Faster Submodular Maximization for Several Classes of Matroids

Monika Henzinger
Institute of Science and Technology Austria (ISTA), Klosterneuburg, Austria

Paul Liu
Stanford University, CA, USA

Jan Vondrák
Stanford University, CA, USA

Da Wei Zheng
University of Illinois Urbana-Champaign, IL, USA

Abstract

The maximization of submodular functions have found widespread application in areas such as machine learning, combinatorial optimization, and economics, where practitioners often wish to enforce various constraints; the matroid constraint has been investigated extensively due to its algorithmic properties and expressive power. Though tight approximation algorithms for general matroid constraints exist in theory, the running times of such algorithms typically scale quadratically, and are not practical for truly large scale settings. Recent progress has focused on fast algorithms for important classes of matroids given in explicit form. Currently, nearly-linear time algorithms only exist for graphic and partition matroids [12]. In this work, we develop algorithms for monotone submodular maximization constrained by graphic, transversal matroids, or laminar matroids in time near-linear in the size of their representation. Our algorithms achieve an optimal approximation of $1 - 1/e - \varepsilon$ and both generalize and accelerate the results of Ene and Nguyen [12]. In fact, the running time of our algorithm cannot be improved within the fast continuous greedy framework of Badanidiyuru and Vondrák [6].

To achieve near-linear running time, we make use of dynamic data structures that maintain bases with approximate maximum cardinality and weight under certain element updates. These data structures need to support a weight decrease operation and a novel Freeze operation that allows the algorithm to freeze elements (i.e. force to be contained) in its basis regardless of future data structure operations. For the laminar matroid, we present a new dynamic data structure using the top tree interface of Alstrup, Holm, de Lichtenberg, and Thorup [2] that maintains the maximum weight basis under insertions and deletions of elements in $O(\log n)$ time. This data structure needs to support certain subtree query and path update operations that are performed every insertion and deletion that are non-trivial to handle in conjunction. For the transversal matroid the Freeze operation corresponds to requiring the data structure to keep a certain set S of vertices matched, a property that we call S-stability. While there is a large body of work on dynamic matching algorithms, none are S-stable *and* maintain an approximate maximum weight matching under vertex updates. We give the first such algorithm for bipartite graphs with total running time linear (up to log factors) in the number of edges.

2012 ACM Subject Classification Theory of computation → Data structures design and analysis; Mathematics of computing → Submodular optimization and polymatroids; Theory of computation → Dynamic graph algorithms

Keywords and phrases submodular optimization, dynamic data structures, matching algorithms

Digital Object Identifier 10.4230/LIPIcs.ICALP.2023.74

Category Track A: Algorithms, Complexity and Games

Related Version *Full Version*: `https://arxiv.org/abs/2305.00122`

Funding *Monika Henzinger*: This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (Grant agreement No. 101019564 "The Design of Modern Fully Dynamic Data Structures (MoDynStruct)" and from the Austrian Science Fund (FWF) project "Static and Dynamic Hierarchical Graph Decompositions", I 5982-N, and project "Fast Algorithms for a Reactive Network Layer (ReactNet)", P 33775-N, with additional funding from the *netidee SCIENCE Stiftung*, 2020–2024.
Jan Vondrák: Supported by NSF Award 2127781.

1 Introduction

Submodular optimization is encountered in a variety of applications – combinatorial optimization, information retrieval, and machine learning, to name a few [7]. Many such applications involve constraints, which are often in the form of cardinality or weight constraints on certain subsets of elements, or combinatorial constraints such as connectivity or matching. A convenient abstraction which has been studied heavily in this context is that of a *matroid constraint*[1]. For instance, transversal matroids appear in ad placement and matching applications [4, 3], laminar and partition matroids capture capacity constraints on subsets which are widely used in recommendation settings (e.g. YouTube video recommendation algorithm [25]), and graphic matroids appear in applications for approximating Metric TSP [26]. Maximization of a submodular function subject to any of these constraints is an APX-hard problem, but a $(1-1/e)$-approximation is known in this setting for any matroid constraint [11], and the factor of $1-1/e$ is also known to be optimal [22, 13]. Considering this, it has been a long-standing quest to develop fast algorithms for the submodular maximization problem that achieve an approximation close to the optimal factor of $1-1/e$. In this work, we achieve this goal for several common classes of matroids.

Perhaps the first step in this direction was the *threshold-greedy technique* which gives a fast $(1/2-\varepsilon)$-approximation [5] for the cardinality constraint. With more work, this technique can be extended to give approximations close to $1-1/e$ [6] and ultimately a $(1-1/e-\varepsilon)$-approximation in running time $O(n/\varepsilon)$ was found for the cardinality constraint.[2]

For general matroids, the fastest known algorithm is the *fast continuous greedy algorithm*, which uses $O(nr\varepsilon^{-4}\log^2(n/\varepsilon))$ oracle, where n is the number of elements in the matroid and r is the rank of the matroid [6]. (The exact running time would depend on the implementation of these queries, which [6] does not address.) We can assume that the rank r scales polynomially with n and hence this algorithm is not near-linear. Further work on fast submodular optimization developed in the direction of parallelized and distributed settings (see [18, 20, 21] and the references therein); we do not discuss these directions in this paper.

A recent line of work initiated by Ene and Nguyen [12] attempts to develop a "nearly-linear" continuous greedy algorithm, i.e., with a running time of $n \cdot \mathrm{poly}(1/\varepsilon, \log n)$. They achieved this goal for partition and graphic matroids. Prior to their work, the fastest known algorithm for any matroid class beyond cardinality was the work of Buchbinder, Feldman, and Schwartz [10], who showed an $O(n^{3/2})$-time algorithm for partition matroids.

This immediately leads to the question of whether such improvements are also possible for other classes of matroid constraints. As observed by Ene and Nguyen [12], even determining feasibility may take longer than linear time for certain matroids. One such example is for

[1] A matroid on N is a family of "independent sets" $\mathcal{I} \subset 2^N$ which is down-closed, and satisfies the extension axiom: For any $A, B \in \mathcal{I}$, if $|A| < |B|$ then there is an element $e \in B \setminus A$ such that $A \cup \{e\} \in \mathcal{I}$.
[2] Running time in this paper includes value queries to the objective function $f(S)$ as unit-time operations.

matroids represented by linear systems, as simply checking the independence of a linear system takes $O(\text{rank}(\mathcal{M})^\omega)$ time, where $\text{rank}(\mathcal{M})$ is the rank of the linear system and ω is the exponent for fast matrix multiplication.

1.1 Our contributions

In this paper, we generalize and significantly improve the work of Ene and Nguyen [12], to develop a $(1-1/e-\varepsilon)$-approximation for maximizing a monotone submodular function subject to a matroid constraint, for several important classes of matroids: namely for graphic, laminar, and transversal matroids. The technical developments behind these results are on two fronts:
(i) a refinement of the optimization framework of [12] and formulation of abstract data structures required for this framework;
(ii) implementation of such data structures for graphic, laminar and transversal matroids. (See Section 2 for definitions of these classes of matroids.)

To describe our results in more detail, the efficiency of optimizing a submodular function f can be broken down into two components: the number of oracle calls to f, and the number of additional arithmetic operations needed to support the algorithm optimizing f. The number of oracle calls to f that our framework need to achieve a $(1-1/e-\varepsilon)$-approximation is $O_\varepsilon(n\log^2(n))$ regardless of the matroid, where n is the number of elements in the matroid constraint. Thus for all results below, the running time is measured in the number of the number of arithmetic operations needed by the data structures supporting the optimization of f. Our contributions are as follows:

- We give nearly-linear time versions of continuous greedy for laminar, graphic, and transversal matroids. These algorithms are accelerated by special data structures we developed for each matroid and which might be of independent interest. For all of our matroids, it is impossible to improve our running time without improving the continuous greedy algorithm itself.
- For graphic matroids on n vertices and m edges, we improve the running time of [12] from $O_\varepsilon(n\log^5 n + m\log^2 n)$ to $O_\varepsilon(m\log^2 n)$.
- We generalize the partition matroids results of [12] to laminar matroids, and match their running time of $O_\varepsilon(n\log^2 n)$ for continuous greedy. As a by-product, we also develop the first data structure that maintains the maximum weight basis on a laminar matroid with $O(\log n)$ update time for insertions and deletions that may be of independent interest. This data structure uses the top tree interface of Alstrup, Holm, de Lichtenberg, and Thorup [2].
- For transversal matroids represented by a bipartite graph with m edges and the ground set being one side of the partition with n vertices, we give an algorithm running in $O_\varepsilon(m\log n + n\log^2 n)$ time.[3] This is the first such fast algorithm for transversal matroids. For this we develop a dynamic matching algorithm in a vertex-weighted, bipartite graph with a weighted vertex sets L and an unweighted vertex set R with the following conditions: (i) There exists a dynamically changing set $S \subseteq L$ such that every vertex in S must be matched in the current matching. (ii) The matching must give both an approximation in terms of cardinality in comparison to the maximum cardinality matching *as well as* the weight of the matching in comparison to the best matching that matches all vertices of S, where the weight of the matching is the sum of the weights of the matched vertices of L.

[3] Any transversal matroid with n elements can be represented as the family of matchable sets the left-hand side of an $n \times n^2$ bipartite graph, with degrees at most n. See Section 2 for more details.

We emphasize that our results are true running times, as opposed to black-box independence queries to the matroid. The only black box operation we need is the query to the objective function $f(S)$.

The performance of the dynamic matching data structure used in our transversal matroid algorithm is interesting as none of the earlier work on dynamic matching can maintain both a constant approximation to the weight as well as to the cardinality of the matching. Our algorithm builds on a recent fast algorithm for maximum-weight matching by Zheng and Henzinger [27]. We briefly mention a few relevant works:

- There is a conditional lower bound based on the OMv conjecture [16] that shows that maintaining a *maximum weight matching* in an edge weighted bipartite graph with only *edge weight increase operations* cannot be done in amortized time $O(n^{1-\delta})$ per edge weight increase and amortized time $O(n^{2-\delta})$ per query operation for any $\delta > 0$ [17]. The reduction from [17] can be adapted to the setting with only *edge weight decrement* operations, achieving the same lower bound. Thus, this shows that our running time bound cannot achieved if a *exact* maximum weight matching has to be maintained under edge weight decrement operations.
- Le, Milenkovic, Solomon, and Vassilevska-Williams [19] studied one-sided vertex updates (insertions and deletions) in bipartite graphs and gave a *maximal matching* algorithm whose total running time is $O(K)$, where K is the total number of inserted and deleted *edges.* Bosek el al. [8] studied one-sided vertex updates (either insertions-only or deletions-only) in bipartite graphs and gave algorithms that for any $\varepsilon > 0$ maintain a $(1-\varepsilon)$-approximate maximum cardinality matching in total time $O(m/\varepsilon)$. Gupta and Peng [15] developed the best known dynamic algorithm that allows both *edge* insertions and deletions and maintains a $(1-\varepsilon)$-approximate maximum cardinality matching (for any $\varepsilon > 0$). It requires time $O(\sqrt{m}/\varepsilon)$ amortized time per operation. For all these algorithms, they either cannot be extended to the weighted setting, or cannot maintain both a constant approximation to the weight as well as to the cardinality of the matching.

1.2 Technical Overview

The submodular optimization framework. Our framework is an adaptation and improvement over that of Ene and Nguyen [12]. The framework consists of two phases:

(1) a LazySamplingGreedy phase, which aims to build a partial solution that either provides a good approximation on its own, or reduces the problem to a residual instance with bounded marginal values of elements.

(2) a ContinuousGreedy phase, which is essentially the original fast continuous greedy algorithm [11, 6], with an improved analysis based on the fact the marginal values are bounded.

The original fast ContinuousGreedy runs in time $O_\varepsilon(nr\log^2 n)$, where the factor of $r\log^2 n$ is due to the cost in evaluating the multilinear extension of f. The multilinear extension is an average over values of f on randomly drawn subsets of the input. In ContinuousGreedy, $O(r\log^2 n)$ samples are needed to estimate the multilinear extension to sufficient accuracy.

The LazySamplingGreedy phase transforms f into a function $\tilde{f}$ such that (a) the number of samples required in ContinuousGreedy for $\tilde{f}$ is reduced by a factor of r and (b) the optimal solution of $\tilde{f}$ is within a $(1-\varepsilon)$ factor of the optimal solution f. LazySamplingGreedy does this by constructing an initial independent set S in our matroid $\mathcal{M}$ such that $\tilde{f}(T) = f(T \cup S)$ has relatively small marginal values. The final solution is obtained by running ContinuousGreedy on $\tilde{f}$ constrained by $\mathcal{M} \setminus S$ and combining the solution with S. The construction of S relies on a fast data structure to get the maximum weight independent set of $\mathcal{M}$ at any given time, subject to weight changes on each element.

We begin by simplifying and improving the LazySamplingGreedy phase (Section 3.2). A significant part of LazySamplingGreedy in [12] is dedicated to randomly checking and refreshing estimates of marginal values for each element in the matroid. We show that (a) this random checking can be dramatically reduced, and (b) the maximum-weight independent set requirement for the data structure can be relaxed to a constant-factor approximation of the maximum-weight independent set. The relaxation to constant-factor approximations enables us to design much more efficient data structures than the previous work of [12], and also allows us to handle more classes of matroids.

In the ContinuousGreedy phase (Section 3.3), a subroutine requires an independence oracle to check if proposed candidate solutions are independent sets of the matroid. Ene and Nguyen use fully dynamic independence oracles to implement this, where independence queries require $O(\text{polylog } n)$ time. We show that only incremental independence oracles are needed. This opens the door to implementing such oracles for new classes of matroids, as sublinear fully dynamic independence oracles are provably hard in many settings (e.g. bipartite maximum-cardinality matchings are hard to maintain exactly in faster than $O(n^{2-\varepsilon})$ amortized time per update [1], which corresponds to finding the maximum independent set in a transversal matroid).

Dynamic data structures for various matroid constraints. From a data structures point of view, we give the first efficient data structure for two settings. (We refer the reader to Section 2 for definitions of laminar, graphic and transversal matroids.)

Maximum-weight basis in a laminar matroid. In a laminar matroid with n elements we are able to output the maximum-weight basis under an online sequence of insertions and deletions of weighted elements with $O(\log n)$ time per update, which we show to be worst-case optimal. The biggest challenge for laminar matroids is that each element may have as many as $O(n)$ constraints that need to be kept track of on each insertion and deletion. We leverage the tree structure induced by a laminar set system to build data structures. Specifically, we show there exists a data structure with $O(\log^2 n)$ query time using the heavy-light decomposition technique of [24]. Since the heavy-light decomposition technique decomposes the tree into paths, we store some carefully chosen auxiliary information at each vertex to transform our subtree queries into path queries. We further improve this to $O(\log n)$ using the top tree interface of Alstrup, Holm, de Lichtenberg, and Thorup [2] that more naturally support both path and subtree operations using a similar idea of carefully storing the right auxiliary information at each top tree node combined with lazily propagating path changes.

Approximate maximum-weight basis in a transversal matroid. In the case of transversal matroids, our LazySamplingGreedy+ algorithm requires what we call a (c, d)-*approximate maximum weight matching oracle*: a matching that is an c-approximation in weight and at least d of the cardinality of the maximum cardinality matching. In addition, the oracle must implement two update operations, (a) a Freeze operation that adds a vertex of L to S and (b) a Decrement operation that reduces the weight of a vertex of L to a given value. We give a novel algorithm that maintains a maximal (inclusion-wise) matching M in a weighted graph such that the weight of the matching is a $(1-\varepsilon)$-approximation of the max-weight solution in total time $O(m(1/\varepsilon + \log n))$. Thus our algorithm is a $(1-\varepsilon, 1/2)$-approximate maximum weight matching oracle. Due to standard rescaling techniques we can assume that the maximum weight $W = O(n)$.

To illustrate the challenges introduced by FREEZE operations, assume we want to maintain a $(1/2, 1/2)$-approximate maximum weight matching oracle and let $n \geq 5$ be an odd integer. Consider a graph consisting of the path $\ell_0, r_0, \ell_1, r_2, \ldots, \ell_{(n+1)/2}$ of length $n-1$ where the first and last edge have large weight $W<$ say $W = 100n$, and all other edges have weight 1. If the initial $(\Omega(1), \Omega(1))$-approximate matching has greedily picked every second edge, it achieves a weight of $W + \frac{n-1}{2} - 1$ versus an optimum weight of $2W + \frac{n-1}{2} - 2$. Now if the weight of the first edge is halved, the weight of the computed solution drops to $W/2 + \frac{n-1}{2} - 1$, which is no longer a 2-approximation of the weight of the optimum solution (for large enough W). Thus the algorithm needs to match the last edge in order to maintain a 2-approximation to the weight. This would only require two changes to the matching, namely un-matching the edge $(\ell_{(n-1)/2}, r_{(n-1)/2})$ and matching the last edge. However, if prior FREEZE operations added the vertices $\ell_1, \ell_2, ...\ell_{(n-1)/2}$ to S, i.e. $S = V \setminus \{\ell_0\}$, then we cannot un-match $\ell_{(n-1)/2}$. Thus, the edge $(\ell_{(n-1)/2}, r_{(n-3)/2})$ needs to be unmatched, which in turn un-matches the vertex $\ell_{(n-3)/2}$, leading to further un-matchings and matchings along the path. More specifically, all $(n-1)/2$ matched edges need to change. Next, the weight of the last edge is divided by 4 and the process along the path starts again. Thus, due to the prior FREEZE operations, each DECREMENT operation can lead to $\Theta(n) = \Theta(m)$ changes in the graph. As this can be repeated $\Theta(\log W) = \Theta(\log n)$ times it shows that work $\Omega(m \log n)$ is unavoidable if a 2-approximation to the weight is maintained with FREEZE operations. This is the running time that we achieve.

More precisely, for any $\varepsilon > 0$, we give an $(1-\varepsilon, 1/2)$-approximate maximum weight matching oracle under FREEZE and DECREMENT operations with total time $O(m(\log n + 1/\varepsilon))$. It follows that $O(m(\log n + 1/\varepsilon))$ is also an upper bound on the number of matching and un-matching operations of the algorithm. To do so we extend a recent algorithm [27] for fast $(1-\varepsilon)$-approximate maximum weight matching algorithm in bipartite graphs based on the multiplicative weight update idea. The analysis within [27] shows stability properties that makes the FREEZE operation trivial to implement. However, LAZYSAMPLINGGREEDY+ requires that the maintained matching is at least 1/2 the size of the maximum matching. Furthermore, we need to support the DECREMENT operation. We show that both extensions are possible and obtain an algorithm with total time (for all operations) of $O(m(1/\varepsilon + \log n))$.

2 Preliminaries

Set notation shorthands. Given a set $S \subseteq \mathcal{N}$ and an element $u \in \mathcal{N}$, we denote $S \cup \{u\}$ and $S \setminus \{u\}$ by $S + u$ and $S - u$ respectively. Similarly, given sets $S, T \subseteq \mathcal{N}$, we denote $S \cup T$ and $S \setminus T$ by $S + T$ and $S - T$ respectively.

Submodular functions. Given a set $\mathcal{N}$, a set function $f \colon 2^{\mathcal{N}} \to \mathbb{R}$ is called *submodular* if for any two sets S and T we have

$$f(S) + f(T) \geq f(S + T) + f(S - T).$$

We only consider monotone submodular functions, where $f(S) \leq f(T)$ for any sets $S \subseteq T$.

Matroids. A set system is a pair $\mathcal{M} = (\mathcal{N}, \mathcal{I})$, where $\mathcal{I} \subseteq 2^{\mathcal{N}}$. We say that a set $S \subseteq \mathcal{N}$ is *independent* in $\mathcal{M}$ if $S \in \mathcal{I}$. The *rank* of the set system M is defined as the maximum size of an independent set in it. The independent sets must satisfy (i) $S \subseteq T, T \in \mathcal{I} \implies S \in \mathcal{T}$, and (ii) $S, T \in I, |S| < |T| \implies \exists e \in T \setminus S$ such that $S + e \in \mathcal{I}$.

A matroid constraint means that f is optimized over the independent sets of a matroid.

Graphic matroids. Let $G = (V, E)$ be a graph. A graphic matroid has $\mathcal{N} = E$ and $\mathcal{I}$ equal to the forests of G. The rank of $\mathcal{M}$ is $|V| - C$ where C is the number of connected components in G.

Laminar matroids. Let $\{S_1, S_2, \ldots, S_m\}$ be a collection of subsets of a set $\mathcal{N}$ such that for any two intersecting sets S_i and S_j where $i \neq j$, either $S_i \subset S_j$ or $S_j \subset S_i$. Furthermore, let there be non-negative integers $\{c_1, c_2, \ldots, c_m\}$ associated with the S_i's. Let $\mathcal{I}$ be the sets $S \subseteq \mathcal{N}$ for which $|S \cap S_i| \leq c_i$ for all i. $\mathcal{I}$ is then the collection of independent sets in a laminar matroid on $\mathcal{N}$. A laminar matroid has a natural representation as a tree on m nodes, which we describe in the full version.

Transversal matroids. Let $G = ((L, R), E)$ be a bipartite graph with a bipartition of vertices (L, R). Let $\mathcal{I}$ be the collection of sets $S \subseteq L$ such that there is a matching of all vertices in S to $|S|$ vertices in R. A transversal matroid has $\mathcal{N} = L$ and $\mathcal{I}$ as its independent sets. From the definition, it is clear that $\text{rank}(\mathcal{M})$ is the size of the maximum cardinality matching in G. It is known that every transveral matroid can be represented by a bipartite graph where L is the ground set of the matroid and $|R| = \text{rank}(\mathcal{M})$ (see [23], Volume B, equation (39.18)).

3 Improved nearly-linear submodular maximization

In what follows, f is the submodular function we want to maximize, $\mathcal{M}$ is the matroid constraint, n is the number of elements in $\mathcal{M}$, and OPT is the optimal independent set. Additionally, we can assume that $\text{rank}(\mathcal{M}) = \omega(\log n)$, as the standard ContinuousGreedy would run in $O(n\,\text{polylog} n)$ time otherwise.

Our high-level framework adapts and improves upon the nearly-linear time framework of Ene and Nguyen [12]. We will do the following:

Algorithm 3.1: Overall Framework.

1. Run LazySamplingGreedy+ (see below), to obtain a partial solution S_0.
2. Run ContinuousGreedy on $\tilde{f}(T) = f(S_0 \cup T) - f(S_0)$ with the constraint $\mathcal{M}/S_0$, to obtain a solution S_1.
3. Return $S_0 \cup S_1$.

As previously discussed, the original ContinuousGreedy runs in time $O_\varepsilon(nr\log^2 n)$, where the $r\log^2 n$ is due to the number of samples needed to evaluate the multilinear extension of f. LazySamplingGreedy+ finds a set S_0 such that $\tilde{f}(T) := f(T \mid S_0) = f(S_0 \cup T) - f(S_0)$ has a tighter range of marginal values. This allows us to reduce the number of samples used in ContinuousGreedy by a factor of r.

The overall idea is to run LazySamplingGreedy+ until the marginals in $\tilde{f}$ are small enough to guarantee good performance in the ContinuousGreedy phase of our overall framework (Algorithm 3.1). To accelerate our algorithms, we construct specialized data structures for both LazySamplingGreedy+ and ContinuousGreedy.

3.1 Data structure requirements

We next describe the data structures needed for the two phases of our algorithm. In the LazySamplingGreedy+ phase we need a c-approximate dynamic max-weight independent set oracle. In the CountinuousGreedy phase we have two options of dynamic independence oracles, both of them unweighted. In addition, our LazySamplingGreedy+ requires the ability to obtain a weighted sample from the approximate max-weight independent set. Since we use these data structures as subroutines in our static algorithm which uses the answers of the data structure to determine future updates, it is important that their running time bounds are valid against an *adaptive* adversary.

Dynamic (c, d)-approximate maximum weight oracle. Let $\mathcal{M} = (E, \mathcal{I})$ be a matroid. Given an independent set $S \subseteq E$ the *independent sets relative to* S are the independent sets of $\mathcal{M}$ that contain S. Let $rank(\mathcal{M})$ denote the size of the largest independent set, which equals the size of the largest independent set containing S. The weight of an independent set is the sum of the weights of its elements. A *maximum weight basis in* $\mathcal{M}$ *relative to* S is a basis B^* that maximizes the sum of w_e over all bases of $\mathcal{M}$ that contain S.

Let $c < 1$ and $d < 1$ be constants. An independent set B is called an (c, d)*-approximate independent set relative to* S if it fulfills the following conditions: (a) its size is at least $\text{rank}(\mathcal{M}) \cdot d$ and (b) its weight is at least a c-approximation to the weight of a maximum weight basis relative to S.

We study the dynamic setting where each element $e \in E$ has a dynamically changing weight $w_e \in \mathbb{R}^+$ and where S is a dynamically growing subset of E. A (c, d)*-approximate dynamic maximum weight oracle* is a data structure which maintains a (c, d)-approximate independent set B relative to S (i.e. in the matroid $\mathcal{M}/S$) while S and the weight of elements not in S can change. Initially S is an empty set and the data structure supports the following operations:

- Freeze(e): Add to S the element e, where e must belong to the current (c, d)-approximate basis relative to (the old) S and return the changes to B.
- Decrement(e, w): Return the weight w_e of $e \notin S$ to w, which is guaranteed to be smaller than the current weight of e and return the changes to B.
- ApproxBaseWeight$()$: Return the weight of the (c, d)-approximate independent set maintained by this data structure.

If $c = 1$ and $d = 1$ we call such a data structure a *dynamic maximum weight oracle relative to* S.

We will use (c, d)-approximate maximum weight oracles in the LazySamplingGreedy+ phase of the algorithm.

We will also need to augment this data structure with two additional sampling operations. Whenever the independent set B maintained by the data structure changes, we need to spend an extra $O(1)$ time updating a sampling data structure. This sampling data structure can be generically and efficiently implemented to augment any (c, d)-approximate maximum weight oracle as long as the (c, d)-approximate maximum weight oracle does not change the independent set B too much amortized over all calls to the data structure. This is described in the full version of the paper.

- Sample(t): Return a subset of $B \setminus S$, where each element is included independently with probability $\min\left(1, \frac{t}{w(B \setminus S)} w_e\right)$.
- UniformSample$()$: Return a uniformly random element from $B \setminus S$.

$(1 - \varepsilon)$-approximate independence oracles. For the second phase of our algorithm ContinuousGreedy we have a choice between two data structures. Both of them are unweighted, i.e., elements have no associated weights. We can either use an incremental (i.e. insertions-only) exact data structure or a dynamic $(1 - \varepsilon)$-approximate data structure, for a small $\varepsilon > 0$. Next we define both in more details.

Incremental independence oracle. The incremental independence oracle data structure maintains an independent set B and supports the following operation:

- TEST(e): Given an element e, decide if $B \cup \{e\}$ is independent. If so, output YES, otherwise output NO.
- INSERT(e): Given an element e such that $B \cup \{e\}$ is independent, add e to B.

$(1-\varepsilon)$*-approximate dynamic maximum independent set data structure.* Let $\varepsilon > 0$ be a small constant and let us call an independent set B of a matroid $\mathcal{M}$ that contains at least $(1-\varepsilon) \cdot \text{rank}(\mathcal{M})$ elements an $(1-\varepsilon)$-*approximate basis* of $\mathcal{M}$. The $(1-\varepsilon)$-approximate data structure maintains an $(1-\varepsilon)$-approximate basis B for a dynamically changing matroid $\mathcal{M}$ and supports the following operations.

- BATCH-INSERT(E'): Given a set E' of new elements, insert all elements of E' into the matroid $\mathcal{M}$ and compute a new $(1-\varepsilon)$-approximate basis B such that all elements that were in the basis before the update belong to B. Return all new elements that were introduced to B.
- DELETE(e): Given an element e of $\mathcal{M}$, delete e from $\mathcal{M}$ and update the independent set B such that it consists of at least $(1-\varepsilon) \cdot \text{rank}(\mathcal{M})$ elements of the new $\mathcal{M}$. If any new elements were added to B, return this set of new elements. Otherwise, return $\emptyset$.

Depending on which version of the algorithm we use, we will need either an exact incremental oracle or a $(1-\varepsilon)$-approximate dynamic maximum independent set data structure.

3.2 The LAZYSAMPLINGGREEDY+ algorithm

In this section, we describe the implementation of LAZYSAMPLINGGREEDY+.

The LAZYSAMPLINGGREEDY+ algorithm is inspired by the Random Greedy algorithm of Buchbinder, Feldman, Naor, and Schwartz [9] and the Lazy Sampling Greedy algorithm of Ene and Nguyen [12]. The algorithm begins with an initially empty solution S, and runs until the function $\tilde{f}(T) = f(T|S)$ has small enough marginals to reduce the sampling requirements of CONTINUOUSGREEDY.

We denote the weight of an element by $w_e(S) := f(S \cup \{e\}) - f(S)$, and weight$(T)$ to denote $\sum_{e \in T} w_e(S)$. The algorithm will only ever add elements to S, so by submodularity, $w_e(S)$ can only decrease as the algorithm runs (satisfying the requirements of Section 3.1). Throughout this algorithm, we use a (c, d)-approximate maximum-weight oracle (Section 3.1) that maintains a maximum-weight independent set B as the weights $w_e(S)$ are updated. For the sake of exposition, we defer proofs to the full version of the paper and assume $c \geq 1/2$, and $d \geq 1/2$.

Discretizing the marginal weights. Whenever S is changed, the weight $w_e(S)$ of all elements e can be changed. To reduce the number of weight changes, we use a standard rounding trick. Assume we have some constant-factor approximation M to $f(OPT)$ (which can be computed in $O(n)$ time via well-known algorithms [10]). Instead of maintaining $w_e(S)$ exactly, we round $w_e(S)$ to one of logarithmically many weight classes, that is, $w_e(S)$ belongs to weight class j if $w_e(S) \in ((1-\varepsilon)^{j+1}M, (1-\varepsilon)^j M]$, with the lowest class containing all weights from $[0, O(\varepsilon M/r)]$. The value of the rounded weight is then $\tilde{w}_e = (1-\varepsilon)^{j_e}$. We denote by *bucket* $\mathcal{B}^{(j)}$ all elements that belong to weight class j. Throughout the algorithm, we maintain estimates $\tilde{j}_e$ for the weight class that e is in (and thus estimates of w_e as well). An estimate is called *stale* if $w_e(S)$ is not actually in the weight class indicated by $\tilde{j}_e$. To achieve a multiplicative error of $(1-\varepsilon)$, it suffices for the number of different weight classes to be at most $O(\varepsilon^{-1} \log(r/\varepsilon))$, where r is the rank of the matroid. We denote by weight(B) the sum of current weight estimates over the set B.

The analysis of our algorithm works with any constant-factor approximation to $f(OPT)$ and any constant c-approximate maximum weight independent set data structure, albeit with slight changes in the approximation factors.

```
Algorithm 3.2: LazySamplingGreedy+.

$S \leftarrow \emptyset$, and set the weight estimate $\tilde{w}_e$ to $w_e(\emptyset)$ for all $e \in \mathcal{M}$.
$\mathcal{D} \leftarrow (c, d)$-approximate dynamic maximum weight oracle on $\mathcal{M}$ and $\tilde{w}$.

While $\mathcal{D}$.ApproxBaseWeight() $\geq \frac{50}{\varepsilon} f(OPT)$:
  1. $B' \leftarrow \mathcal{D}$.Sample($128 \log n$)
     (a random subset of $B \setminus S$, each element included independently with probability
     $p_e = \min\{1, \frac{128 \log n}{\tilde{w}(B \setminus S)} \tilde{w}_e\}$)
  2. Update the weights of all stale elements $e \in B'$ by computing $j_e$, $\tilde{w}_e = (1-\varepsilon)^{j_e}$
     and then calling $\mathcal{D}$.Decrement($e, \tilde{w}_e$).
  3. If less than half of the elements in $B'$ where $p_e = 1$ were stale (i.e. needed an
     update), and less than half of the elements in $B'$ where $p_e < 1$ were stale, then
     add $e = \mathcal{D}$.UniformSample() to $S$ by calling $\mathcal{D}$.Freeze($e$).
```

Note that in each iteration, we check and update only the weights of some random sample of elements. This is for efficiency; we show the estimated weight $\tilde{w}(B)$ is still correct in expectation up to a constant multiplicative factor. We begin the correctness proof by showing the following lemma.

▶ **Lemma 1.** *Assume $0 < \varepsilon < 1/3$. With high probability, if less than $\frac{1}{2}$ of the estimated weight of B' is in elements which are stale, then*

$$\sum_{e \in B \setminus S} w_e(S) \geq \frac{4}{\varepsilon} f(OPT).$$

Next, we show a bound on the computational complexity of LazySamplingGreedy+.

▶ **Lemma 2.** LazySamplingGreedy+ *uses at most $O(n\varepsilon^{-1} \log(r/\varepsilon))$ arithmetic operations, calls to f, and calls to the maximum weight data structure.*

Next we observe that S cannot have too many elements, otherwise $f(S)$ is close to $f(OPT)$ and we are done.

▶ **Observation 3.** *With high probability, S at the end of the algorithm has at most $\varepsilon r/2$ elements.*

▶ **Theorem 4.** *Let S be the set returned at the end of* LazySamplingGreedy+, *$OPT := \arg\max_{T \in \mathcal{M}} f(T)$, and $OPT^* := \arg\max_{T \in \mathcal{M}/S} f(T|S)$. The following inequality holds:*

$$\mathbf{E}[f(OPT^* \cup S)] \geq (1 - 2\varepsilon) f(OPT).$$

3.3 The ContinuousGreedy algorithm

In this section we discuss our implementation of the ContinuousGreedy algorithm. The basis of our algorithm is the fast implementation from [6], with additional speed-up due to the fact that the LazySamplingGreedy+ stage reduces the marginal values of the remaining elements. The previous section shows that our LazySamplingGreedy+ algorithm runs with at most $O_\varepsilon(n \log r)$ arithmetic operations, calls to f, and calls to the maximum weight data structure. In this section, we describe how LazySamplingGreedy+ helps the runtime of ContinuousGreedy. The proofs are given in the full version of the paper.

At the termination of LazySamplingGreedy+ it holds that $\tilde{w}(B) \leq \frac{50}{\varepsilon} f(OPT)$. Stale weights in B have true weights lower than its weight estimate $\tilde{w}_e$. Therefore, the true weight of elements of B must be also at most $\frac{50}{\varepsilon} f(OPT)$. Furthermore, since B is a constant-factor approximation to the true maximum weight basis $B^\star$, this implies that weight$(B^\star) = O(\frac{1}{\varepsilon} f(OPT))$.

Let $\tilde{f}(T) = f(T|S)$, where S is the set output at the termination of LazySampling-Greedy+. We observe that for any set $T \in \mathcal{M}/S$, $\sum_{e \in T} \tilde{f}(e) = O(\frac{1}{\varepsilon} f(OPT))$. When this is the case, [10] (Corollary 3.2) gives the following result:

▶ **Lemma 5** **([10]).** ContinuousGreedy *to obtain a* $(1 - 1/e - \varepsilon)$*-approximation uses* $O(n\varepsilon^{-2} \log(n/\varepsilon))$ *independent set data structure operations and* $O(n\varepsilon^{-5} \log^2(n/\varepsilon))$ *queries to* $\tilde{f}$*.*

In this section, we make two observations that improve the number of independent set queries by a log factor. The inner loop of the ContinuousGreedy algorithm is essentially a greedy algorithm which operates on a function derived from the multilinear extension of $\tilde{f}$: $g(T) = F(\mathbf{x} + \varepsilon \mathbf{1}_T)$ where $F(\mathbf{x}) = \mathbf{E}[\tilde{f}(R)]$, R sampled independently with probabilities x_e. The inner loop of ContinuousGreedy finds an increment of the current fractional solution $\mathbf{x}$ by running a greedy algorithm to approximate a maximum-weight basis with respect to the function g. Let us define $w_e(T) = g(T + e) - g(T)$ to be the marginal values of this function.

A fast implementation of this inner loop is the DescendingThreshold subroutine of Badanidiyuru and Vondrák [6], which also appears in the algorithm of [10]. This subroutine uses the marginal values $w_e(B)$ defined above; the expectation requires $O(\varepsilon^{-1} \log^2(n/\varepsilon))$ samples to estimate for the required accuracy of ContinuousGreedy. In the algorithms below, $w_e(S)$ can be thought of as a black-box that issues $O(\varepsilon^{-1} \log^2(n/\varepsilon))$ calls to the function $\tilde{f}$.

Algorithm 3.3: DescendingThreshold.

```
$B \leftarrow \emptyset$
$\tau \leftarrow \max_{\{e\} \in \mathcal{M}} w_e(\emptyset)$
While $\tau \geq \frac{\varepsilon}{r} f(O)$:
  1. Iterate through $e \in E$ one by one. If $B \cup \{e\} \in \mathcal{I}$ and $w_e(B) \geq \tau$, add $e$ to $B$.
     Otherwise, if $B \cup \{e\} \notin \mathcal{I}$, remove $e$ from $E$.
  2. $\tau \leftarrow (1 - \varepsilon)\tau$
Return $B$
```

The number of independent set queries in ContinuousGreedy is dominated by the first line of the while loop in DescendingThreshold.

We make two observations about the DescendingThreshold algorithm of Badanidiyuru and Vondrák [6], resulting in two modifications to DescendingThreshold that uses the *incremental independence oracle* and *approximate maximum independent set data structure* outlined in Section 3.1.

▶ **Observation 6.** *Only* $O(n/\varepsilon)$ *independence oracle queries are required. Furthermore, it is sufficient to use an* ***incremental*** **independence oracle.**

Thus, we can modify the DescendingThreshold of [6] by simply ignoring elements that have been previously rejected within the descending threshold greedy subprocedure (see Algorithm 3.4). This yields the following:

▶ **Lemma 7.** ContinuousGreedy *uses $O(n/\varepsilon)$ incremental independent set data structure operations and $O(n\varepsilon^{-5}\log^2(n/\varepsilon))$ queries to $\tilde{f}$.*

Algorithm 3.4: DT-Incremental.

```
$\mathcal{D} \leftarrow$ Incremental independence oracle maintaining a set $B$ (Section 3.1)
$\tau \leftarrow \max_{\{e\}\in\mathcal{M}} w_e(\emptyset)$
While $\tau \geq \frac{\varepsilon}{r} f(O)$:
 1. $E_\tau \leftarrow \{e \mid w_e(B) \geq \tau, e \in E \setminus B\}$
 2. Iterate through $e \in E_\tau$ one by one. If $\mathcal{D}$.Test($e$) returns YES and $w_e(B) \geq \tau$,
    call $\mathcal{D}$.Insert($e$) and add $e$ to $B$. Otherwise, if $B \cup \{e\} \notin \mathcal{I}$, remove $e$ from $E$.
 3. $\tau \leftarrow (1-\varepsilon)\tau$
Return $B$
```

An alternative observation

In the case of transversal matroids, exact incremental independence oracle with polylogarithmic update times are not known. Instead, we will make the following observation: An approximate *decremental maximal independent set* data structure can be used instead of an incremental independence oracle. This results in the modification of descending threshold described in Algorithm 3.5.

Algorithm 3.5: DT-ApproxIndepSet.

```
$\mathcal{D} \leftarrow$ Approximate dynamic maximum independent set data structure maintaining
      a set $B$ (Section 3.1)
$\tau \leftarrow \max_{\{e\}\in\mathcal{M}} w_e(\emptyset)$
While $\tau \geq \frac{\varepsilon}{r} f(O)$:
 1. $E_\tau \leftarrow \{e \mid w_e(B) \geq \tau, e \in E \setminus B\}$
 2. $B^+ \leftarrow \mathcal{D}$.Batch-Insert($E_\tau$)
 3. While $B^+ \neq \emptyset$:
    a. Get any $e \in B^+$. If $w_e(B) < \tau$, $D \leftarrow \mathcal{D}$.Delete($e$) and set $B^+ \leftarrow B^+ \cup D$.
    b. Remove $e$ from $B^+$.
 4. $\tau \leftarrow (1-\varepsilon)\tau$

Return $B$
```

▶ **Observation 8.** *Instead of an incremental independence oracle, ContinuousGreedy can be implemented with a decremental approximate maximum independent set data structure. Furthermore, ContinuousGreedy will only make $O(\varepsilon^{-1}\log r)$ calls to* Batch-Insert *and $O(n\varepsilon^{-1}\log r)$ calls to* Delete.

Rounding the fractional solutions. The ContinuousGreedy algorithm makes $O(1/\varepsilon)$ calls to Algorithm 3.3, and outputs a fractional solution that is a convex combination of the $O(1/\varepsilon)$ bases returned by these calls [6]. This fractional solution then needs to be rounded to an integral solution efficiently. In the full version, we show that the data structures we develop can speed up the rounding as well, leading to the overall cost being dominated by the LazySamplingGreedy+ and ContinuousGreedy phases.

3.4 Analysis of the overall framework

▶ **Lemma 9.** *The approximation returned by our framework has approximation ratio at least* $1 - 1/e - \varepsilon$*.*

Proof. Let S_0 be the set returned by LazySamplingGreedy+. Recall that $\tilde{f}(T) := f(T|S_0)$. By the results in the previous sections, there exists a set $OPT^\star$ such that $OPT^\star \cup S_0$ is independent and $\mathbf{E}[\tilde{f}(OPT^\star)] \geq (1-\varepsilon/2)f(OPT) - f(S_0)$ (by running LazySamplingGreedy+ with $\varepsilon/4$ instead of ε). Running continuous greedy on $\tilde{f}$ yields a $(1-1/e-\varepsilon/2)$-approximation S_1 to $OPT^\star$. Thus the final value of our solution $f(S_0 + S_1)$ is:

$$\begin{aligned}\mathbf{E}[f(S_0 + S_1)] &= \mathbf{E}[\tilde{f}(S_1) + f(S_0)]\\ &\geq (1 - 1/e - \varepsilon/2)\mathbf{E}[\tilde{f}(OPT^\star) + f(S_0)]\\ &\geq (1 - 1/e - \varepsilon/2)(1 - \varepsilon/2)f(OPT)\\ &\geq (1 - 1/e - \varepsilon)f(OPT).\end{aligned}$$ ◀

▶ **Observation 10.** *Our framework uses at most:*

- $O(n\varepsilon^{-5}\log^2(n/\varepsilon))$ *calls to the submodular function oracle* f*.*
- $O(n\varepsilon^{-1}\log(r/\varepsilon))$ *calls to an approximate maximum weight oracle (Section 3.2).*
- *Either* $O(n/\varepsilon)$ *incremental oracle data structure operations or* $O(\varepsilon^{-1}\log r)$ *calls to* BatchInsert *and* $O(n\varepsilon^{-1}\log r)$ *calls to* Delete *on a decremental approximate maximum independent set data structure (Section 3.3).*

The cost of evaluating f is dominated by the ContinuousGreedy phase (see Lemma 7), as LazySamplingGreedy+ only uses $O(n\varepsilon^{-1}\log(r/\varepsilon))$ oracle calls to f, where r is the rank of the matroid (Lemma 2).

4 Data structures for various matroids

In this section, we give dynamic (c, d)-approximate maximum weight oracles and $(1 - \varepsilon)$-approximate independence oracles for laminar matroids, graphic matroids, and transversal matroids.

Limitations for further improvements. For both the laminar, graphic, and transversal matroid, the total runtime of the data structure operations in LazySamplingGreedy+ and ContinuousGreedy is $O_\varepsilon(|\mathcal{M}|\log^2|\mathcal{M}|)$, where $|\mathcal{M}|$ is the number of matroid elements. Without improving the original ContinuousGreedy algorithm itself, it is impossible to improve the runtime further. This is because the ContinuousGreedy phase requires at least $O_\varepsilon(|\mathcal{M}|\log^2|\mathcal{M}|)$ oracle calls to f, which is at least $O(1)$ cost in any reasonable model of computing.

Weighted sampling on (c, d)-approximate independent sets. Our (c, d)-approximate maximum weight oracles in Section 3.1 require the ability to sample from the independent set they maintain. This sampling operation can be handled independently from the other operations of the data structure, by augmenting the Decrement and Freeze operations. As this augmentation is the same in all our data structures, we describe it in the full version.

4.1 Laminar matroids

Laminar matroids generalize uniform and partition matroids. In the full version of the paper we present a data structure $\mathcal{D}$ using top trees [2] that maintains a fully dynamic maximum weight basis for a laminar matroid under insertions and deletions of elements with arbitrary weights in $O(\log n)$ update time. This data structure satisfies the (c, d)-approximate maximum weight oracle requirements with $c = d = 1$ and satisfies the $(1 - \varepsilon)$-approximate independence oracle requirements with $\varepsilon = 0$.

Dynamic maximum weight oracle. The data structure $\mathcal{D}$ maintains the maximum weight basis under insertion and deletions. For Freeze(e) operations, we don't need to do anything. For Decrement(e, w) operations, we can simulate a decrement with the deletion of e and an insertion of e with the changed weight. As we show in the appendix of the full version, deleting and inserting an element removes at most the deleted element and adds at most one element to the maximum weight basis, and thus would never remove a frozen element from the basis whose weight never decreases.

Incremental independence oracle. This data structure can also be used to implement an incremental independence oracle as follows: Run the data structure $\mathcal{D}$ where every element has the same weight and that maintains a maximum basis B. Both Test and Insert can easily be handled by our data structure.

4.2 Graphic matroids

A graphic matroid can be represented with a weighted undirected graph $G = (V, E, w)$ where the weight of and edge $e \in E$ is given by $w(e)$.

Dynamic $(1/2, 1/2)$-approximate maximum weight oracle. To obtain an approximate maximum spanning tree of a graph $G = (V, E)$, take the largest edge incident to every vertex, with ties broken according to the edge numbering. For every vertex $v \in V$, let E_v denote the set of edges incident to v. We can store the weights of edges in E_v in a heap H_v and maintain that the maximum element of H_v is part of our approximate maximum spanning tree. It is easy to show that the set of edges maintained, F, is a forest with at least $1/2$ the weights and $1/2$ the number of edges of the optimal maximum spanning tree T.

For the correctness of the algorithm we show first that there cannot be any cycle in F. Assume by contradiction that there is a cycle C in F. Direct each edge in C towards the vertex where it was the maximum weight edge, breaking ties according to the vertex number. If C is a cycle, then C must give a directed cycle, where each edge is larger than the next edge in the directed cycle in the lexicographic order induced by the edge weight and the vertex number. This is a contradiction.

Approximation factor. Root T at an arbitrary vertex and consider the vertices of T starting at the leaves. We will use a simple charging argument to show that F has at least $1/2$ the weight of T and that $|F| \geq |T|/2$. The edge of a vertex v going to its parent u in the tree T can be charged to the largest weight edge leaving v, which is in F. Since each edge of $e \in F$ can be charged at most twice from the two endpoints of e by edges of lesser or the same weight, F has at least half the weight of T and at least half the edges as well.

Decrement(e, w): When the weight of an edge $e = (u, v) \in E$ changes to w, we update H_u and H_v accordingly. This may change the maximum weight edge incident to u or v, but we can lookup and accordingly modify our approximate maximum spanning tree with the new maximum weight edge of T_u and T_v in $O(\log n)$ time and report these changes.

Freeze(e): When we freeze an edge $e = (u, v) \in F$, we can contract the graph along the edge. To do so, we can merge the heaps H_u and H_v and associate the merged heap with the new merged vertex. This can be done in $O(\log n)$ time with binomial heaps or $O(1)$ time using the Fibonacci heaps of Fredman and Tarjan [14]. When we merge two vertices, the maximum weight edge incident to the new merged vertex may be added to the approximate maximum spanning tree.

Incremental independence oracle. Unweighted incremental maximum spanning tree involves checking if inserting any edge increases the size of the spanning tree. This can be done in $O(\alpha(n))$ update and query time with the disjoint set union data structure of Tarjan [24].

4.3 Transversal matroids

Representation of transversal matroids. As stated in Section 2, we assume that our transversal matroids are given as minimal representations. This means that the matroid $\mathcal{M}$ is represented by a bipartite graph $G = ((L, R), E)$ where $|R| = \text{rank}(\mathcal{M})$. For sake of notation let $n = |L|$ and $m = |E|$. As a reminder, each element of the matroid corresponds to a node in L, and an independent set I is a subset of L such that there exists a matching in G that matches every element of I. We will let $N(v)$ denote the neighbors of v in G, that is $N(v) = \{u \mid (u, v) \in E\}$. If $m > n^2$ we can remove neighbours from each vertex in L until their degree is at most n. This doesn't affect whether a vertex belongs to an independent set, as it can always be matched. This reduces m to at most $O(n^2)$.

Dynamic $(1 - \varepsilon, 1/2)$-approximate maximum weight oracles. Recall that in the case of transversal matroids, the weighted setting of LazySamplingGreedy+ leads to a dynamic matching problem on a *vertex-weighted* bipartite graph $G = ((L, R), E)$, where each vertex $\ell \in L$ has a non-negative weight $w(\ell)$ and all edges incident to L have weight $w(\ell)$. We assume that each vertex in L has a value $w_{min} \geq O(w_{max}\varepsilon/n)$ such that we may ignore the weight of any vertex that drops below w_{min}. For the purposes of LazySamplingGreedy+, we stop if the maximum weight basis decreases below $O(f(OPT)) \geq w_{max}$, and so even if we discard all items with weight less than $O(w_{max}\varepsilon/n)$, we can discard at most an ε fraction of $f(OPT)$. Thus after appropriate multiplicative rescaling, we may assume that $w_{min} = 1$ and $w_{max} = (1 + \varepsilon)^k$ for $k = O(\log_{1+\varepsilon} n)$. Furthermore we may assume that the weight of any $\ell \in L$ is $(1+\varepsilon)^j$ for some $j \geq 0$ as we can round all weights in the range of $[(1+\varepsilon)^j, (1+\varepsilon)^{j+1}]$ down to the nearest $(1 + \varepsilon)^j$ and lose only a $(1 + \varepsilon)^{-1}$ factor in the value of the solution.

We will design a data structure that maintains a matching M such that whenever a Decrement(ℓ, w) operation is performed on $\ell \in L$, then ℓ will be the only node of L that may potentially become unmatched in M. We will call a data structure that has this property *L-stable*. The basis we output will be the set of nodes of L matched in M. Note that L-stable data structures can handle the Freeze operation by not doing anything and always returning an empty set. No frozen element will be removed from the basis because frozen elements are never decremented.

The high level idea of our algorithm is as follows: We want to maintain a maximal matching according to some weights, as this guarantees that at least half as many nodes of L are matched as in the optimum solution. The question is just which weights to choose and which algorithm to use to guarantee maximality while fulfilling L-stability. Note that L-stability allows edges in the matching to change, just un-matching a matched vertex of L is forbidden. For this reason we chose an algorithm that is greedy for the vertices in R, i.e., each vertex in R is matched with a neighbor of largest weight for a suitable choice of weight. In order to maintain the invariant at every vertex r of R our greedy algorithm allows r to "steal" the matched neighbor l of another vertex r' of R. This maintains L-stability as l remains matched. However, the newly un-matched vertex r' might want to steal l right back from r. To avoid this, we do not use the original weights in the greedy algorithm, but instead we use "virtual weights" that are initialized by the original weights and that decrease by a factor of $(1 + \varepsilon)$ whenever l is (re-)matched. This makes l less attractive for r' and, as l

is never re-matched when its weight is below 1, it also guarantees that l is only re-matched $\tilde{O}_\varepsilon(1)$ times in total over all decrement operations. For formal details and the proof, see the full version of the paper.

▶ **Theorem 11.** *Given a bipartite graph $G = ((L, R), E)$ and a value w_{min}, there exists a L-stable data structure that handles* Decrement *operations and maintains a $(1 - \varepsilon, 1/2)$-approximate maximum weighted matching provided that the maximum weighted matching has cost at least w_{min}. The total running time for preprocessing and all operations as well as the total number of changes to the set of matched vertices is $O(|E|(1/\varepsilon + \log |L|))$. Furthermore, the matching maintained is maximal.*

$(1 - \varepsilon)$-approximate dynamic maximum independent set data structure. The fastest known algorithm for incremental maximum bipartite matching takes $O(m\sqrt{n})$ total time [8]. However, given a bipartite graph $G = (L, R)$ there is a $(1 - \varepsilon)$-approximate maximum matching data structure $\mathcal{D}_M$ for deletions of vertices in L [8]. It has three properties that are crucial for our algorithm: (1) It does not unmatch a previously matched vertex of L as long as it is not deleted, (2) it maintains an explicit integral matching, i.e., it stores at each vertex whether and if so, along which edge it is matched, and (3) the total time for computing the initial matching and all vertex deletions is $O((m + |L|)/\varepsilon)$, where m is the number of edges in the initial graph.

Given an initial graph G_0 and a partial matching B of G_0 this algorithm can be modified to guarantee that the initial $(1 - \varepsilon)$-approximate matching computed for G_0 matches all vertices of $B \cap L$. See teh full version of the paper for details. We use this data structure $\mathcal{D}_M$ to implement a $(1 - \varepsilon)$-approximate dynamic maximum independent set data structure for the transversal matroid as follows:

Batch-Insert(E'): Let B be the $(1 - \varepsilon)$-approximate matching before the update. Initialize a new data $\mathcal{D}_M$ with all current elements and compute an initial $(1-\varepsilon)$-approximate matching computed for G_0 matching all vertices in $B \cap L$. This is possible by the discussion above.

Delete(e): Execute a vertex deletion of vertex e in $\mathcal{D}_M$.

Test(e): Return YES if e is matched and NO otherwise.

▶ **Lemma 12.** *Given a transversal matroid there exists a $(1 - \varepsilon)$-approximate dynamic maximum independent set data structure such that each* Batch-Insert*(B, E_1, E_2) and all* Delete *operations until the next* Batch-Insert *take $O((m' + |E_1| + |E_2|)/\varepsilon)$ total worst-case time and each* Test *operation takes $O(1)$ worst-case time.*

References

1 Amir Abboud and Virginia Vassilevska Williams. Popular conjectures imply strong lower bounds for dynamic problems. In *55th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2014, Philadelphia, PA, USA, October 18-21, 2014*, pages 434–443. IEEE Computer Society, 2014. doi:10.1109/FOCS.2014.53.

2 Stephen Alstrup, Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. Maintaining information in fully dynamic trees with top trees. *ACM Trans. Algorithms*, 1(2):243–264, 2005. doi:10.1145/1103963.1103966.

3 Moshe Babaioff, Jason Hartline, and Robert Kleinberg. Selling banner ads: Online algorithms with buyback. In *Fourth workshop on ad auctions*, 2008.

4 Moshe Babaioff, Nicole Immorlica, and Robert Kleinberg. Matroids, secretary problems, and online mechanisms. In Nikhil Bansal, Kirk Pruhs, and Clifford Stein, editors, *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2007, New Orleans, Louisiana, USA, January 7-9, 2007*, pages 434–443. SIAM, 2007. URL: `http://dl.acm.org/citation.cfm?id=1283383.1283429`.

5 Ashwinkumar Badanidiyuru, Baharan Mirzasoleiman, Amin Karbasi, and Andreas Krause. Streaming submodular maximization: massive data summarization on the fly. In Sofus A. Macskassy, Claudia Perlich, Jure Leskovec, Wei Wang, and Rayid Ghani, editors, *The 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '14, New York, NY, USA – August 24 – 27, 2014*, pages 671–680. ACM, 2014. `doi:10.1145/2623330.2623637`.

6 Ashwinkumar Badanidiyuru and Jan Vondrák. Fast algorithms for maximizing submodular functions. In Chandra Chekuri, editor, *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2014, Portland, Oregon, USA, January 5-7, 2014*, pages 1497–1514. SIAM, 2014. `doi:10.1137/1.9781611973402.110`.

7 Jeff A. Bilmes. Submodularity in machine learning and artificial intelligence. *CoRR*, abs/2202.00132, 2022. `arXiv:2202.00132`.

8 Bartlomiej Bosek, Dariusz Leniowski, Piotr Sankowski, and Anna Zych. Online bipartite matching in offline time. In *55th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2014, Philadelphia, PA, USA, October 18-21, 2014*, pages 384–393. IEEE Computer Society, 2014. `doi:10.1109/FOCS.2014.48`.

9 Niv Buchbinder, Moran Feldman, Joseph Naor, and Roy Schwartz. Submodular maximization with cardinality constraints. In Chandra Chekuri, editor, *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2014, Portland, Oregon, USA, January 5-7, 2014*, pages 1433–1452. SIAM, 2014. `doi:10.1137/1.9781611973402.106`.

10 Niv Buchbinder, Moran Feldman, and Roy Schwartz. Comparing apples and oranges: Query trade-off in submodular maximization. *Math. Oper. Res.*, 42(2):308–329, 2017. `doi:10.1287/moor.2016.0809`.

11 Gruia Călinescu, Chandra Chekuri, Martin Pál, and Jan Vondrák. Maximizing a monotone submodular function subject to a matroid constraint. *SIAM J. Comput.*, 40(6):1740–1766, 2011. `doi:10.1137/080733991`.

12 Alina Ene and Huy L. Nguyen. Towards nearly-linear time algorithms for submodular maximization with a matroid constraint. In Christel Baier, Ioannis Chatzigiannakis, Paola Flocchini, and Stefano Leonardi, editors, *46th International Colloquium on Automata, Languages, and Programming, ICALP 2019, July 9-12, 2019, Patras, Greece*, volume 132 of *LIPIcs*, pages 54:1–54:14. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019. `doi:10.4230/LIPIcs.ICALP.2019.54`.

13 Uriel Feige. A threshold of ln n for approximating set cover. *J. ACM*, 45(4):634–652, 1998. `doi:10.1145/285055.285059`.

14 Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34(3):596–615, 1987. `doi:10.1145/28869.28874`.

15 Manoj Gupta and Richard Peng. Fully dynamic $(1+\varepsilon)$-approximate matchings. In *54th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2013, 26-29 October, 2013, Berkeley, CA, USA*, pages 548–557. IEEE Computer Society, 2013. `doi:10.1109/FOCS.2013.65`.

16 Monika Henzinger, Sebastian Krinninger, Danupon Nanongkai, and Thatchaphol Saranurak. Unifying and strengthening hardness for dynamic problems via the online matrix-vector multiplication conjecture. In Rocco A. Servedio and Ronitt Rubinfeld, editors, *Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing, STOC 2015, Portland, OR, USA, June 14-17, 2015*, pages 21–30. ACM, 2015. `doi:10.1145/2746539.2746609`.

17 Monika Henzinger, Ami Paz, and Stefan Schmid. On the complexity of weight-dynamic network algorithms. In Zheng Yan, Gareth Tyson, and Dimitrios Koutsonikolas, editors, *IFIP Networking Conference, IFIP Networking 2021, Espoo and Helsinki, Finland, June 21-24, 2021*, pages 1–9. IEEE, 2021. doi:10.23919/IFIPNetworking52078.2021.9472803.

18 Ehsan Kazemi, Marko Mitrovic, Morteza Zadimoghaddam, Silvio Lattanzi, and Amin Karbasi. Submodular streaming in all its glory: Tight approximation, minimum memory and low adaptive complexity. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, volume 97 of *Proceedings of Machine Learning Research*, pages 3311–3320. PMLR, 2019. URL: http://proceedings.mlr.press/v97/kazemi19a.html.

19 Hung Le, Lazar Milenkovic, Shay Solomon, and Virginia Vassilevska Williams. Dynamic matching algorithms under vertex updates. In Mark Braverman, editor, *13th Innovations in Theoretical Computer Science Conference, ITCS 2022, January 31 – February 3, 2022, Berkeley, CA, USA*, volume 215 of *LIPIcs*, pages 96:1–96:24. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPIcs.ITCS.2022.96.

20 Wenxin Li, Moran Feldman, Ehsan Kazemi, and Amin Karbasi. Submodular maximization in clean linear time. *CoRR*, abs/2006.09327, 2022. arXiv:2006.09327.

21 Paul Liu and Jan Vondrák. Submodular optimization in the mapreduce model. In Jeremy T. Fineman and Michael Mitzenmacher, editors, *2nd Symposium on Simplicity in Algorithms, SOSA 2019, January 8-9, 2019, San Diego, CA, USA*, volume 69 of *OASIcs*, pages 18:1–18:10. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019. doi:10.4230/OASIcs.SOSA.2019.18.

22 George L. Nemhauser and Laurence A. Wolsey. Best algorithms for approximating the maximum of a submodular set function. *Math. Oper. Res.*, 3(3):177–188, 1978.

23 Alexander Schrijver. *Combinatorial optimization: polyhedra and efficiency*, volume 24. Springer, 2003.

24 Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *J. ACM*, 22(2):215–225, 1975. doi:10.1145/321879.321884.

25 Mark Wilhelm, Ajith Ramanathan, Alexander Bonomo, Sagar Jain, Ed H. Chi, and Jennifer Gillenwater. Practical diversified recommendations on youtube with determinantal point processes. In Alfredo Cuzzocrea, James Allan, Norman W. Paton, Divesh Srivastava, Rakesh Agrawal, Andrei Z. Broder, Mohammed J. Zaki, K. Selçuk Candan, Alexandros Labrinidis, Assaf Schuster, and Haixun Wang, editors, *Proceedings of the 27th ACM International Conference on Information and Knowledge Management, CIKM 2018, Torino, Italy, October 22-26, 2018*, pages 2165–2173. ACM, 2018. doi:10.1145/3269206.3272018.

26 Zhou Xu and Brian Rodrigues. A 3/2-approximation algorithm for the multiple tsp with a fixed number of depots. *INFORMS Journal on Computing*, 27(4):636–645, 2015.

27 Da Wei Zheng and Monika Henzinger. Multiplicative auction algorithm for approximate maximum weight bipartite matching. *CoRR*, abs/2301.09217, 2023. doi:10.48550/arXiv.2301.09217.