathcal{M}}$, Cualinescu et al. (2011) proposed the so-called PIPAGEROUNDING that returns a discrete solution $S \in \mathcal{I}$ satisfying $f(S) \geq F(x)$ without making any queries to f.

Streaming and Dynamic Model. In the streaming model for submodular maximization, elements from the ground set arrive one by one in fixed order. Upon arrival of each element, the algorithm must determine whether to retain or permanently discard it, while maintaining at most $O(\text{poly}(k), \text{polylog}(n))$ elements in memory at all times. The algorithm may evaluate the submodular function on the subset of elements currently held in memory. After the stream terminates, the algorithm outputs a feasible set over the ground set with a guaranteed approximation ratio.

In the fully dynamic model, the algorithm receives a stream of both insertions and deletions of elements. After each update, it must output a solution over the current element set. Formally, the algorithm proceeds for T rounds. In each round $1 \leq i \leq T$, there is a current ground set $\mathcal{N}_i \subseteq \mathcal{N}$, which evolves by either inserting or deleting a single element, i.e., $\mathcal{N}_{i+1} = \mathcal{N}_i + u$ or $\mathcal{N}_{i+1} = \mathcal{N}_i - u$ for some element u. A dynamic algorithm is said to achieve a γ-approximation if, in every round i, it outputs a solution S_i such that $f(S_i) \geq \gamma f(\text{OPT}_i)$, where OPT_i denotes the optimal solution with respect to the ground set $\mathcal{N}_i$. we use π to denote a stream over the ground set $\mathcal{N}$, and similarly, we use π_d to denote a dynamic stream that includes both insertions and deletions. The algorithm also requires only an average of $O(\text{poly}(k), \text{polylog}(n))$ queries per update.

Dynamic Algorithm for Monotone Submodular Maximization

In this section, we describe our dynamic algorithm for montone functions; The analysis of the approximation guarantee and query complexity will be presented in the following section. Throughout this and the next section, we use the fixed constants $\varepsilon \in (0, 1]$ and $\alpha \approx 1.14$, where α is the unique positive solution to the equation $\alpha + 2 = e^{\alpha}$. We begin this section by assuming access to the exact value of $f(\text{OPT})$; we will later show how this assumption can be eliminated. For convenience, we denote $f(\text{OPT})$ by τ whenever no ambiguity arises. We further define the following two functions: $\text{lb}(x) = \left\lfloor \log_{1+\varepsilon}\left(\frac{\varepsilon x}{k}\right)\right\rfloor$ and $\text{ub}(x) = \left\lceil \log_{1+\varepsilon}(x)\right\rceil$.

Dynamic Data Structure. Our algorithm reduces the amortized query complexity by maintaining a data structure. The data structure consists of $T = O(k\log(k/\varepsilon))$ levels. At each level i, it maintains a vector $a_i \in [0,1]^{\mathcal{N}}$, $O(\log(k/\varepsilon))$ independent sets $\{A_{j,i}\}_{j=\text{lb}(\tau)}^{\text{ub}(\tau)}$[1], a candidate set R_i, and a selected element u_i.

We begin by introducing our PICK algorithm, which is primarily used to compute a_i and $\{A_{j,i}\}$ at level i from a_{i-1} and $\{A_{j,i-1}\}$ at level $i-1$ using the selected element u_i. The algorithm takes as input a value a, a collection $\{A_j\}$, and an element u. It first queries the value of $f(u)$. If $f(u)$ does not lie in the interval $[\varepsilon\tau/k, \tau]$, the algorithm immediately returns **False**. Otherwise, it attempts to insert u into each A_j such that $(1+\varepsilon)^i$ lies within the interval $(-\infty, \lfloor \log_c(\partial_u F(a))\rfloor \cap [\text{lb}(\tau), \text{ub}(\tau)]$, as long as the insertion preserves the matroid independence. If u is successfully inserted into some A_j, we increment the coordinate of a corresponding to u by $\frac{\varepsilon\cdot(1+\varepsilon)^i}{\alpha\cdot\partial_u F(a)}$. If the above process results in any change to the sets A_i, the algorithm returns the updated vector a and the collection A_i. Otherwise, it returns **False**.

This algorithm can be used not only to acquire the i-th level's data from the $(i-1)$-th level, but also to filter which elements in the $(i-1)$-th level can be used for effective updates. We say that an element u can be *picked at level* i if

$$\text{PICK}(a_{i-1}, \{A_{j,i-1}\}, u, \tau) \neq \textbf{False}.$$

For convenience, we let Level 0 be the initialization level, where $a_0 = 0$ and $A_j = \emptyset$ for all i. For each $1 \leq i \leq T$, R_i records all elements in R_{i-1} that can be picked at level i. Then, u_i is selected from R_i. It can perform a valid PICK at level i, and the result $\text{PICK}(a_i, \{A_{j,i}\}, u_i, \tau)$ is stored as $\{a_{i+1}, \{A_{j,i+1}\}\}$.

[1]Throughout the rest of the paper, we encounter collections of sets denoted by $\{A_{j,i}\}_{j=\text{lb}(\tau)}^{\text{ub}(\tau)}$ and $\{A_j\}_{j=\text{lb}(\tau)}^{\text{ub}(\tau)}$ multiple times, where the index j always ranges from $\text{lb}(\tau)$ to $\text{ub}(\tau)$. For notational elegance, we write them as $\{A_{j,i}\}$ and $\{A_i\}$, where the first subscript indicates the enumeration over the sets.

As this data structure extends level by level, note that there are $\log_{1+\varepsilon}(k/\varepsilon)$ sets $\{A_{j,i}\}$, and each $A_{j,i}$ is an independent set containing at most k elements. Moreover, each level update inserts at least one element into some $A_{j,i}$. Therefore, after at most $k\log_{1+\varepsilon}(k/\varepsilon)$ levels, no further elements can be picked. In practice, the process may terminate earlier if $R_i = \emptyset$ for some level $i < k\log_{1+\varepsilon}(k/\varepsilon)$. We define this level as the final level T of the data structure.

Upon reaching the final level T, we obtain a vector $a_T \in [0,1]^{\mathcal{N}}$ such that $F(a_T) \geq \left(\frac{1}{1+\alpha} - O(\varepsilon)\right)\tau$. Direct rounding cannot be applied to a_T since it is only guaranteed to lie in the hypercube $[0,1]^{\mathcal{N}}$, and not necessarily in the matroid polytope $\mathcal{P}(\mathcal{M})$.

To overcome this issue, we apply a procedure called PACK to the collection $\{A_{j,T}\}$. Specifically, the sets in $\{A_{j,T}\}$ are partitioned into $m \coloneqq \lceil \alpha/\varepsilon \rceil$ groups, where each group consists of sets whose indices are congruent modulo m. For each group of sets of the form $A_{j,T}, A_{j+m,T}, A_{j+2m,T}, \ldots$, the algorithm initializes an empty set S_i and processes the sets in decreasing order of indices, starting from the highest-indexed set $A_{j+tm,T}$ down to $A_{j,T}$, inserting elements into S_i and keeping S_i to be an independent set. This ensures that the average of the indicator vectors $s \coloneqq \frac{1}{m}\sum_{i=1}^{m} \mathbf{1}_{S_i}$ achieves multilinear value at least $(1 - e^{-\alpha} - O(\varepsilon))f(a)$, and lies within the matroid polytope as each S_i is a feasible independent set. Finally, we apply PIPAGEROUNDING(s) to obtain a discrete, feasible solution. Combined with the above guarantees, this yields a solution with provable approximation ratio. We note that the PACK algorithm incurs only $O_\varepsilon(k^2)$ oracle queries, and thus we do not need to maintain the sets S_i within the data structure. It suffices to run PACK on $\{A_{j,T}\}$ once at the end of update.

We remark that both the PICK and PACK procedures are adapted from the streaming algorithm in Feldman et al. (2022). We just modify them to be compatible with our data structure.

Back to the data structure, if an update operation removes some element u_i, it appears that we must reconstruct all levels following i in order to maintain correctness. If we were to reconstruct from level one after every update, this would incur at least an $O(n)$ overhead. To avoid this, we ensure that in each update, the element u_i is selected uniformly at random from R_i. As a result, a deletion triggers reconstruction at level i with probability only $1/|R_i|$. Consequently, the earlier the level, the lower the probability of reconstruction, since the candidate sets satisfy $R_i \subseteq R_{i+1}$.

We now describe how to reconstruct the data structure starting from level i. The corresponding procedure is given in Algorithm 4. The algorithm begins by generating a random permutation of the elements in R_i, and process them one by one. Suppose we are currently at level ℓ. If the current element u can be picked at level ℓ and meet the element u, we update a_ℓ and $A_{j,\ell}$ accordingly using u, and proceed to level $\ell+1$ to process the next element. At this point, we postpone the computation of $R_{\ell+1}$. If, on the other hand, u cannot be picked at level ℓ, we perform a binary search backwards to find the most recent level $z \leq \ell$ where u can still be picked. Once such a level is found, we add u to all sets R_j for $i < j \leq z$. The correctness of this reconstruction algorithm relies on a key structural property: for each element u, the sets R_i satisfy a binary-searchable structure. Specifically, if u can be picked at some level z, then it must also be pickable at every earlier level $j < z$. We will formally prove this property in our analysis.

The algorithm finally reconstructs all information for level $\ell \geq i$. Moreover, since the reconstruction is driven by a random permutation over R_i, the selected element u_i still preserves the original randomness assumption of being uniformly drawn from R_i.

Algorithm 1: INITIALIZE(τ)

$a_i \leftarrow 0 \in [0,1]^{\mathcal{N}}, A_{i,j} \leftarrow \emptyset, R_i \leftarrow \emptyset$.

Algorithm 2: PICK($a, \{A_j\}, u, \tau$)

if $f(u) < \varepsilon\tau/k$ **or** $f(u) > \tau$ **then**
  return False.
$j(u) \leftarrow \lfloor \log_c(\partial_u F(a)) \rfloor$.
for i **from** $\mathrm{lb}(\tau)$ **to** $\min\{j(u), \mathrm{ub}(\tau)\}$ **do**
  if $A_j + u \in \mathcal{I}$ **then**
    $A_j \leftarrow A_j + u$.
    $a \leftarrow a + \frac{\varepsilon\cdot(1+\varepsilon)^i}{\alpha\cdot\partial_u F(a)}\mathbf{1}_u$.
if *the above* **for loop** *results in no changes to any* A_i **then**
  return False.
else
  return $\{a, \{A_i\}\}$.

Algorithm 3: PACK($\{A_j\}, \tau$)

$m \leftarrow \lceil \alpha/\varepsilon \rceil$.
$S_j \leftarrow \emptyset$ for $j \in \{0, ..., m\}$.
for *i* ***from*** $\mathrm{ub}(\tau)$ ***to*** $\mathrm{lb}(\tau)$ **do**
  while $\exists u \in A_j \setminus S_{(j \bmod m)}$ *with* $S_{(j \bmod m)} + u \in \mathcal{I}$ **do**
    $S_{(j \bmod m)} \leftarrow S_{(j \bmod m)} + u$.
$s \leftarrow \frac{1}{m}\sum_{j=1}^{t} \mathbf{1}_{S_j}$.
return PIPAGEROUNDING(s).

Insertion. When an element u is inserted, we traverse the data structure from level 1 to T. At level i, if u can't be picked , then it can't be picked anymore at any deeper level either, and can thus be safely ignored. Otherwise, we attempt to insert it into R_i to maintain a uniform distribution: we add u to R_i and select it as the new u_i with probability $\frac{1}{|R_i|}$. If u is selected as u_i, and then reconstruct the data structure from level i. Else, we proceed to the next level.

Algorithm 4: MATROIDCONSTRUCTLEVEL(i, τ)

Let P be a random permutation of elements of R_i.
$\ell \leftarrow i$.
for u **in** P **do**
  if PICK$(a_{\ell-1}, \{A_{j,\ell-1}\}, u, \tau) \neq$ **False then**
    $\{a_\ell, \{A_{j,\ell}\}\} \leftarrow$ PICK$(a_{\ell-1}, \{A_{j,\ell-1}\}, u, \tau)$.
    $R_{\ell+1} \leftarrow \emptyset, \ell \leftarrow \ell + 1, u_\ell \leftarrow u$.
  else
    Run binary search to find the lowest $z \in [i, \ell - 1]$ such that PICK$(a_z, \{A_{j,z}\}, u, \tau) =$ **False**.
    for r **from** $i + 1$ **to** z **do**
      $R_r \leftarrow R_r + u$.
return $T \leftarrow \ell - 1$ which is the final ℓ that the for-loop above returns subtracted by one.

Algorithm 5: INSERT(u, τ)

$R_0 \leftarrow R_0 + u$.
for i **from** 1 **to** $T + 1$ **do**
  if PICK$(a_{i-1}, \{A_{j,i-1}\}, u, \tau) =$ **False then**
    break.
  $R_i \leftarrow R_i + u$.
  Let p_i with probability $\frac{1}{|R_i|}$, and otherwise $p_i = 0$.
  if $p_i = 1$ **then**
    $u_i \leftarrow u$.
    $\{a_i, \{A_{j,i}\}\} \leftarrow$ PICK$(a_{i-1}, \{A_{j,i-1}\}, u, \tau)$.
    $R_{i+1} \leftarrow \{v \in R_i :$ PICK$(a_{i-1}, \{A_{j,i}\}, v, \tau) \neq$ **False**$\}$.
    MATROIDCONSTRUCTLEVEL$(i + 1)$.
return PACK$(\{A_{j,T}\})$.

Deletion. The deletion operation is simpler: we iterate from level 1 to T and remove the element u from each R_i until either $u \notin R_i$ or $u = u_i$. In the former case, we terminate the process; in the latter, we trigger a reconstruction starting from level i. This procedure naturally preserves the uniformity of the selected representatives u_i, since in levels where no reconstruction occurs, we are merely removing u from the candidate set, which does not affect the uniform distribution of the remaining elements.

After the insertion or deletion, we send the collection $\{A_{j,T}\}$ into the PACK algorithm to obtain the final result.

Relaxing the assumption of known $f(\mathrm{OPT})$. Finally, we discuss how to eliminate the assumption that $\tau = f(\mathrm{OPT})$. The technique we employ was first introduced by (Monemizadeh 2020) and has been widely adopted in the context of dynamic algorithms for submodular maximization.

First, it is easy to show that if we are given an estimate $\tau' \in [f(\mathrm{OPT}), (1 + \varepsilon) f(\mathrm{OPT})]$ instead of the exact value $f(\mathrm{OPT})$, then the resulting loss in the approximation guarantee is only additive in ε.

Algorithm 6: DELETE(u, τ)

$R_0 \leftarrow R_0 - u$.
for i **from** 1 **to** $T + 1$ **do**
  if $u \notin R_i$ **then**
    return False.
  $R_i \leftarrow R_i - u$.
  if $u_i = u$ **then**
    MATROIDCONSTRUCTLEVEL(i).
return PACK$(\{A_{j,T}\})$.

We then assume that there exist infinitely many parallel instances of the dynamic algorithm, each indexed by an integer i, where the i-th instance assumes $\tau = (1 + \varepsilon)^i$ as an estimate for $f(\mathrm{OPT})$, and each process maintains its own data structure independently.

Whenever an element u is inserted or deleted, we only need to update those processes i for which

$$f(u) \in \left[\varepsilon(1 + \varepsilon)^i / k, (1 + \varepsilon)^i\right],$$

and the number of such processes is at most $O(\log_{1+\varepsilon}(k/\varepsilon))$. In all other processes, the element u will be filtered out during the first step of PICK. Therefore, it suffices to perform the insertion or deletion operation only for these relevant instances. Putting everything together, our final algorithm is presented as Algorithm 7.

Algorithm 7: DYNAMICMATROID$(\mathcal{M}(\mathcal{N}, \mathcal{I}), \pi_d)$

Let i denote whether the process with the guess $f(\mathrm{OPT}) = (1 + \varepsilon)^i$.
for *all command in* π_d **do**
  if *the command is* ***insert*** *the element* u **then**
    for *i with* $f(u) \in [\frac{\varepsilon(1+\varepsilon)^i}{k}, (1 + \varepsilon)^i]$ **do**
      INSERT$(u, (1 + \varepsilon)^i)$.
  if *the command is* ***delete*** *the element* u **then**
    for *i with* $f(u) \in [\frac{\varepsilon(1+\varepsilon)^i}{k}, (1 + \varepsilon)^i]$ **do**
      DELETE$(u, (1 + \varepsilon)^i)$.

The following theorem summarizes the guarantees of our algorithm. We defer its proof to the next section.

Theorem 1. *For a monotone submodular function f and a fully dynamic stream with insertion and deletion,* DYNAMICMATROID *guarantees a $(0.3178 - O(\varepsilon))$-approximation and have amortized query complexity $O(\varepsilon^{-2} k^3 \log(k) \log^3(k/\varepsilon))$.*

Analysis

In this section, we prove Theorem 1. We first prove the correctness of the binary search procedure in Algorithm 4. Then, we explain the properties maintained by the data structure after each dynamic update. Finally, we leverage these properties to derive the algorithm's approximation guarantee and its average query complexity.

Correctness of Binary Search. We first observe that during the execution of algorithm 4 at level i, the stored vectors and sets from level i to level T are monotonically increasing.

Observation 2. *After the execution of* MATROIDCONSTRUCTLEVEL(x, τ)*, for any* $x \leq y \leq z \leq T$ *and* $\mathrm{lb}(\tau) \leq j \leq \mathrm{ub}(\tau)$ *we have* $a_y < a_z$ *and* $A_{j,y} \subseteq A_{j,z}$.

Based on this observation, we proceed to prove the following lemma.

Lemma 3. *After the execution of* MATROIDCONSTRUCTLEVEL(x, τ)*, for any* $x \leq y \leq z \leq T$ *and any element* $u \in R_x$*, if* PICK$(a_x, \{A_{j,x}\}, u, \tau) =$ **False***, then* PICK$(a_y, \{A_{j,y}\}, u, \tau) =$ **False**.

Proof. PICK$(a_x, \{A_{j,x}\}, u, \tau) =$ **False** means that at least one of the following three situations must occur:

1. $f(u) \leq \varepsilon\tau/k$ or $f(u) \geq \tau$;
2. $\left\lfloor \log_{1+\varepsilon}(\partial_u F(a_x)) \right\rfloor \leq \mathrm{lb}(\tau)$;
3. for all $\mathrm{lb}(\tau) \leq j \leq \min\{\left\lfloor \log_{1+\varepsilon}(\partial_u F(a_x)) \right\rfloor, \mathrm{ub}(\tau)\}$, $A_{j,x} + u \notin \mathcal{I}$.

The first case directly leads to the conclusion we aim to prove. By submodularity of F and $a_x < a_y$ we have $\partial_u F(a_y) \leq \partial_u F(a_x)$. Hence, the second case implies that $\left\lfloor \log_{1+\varepsilon}(\partial_u F(a_y)) \right\rfloor \leq \mathrm{lb}(\tau)$, and combining with $A_{j,x} \subseteq A_{j,y}$ the third case impiles that $A_{j,y} + u \notin \mathcal{I}$ for all $\mathrm{lb}(\tau) \leq j \leq \min\{\left\lfloor \log_{1+\varepsilon}(\partial_u F(a_y)) \right\rfloor, \mathrm{ub}(\tau)\}$. □

Lemma 3 means that for any $u \in R_x$, there exist index $y > x$ such that, PICK$(a_z, \{A_{j,z}\}, u, \tau) =$ **False** if and only if $z \geq x$. Thus, we can perform a binary search to find this z in algorithm 4 to acquire R_i correctly.

Theorem 4. *After the execution of* MATROIDCONSTRUCTLEVEL(x, τ)*, we have*

$$R_i = \{u \in R_{i-1} : \textsc{Pick}(a_{i-1}, \{A_{j,i}\}, u, \tau) \neq \textbf{False}\}$$

for any $x < i \leq T$.

Maintaining Invariants. Now we can proceed to establish the invariants maintained by this data structure.

Theorem 5. *After each* INSERT *or* DELETE *operation, the data structure maintains the following properties for any* $1 \leq i \leq T$*:*

1. $T = O(k \log(k/\varepsilon))$.
2. $R_i = \{u \in R_{i-1} : \textsc{Pick}(a_{i-1}, \{A_{j,i}\}, u, \tau) \neq \textbf{False}\}$.
3. $\{a_i, \{A_{j,i}\}\} = \textsc{Pick}(a_{i-1}, \{A_{j,i-1}\}, u_i, \tau)$.
4. u_i *is picked uniformly at random from* R_i*. Formally, if we treat the contents of the data structure as random variables, with the randomness stemming from the algorithm's internal random seed, and denote them in boldface then for* $u \in R_i$*, we have* $\Pr[u = \mathbf{u_i} \mid \mathbf{T} \geq i, u_1 = \mathbf{u_1}, u_2 = \mathbf{u_2}, \ldots, u_{i-1} = \mathbf{u_{i-1}}]$.

Proof. We observe that during each deletion or insertion operation, the algorithm performs at most one level construction. Suppose the level construction is triggered at level x. For Property 2 and Property 3, if they hold for all levels $1 \leq i < x$ before the update, they continue to hold afterwards as those levels remain unchanged. For levels $x \leq i \leq T$, Property 2 is guaranteed by the lemma 3, and Property 3 holds because the execution of line $5-6$ of MATROIDCONSTRUCTLEVEL.

We remark that the proof of Property 4 is essentially identical to that in (Banihashem et al. 2024), and we omit it for brevity. This concludes the proof.

□

Algorithm 8: STREAMALG$(\mathcal{M}(\mathcal{N}, \mathcal{I}), \pi, \tau)$

```
a ← [0,1]^N.
A_j ← ∅ for j ∈ {lb(τ), lb(τ)+1, ..., ub(τ)}.
for every arriving element u ∈ π do
    if PICK(a, {A_j}, u) ≠ False then
        {a, {A_j}} ← PICK(a, {A_j}, u).
return PACK{A_j}.
```

Approximation Guarantee. We can use the properties of the above data structure to establish the approximation ratio of the algorithm.

We first state the approximation guarantee of the STREAMALG algorithm. Here, we point out that this algorithm is a modified version of that in Feldman et al. (2022). We modify it by avoiding using the sliding window technique to ensure it can be more effectively maintained by our dynamic data structure. Our proof follows a similar approach to that of Feldman et al. (2022) so we defer it to the appendix; here we only present the main theorems.

Theorem 6. *For a submodular function* $f : 2^{\mathcal{N}} \to \mathbb{R}$ *and an arbitrary data stream* π*, if we are given in advance a value* $\tau \in [f(\mathrm{OPT}), (1+\varepsilon)f(\mathrm{OPT})]$*, then Algorithm 8 returns a set* S *with* $f(S) \geq (\frac{1-e^{-\alpha}}{1+\alpha} - O(\varepsilon)) \cdot f(OPT) \approx 0.3178 - O(\varepsilon) \cdot f(OPT)$.

Next, we show that after each update, the solution maintained by our data structure preserves the same approximation guarantee. In fact, the data structure is designed to faithfully simulate the execution of STREAMALG on a specific input stream.

Theorem 7. *After each* INSERT *or* DELETE *operation, our data structure simulates* STREAMALG *with data stream* $\pi = (R_0 \setminus R_1, u_1, R_1 \setminus R_2, u_2, , \ldots, R_{T-1}, u_T)$*, where where each set* $R_t \setminus R_{t+1}$ *appears as a contiguous block, and the elements within each block can be arranged in an arbitrary order. Hence, our algorithm returns a solution* S *with* $f(S) \geq (0.3178 - O(\varepsilon)) \cdot f(OPT)$ *after each update.*

Proof. According to Properties 2 and 3 of Theorem 5, we consider the execution of STREAMALG on the stream π. The algorithm first picks u_1 and updates accordingly. Then, Property 2 implies that after u_1 is picked, the elements will no longer be picked for all elements in $R_0 \setminus R_1$. Hence we can just skip them without leading to any effective update. The algorithm then picks u_2 and skips all elements in

$R_1 \setminus R_2$, accepts u_3, and so on, proceeding through the entire stream. After doing so, it produces the solution vector $a_T, \{A_{i,T}\}$. Finally, the algorithm passes this information to PACK, which returns a set S, just as it does in the INSERT and DELETE procedures.

□

Query Complexity. We first note that our algorithm uses an estimated value of F via sampling. Nevertheless, we can show that compared to using the exact value of F, this only incurs an additional overhead of $\tilde{O}_\varepsilon(k^2)$ and results in at most an ε loss in the approximation ratio. This allows us to incur an additional $\tilde{O}_\varepsilon(k^2)$ factor in the query complexity when converting between queries to f and F.

Lemma 8. *If we use Monte Carlo sampling to estimate the value of F instead of accessing its exact value in our dynamic algorithm, then using $O\left(\varepsilon^{-2}k^2\log^2(k/\varepsilon)\log k\right)$ value oracle of f per evaluation suffices to return a solution S whose value $f(S)$ is within an additive $\varepsilon f(\mathrm{OPT})$ error, with failure probability $o(1)$.*

We also state the queries required by PIPAGEROUNDING algorithm.

Theorem 9 (Cualinescu et al. (2011))**.** *The* PIPAGEROUNDING *algorithm takes as input a vector $x \in [0,1]^{\mathcal{N}}$. It uses $\mathrm{supp}^2(x)$ membership oracle queries and does not require any value oracle queries, where $\mathrm{supp}(x)$ denotes the number of nonzero components in x.*

Observe that in our algorithm, the support of any vector x used has size at most $O(k\log(k/\varepsilon))$. Therefore, PIPAGEROUNDING does not constitute the main part of the query complexity of our algorithm.

Then we analyze the algorithm MATROIDCONSTRUCTLEVEL, and further examine the amortized query complexity of INSERT and DELETE.

Lemma 10. MATROIDCONSTRUCTLEVEL(i,τ) *requires $O\left(|R_i|\varepsilon^{-2}k^2\log^3(k/\varepsilon)\log^2 k\right)$ orcale queries.*

Proof. All oracle queries in the execution of MATROIDCONSTRUCTLEVEL are made within calls to PICK. We first count the number of calls to PICK. Each element in every set R_i is invoked in a single call to PICK at Line 4 of the algorithm. Additionally, for each element that returns **False** by PICK , the algorithm performs a binary search (line 8), which incurs at most $O(\log T) = O(\log(k/\varepsilon))$ further calls to PICK per element. Thus We totally need $O(|R_i|\log(k/\varepsilon))$ calls of PICK.

Each PICK operation queries the value of the multilinear extension once, and queries the independence of at most $O(\log(k/\varepsilon))$ sets, so each PICK requires at most $O\left(\varepsilon^{-2}k^2\log^2(k/\varepsilon)\log k\right)$ oracle queries, where the value oracle dominates. Thus, the execution of MATROIDCONSTRUCTLEVEL uses

$$O\left(|R_i|\varepsilon^{-2}k^2\log^3(k/\varepsilon)\log^2 k\right)$$

oracle queries in total.

□

Lemma 11. *Both* INSERT(u,τ) *and* DELETE(u,τ) *require $O\left(\varepsilon^{-2}k^3\log^4(k/\varepsilon)\log^2 k\right)$ amoritzed oracle queries.*

Proof. Due to Property 4 of Theorem 5, we know that each update calls MATROIDCONSTRUCTLEVEL(i,τ) with probablity $\frac{1}{|R_i|}$. By linearity of expectation, the total expected number of oracle queries by MATROIDCONSTRUCTLEVEL is:

$$\sum_{i=1}^{T}\frac{1}{|R_i|}O\left(|R_i|\varepsilon^{-2}k^2\log^3(k/\varepsilon)\log^2 k\right)$$
$$= O\left(\varepsilon^{-2}k^3\log^4(k/\varepsilon)\log^2 k\right).$$

Each update also invokes the PACK algorithm, which does not make any value oracle queries. It only performs $O(k\log(k/\varepsilon)) + O(k^2\log^2(k/\varepsilon)) = O(k^2\log^2(k/\varepsilon))$ membership oracle queries, which are negligible in terms of the overall query complexity, which complete the proof. □

Finally, since the technique for removing the assumption on τ incurs only an additional $\log(k/\varepsilon)$ factor, the final theorem follows immediately.

Theorem 12. DYNAMICMATROID *algorithm requires at most $O\left(\varepsilon^{-2}k^3\log^5(k/\varepsilon)\log^2 k\right)$ amortized oracle queries.*

Dynamic Algorithm for Non-monotone Submodular Functions

The approach for the non-monotone case is largely similar; we essentially replace the original PICK and PACK procedures with their non-monotone counterparts, PICKNM and PACKNM, both adapted from Feldman et al. (2022). Notably, we identified a minor error in the proof of the original PACK algorithm and fixed it through a modification of the algorithm. Due to space limitations, we defer the algorithm DYNAMICMATROIDNM and its analysis to the appendix.

Theorem 13. *For a submodular function f and a fully dynamic stream with insertion and deletion,* DYNAMICMATROIDNM *guarantees a $(0.1921 - O(\varepsilon))$-approximation and have amortized query complexity $O(\varepsilon^{-2}k^3\log(k)\log^3(k/\varepsilon))$.*

Conclusion

In recent years, the dynamic model has attracted considerable attention in submodular maximization, leading to a number of intriguing advances. In this work, we significantly improve the approximation ratios for both the monotone and non-monotone settings of fully dynamic submodular maximization under matroid constraints. Our algorithm is independent of the ground-set size n, though its dependence on the rank k remains relatively high. This, however, is unavoidable, since even approximating the multilinear extension requires $\tilde{O}(k^2)$ time. A major open question also remains: whether one can design algorithms with $\mathrm{poly}(\log n, \log k)$ update complexity under matroid constraints, analogous to what is achievable under cardinality constraints.

Acknowledgments

This work was supported in part by the National Natural Science Foundation of China Grants No. 62272441, 62402110.

References

Bach, F. R. 2010. 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*, 118–126. Curran Associates, Inc.

Bairi, R.; Iyer, R. K.; Ramakrishnan, G.; and Bilmes, J. A. 2015. 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*, 553–563. The Association for Computer Linguistics.

Banihashem, K.; Biabani, L.; Goudarzi, S.; Hajiaghayi, M.; Jabbarzade, P.; and Monemizadeh, M. 2023a. Dynamic Constrained Submodular Optimization with Polylogarithmic Update Time. In *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, volume 202 of *Proceedings of Machine Learning Research*, 1660–1691. PMLR.

Banihashem, K.; Biabani, L.; Goudarzi, S.; Hajiaghayi, M.; Jabbarzade, P.; and Monemizadeh, M. 2023b. Dynamic Non-monotone Submodular Maximization. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.

Banihashem, K.; Biabani, L.; Goudarzi, S.; Hajiaghayi, M.; Jabbarzade, P.; and Monemizadeh, M. 2024. Dynamic Algorithms for Matroid Submodular Maximization. In *Proceedings of the 2024 ACM-SIAM Symposium on Discrete Algorithms, SODA 2024, Alexandria, VA, USA, January 7-10, 2024*, 3485–3533. SIAM.

Buchbinder, N.; and Feldman, M. 2024. Constrained Submodular Maximization via New Bounds for DR-Submodular Functions. In *Proceedings of the 56th Annual ACM Symposium on Theory of Computing, STOC 2024, Vancouver, BC, Canada, June 24-28, 2024*, 1820–1831. ACM.

Chen, X.; and Peng, B. 2022. On the complexity of dynamic submodular maximization. In *STOC '22: 54th Annual ACM SIGACT Symposium on Theory of Computing, Rome, Italy, June 20 - 24, 2022*, 1685–1698. ACM.

Cualinescu, G.; Chekuri, C.; Pál, M.; and Vondrák, J. 2011. Maximizing a Monotone Submodular Function Subject to a Matroid Constraint. *SIAM J. Comput.*, 40(6): 1740–1766.

Das, A.; and Kempe, D. 2011. 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*, 1057–1064. Omnipress.

Das, A.; and Kempe, D. 2018. Approximate Submodularity and its Applications: Subset Selection, Sparse Approximation and Dictionary Selection. *J. Mach. Learn. Res.*, 19: 3:1–3:34.

Duetting, P.; Fusco, F.; Lattanzi, S.; Norouzi-Fard, A.; and Zadimoghaddam, M. 2023. Fully Dynamic Submodular Maximization over Matroids. In *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, volume 202 of *Proceedings of Machine Learning Research*, 8821–8835. PMLR.

Feldman, M.; Liu, P.; Norouzi-Fard, A.; Svensson, O.; and Zenklusen, R. 2022. Streaming Submodular Maximization Under Matroid Constraints. In *49th International Colloquium on Automata, Languages, and Programming, ICALP 2022, July 4-8, 2022, Paris, France*, volume 229 of *LIPIcs*, 59:1–59:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

Gharan, S. O.; and Vondrák, J. 2011. Submodular Maximization by Simulated Annealing. In *Proceedings of the Twenty-Second Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2011, San Francisco, California, USA, January 23-25, 2011*, 1098–1116. SIAM.

Kempe, D.; Kleinberg, J. M.; and Tardos, É. 2003. Maximizing the spread of influence through a social network. In *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Washington, DC, USA, August 24 - 27, 2003*, 137–146. ACM.

Khanna, R.; Elenberg, E. R.; Dimakis, A. G.; Negahban, S. N.; and Ghosh, J. 2017. Scalable Greedy Feature Selection via Weak Submodularity. In *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics, AISTATS 2017, 20-22 April 2017, Fort Lauderdale, FL, USA*, volume 54 of *Proceedings of Machine Learning Research*, 1560–1568. PMLR.

Kumari, L.; and Bilmes, J. A. 2021. Submodular Span, with Applications to Conditional Data Summarization. In *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Thirty-Third Conference on Innovative Applications of Artificial Intelligence, IAAI 2021, The Eleventh Symposium on Educational Advances in Artificial Intelligence, EAAI 2021, Virtual Event, February 2-9, 2021*, 12344–12352. AAAI Press.

Lattanzi, S.; Mitrovic, S.; Norouzi-Fard, A.; Tarnawski, J.; and Zadimoghaddam, M. 2020a. Fully Dynamic Algorithm for Constrained Submodular Optimization. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*.

Lattanzi, S.; Mitrovic, S.; Norouzi-Fard, A.; Tarnawski, J.; and Zadimoghaddam, M. 2020b. Fully Dynamic Algorithm for Constrained Submodular Optimization. *CoRR*, abs/2006.04704.

Liu, Y.; and Yang, W. 2024. Dynamic Algorithms for Non-monotone Submodular Maximization. In *Combinatorial*

Optimization and Applications - 17th International Conference, COCOA 2024, Beijing, China, December 6-8, 2024, Proceedings, Part II, volume 15435 of *Lecture Notes in Computer Science*, 119–131. Springer.

Monemizadeh, M. 2020. Dynamic Submodular Maximization. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*.

Nemhauser, G. L.; and Wolsey, L. A. 1978. Best Algorithms for Approximating the Maximum of a Submodular Set Function. *Math. Oper. Res.*, 3(3): 177–188.

Qi, B. 2022. On Maximizing Sums of Non-Monotone Submodular and Linear Functions. In *33rd International Symposium on Algorithms and Computation, ISAAC 2022, December 19-21, 2022, Seoul, Korea*, volume 248 of *LIPIcs*, 41:1–41:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

Radanovic, G.; Singla, A.; Krause, A.; and Faltings, B. 2018. Information Gathering With Peers: Submodular Optimization With Peer-Prediction Constraints. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*, 1603–1610. AAAI Press.

Vondrák, J. 2008. Optimal approximation for the submodular welfare problem in the value oracle model. In *Proceedings of the 40th Annual ACM Symposium on Theory of Computing, Victoria, British Columbia, Canada, May 17-20, 2008*, 67–74. ACM.

Zheng, J.; Jiang, Z.; Chellappa, R.; and Phillips, P. J. 2014. 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*, 1341–1349.