Faster submodular maximization for several classes of matroids

Monika Henzinger*
Institute of Science and Technology Austria

Paul Liu
Stanford University

Jan Vondrák
Stanford University

Da Wei Zheng
University of Illinois Urbana-Champaign

arXiv:2305.00122v1 [cs.DS] 28 Apr 2023

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 [EN19]. 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 [EN19]. In fact, the running time of our algorithm cannot be improved within the fast continuous greedy framework of Badanidiyuru and Vondrák [BV14].

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 [AHdLT05] 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.

---
*

1   Introduction

Submodular optimization is encountered in a variety of applications – combinatorial optimization, information retrieval, and machine learning, to name a few [Bil22]. 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 [BIK07, BHK08], laminar and partition matroids capture capacity constraints on subsets which are widely used in recommendation settings (e.g. YouTube video recommendation algorithm [WRB+18]), and graphic matroids appear in applications for approximating Metric TSP [XR15]. 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 [CCPV11], and the factor of $1 - 1/e$ is also known to be optimal [NW78, Fei98]. 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 [BMKK14] for the cardinality constraint. With more work, this technique can be extended to give approximations close to $1 - 1/e$ [BV14] 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 [BV14]. (The exact running time would depend on the implementation of these queries, which [BV14] 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 [KMZ+19, LFKK22, LV19] and the references therein); we do not discuss these directions in this paper.

A recent line of work initiated by Ene and Nguyen [EN19] attempts to develop a "nearly-linear" continuous greedy algorithm, i.e., with a running time of $n \cdot \text{poly}(1/\varepsilon, \log n)$. 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 [BFS17], 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 [EN19], even determining feasibility may take longer than linear time for certain matroids. One such example is for 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 [EN19], 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

---

[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. The technical developments behind these results are on two fronts:
(i) a refinement of the optimization framework of [EN19] 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 [EN19] 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 [EN19] 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 [AHdLT05].

- 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.

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 [ZH23]. We briefly mention a few relevant works:

---

[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.

- There is a conditional lower bound based on the OMv conjecture [HKNS15] 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 $\delta > 0$ [HPS21]. The reduction from [HPS21] 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 [LMSW22] 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. [BLSZ14] 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 [GP13] 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 [EN19]. 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 [CCPV11, BV14], 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 [EN19] 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 [EN19], 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 [AW14], 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 [Tar75]. 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 [AHdLT05] 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 [ZH23] for fast $(1 - \varepsilon)$-approximate maximum weight matching algorithm in bipartite graphs based on the multiplicative weight update idea. The analysis within [ZH23] 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 $f : 2^{\mathcal{N}} \to \mathbb{R}$ 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 Section 5.

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 [Sch03], 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 [EN19]. 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 $rank(\mathcal{M}) \cdot d$ 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 Appendix C.

- 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 [BFNS14] and the Lazy Sampling Greedy algorithm of Ene and Nguyen [EN19]. 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 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 [BFS17]). 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 $\text{weight}(B)$ 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 $\mathcal{D}.\text{APPROXBASEWEIGHT}() \geq \frac{50}{\varepsilon} f(OPT)$:

1. $B' \leftarrow \mathcal{D}.\text{SAMPLE}(128 \log n)$
   (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 $\mathcal{D}.\text{DECREMENT}(e, \tilde{w}_e)$.

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 $e = \mathcal{D}.\text{UNIFORMSAMPLE}()$ to S by calling $\mathcal{D}.\text{FREEZE}(e)$.

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 3.1. *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*

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

Proof. First, observe that $\sum_{e \in S} \tilde{w}_e \leq (1 + \varepsilon) f(S)$. This is because each element in S has their weight $\tilde{w}_e$ frozen when they enter S, and at that point $\tilde{w}_e$ is at most a $(1 + \varepsilon)$ factor off from their true marginal value on top of S. Also, since S is a feasible solution, we have $f(S) \leq f(OPT)$. As long as the while loop is running, we have $\sum_{j \in B} \tilde{w}_e \geq \frac{50}{\varepsilon} f(OPT)$, and hence

$$\sum_{e \in B \setminus S} \tilde{w}_e \geq \sum_{e \in B} \tilde{w}_e - (1 + \varepsilon) f(S) \geq \frac{48}{\varepsilon} f(OPT).$$

Next, we observe that the expected number of elements included in B' is $\sum_{e \in B \setminus S} p_e \leq 128 \log n$.

9

Hence by the Chernoff bound,

$$\Pr[|B'| > 160\log n] < e^{-(1/4)^2 128\log n/3} < \frac{1}{n^{2.5}}$$

and with high probability we have $O(\log n)$ elements to check.

Let $\tilde{B}$ be the subset of $B \setminus S$ which is not stale. We distinguish two types of elements in $B \setminus S$: large elements, for which $\tilde{w}_e > \frac{\tilde{w}(B \setminus S)}{128\log n}$, and small elements, for which $\tilde{w}_e \leq \frac{\tilde{w}(B \setminus S)}{128\log n}$. At least half of the weight of $B \setminus S$ must come from either the large elements or the small elements. In the first case, we check all the large elements, and if more than half of them are stale, we skip line 3. Hence the only way we can execute line 3 is that the non-stale large elements contain weight at least $\frac{1}{4}\sum_{e \in B \setminus S} \tilde{w}_e$.

In the second case, let B_s denote the small elements in $B \setminus S$; we have $\sum_{e \in B_s} \tilde{w}_e \geq \frac{1}{2}\sum_{e \in B \setminus S} \tilde{w}_e$, and each small element is sampled with probability $p_e = \frac{128\log n}{\tilde{w}(B \setminus S)}\tilde{w}_e$. Therefore, the expected number of small elements chosen in B' is $\mathbf{E}[|B' \cap B_s|] \geq 64\log n$, and by the Chernoff bound,

$$\Pr[|B' \cap B_s| < 48\log n] < e^{(1/4)^2 64\log n/2} = \frac{1}{n^2}.$$

On the other hand, if the weight of non-stale small elements is less than $\frac{1}{8}\sum_{e \in B \setminus S} \tilde{w}_e$, the expected number of such elements chosen in B' is $\mathbf{E}[|B' \cap B_s \cap \tilde{B}|] \leq 16\log n$, and again by the Chernoff bound, the probability that more than this quantity is more than $24\log n$ is less than $e^{-(1/2)^2 16\log n/2} = 1/n^2$.

Hence either way, if the weight of non-stale elements in $B \setminus S$ is less than $\frac{1}{8}\sum_{e \in B \setminus S} \tilde{w}_e$, with high probability we do not execute line 3. Hence we can assume that $\sum_{e \in \tilde{B}} w_e(S) \geq (1 - \varepsilon)\sum_{e \in \tilde{B}} \tilde{w}_e \geq \frac{1-\varepsilon}{8}\sum_{e \in B \setminus S} \tilde{w}_e$ whenever we execute line 3. Then,

$$\sum_{e \in B \setminus S} w_e(S) \geq \sum_{e \in \tilde{B}} w_e(S) \geq \frac{1-\varepsilon}{8}\sum_{e \in B \setminus S} \tilde{w}_e \geq \frac{1-\varepsilon}{8}\cdot\frac{48}{\varepsilon}f(OPT) \geq \frac{4}{\varepsilon}f(OPT).$$

The second inequality is due to the fact that $\sum_{e \in B \setminus S} \tilde{w}_e \geq \frac{48}{\varepsilon}OPT$, and the fact that at least $1/8$ of the weight in $B \setminus S$ is not stale. The last inequality is due to the assumption that $\varepsilon < 1/3$. $\square$

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

Lemma 3.2. LazySamplingGreedy+ *uses at most $O(n\varepsilon^{-1}\log(r/\varepsilon))$ arithmetic operations, calls to f, and calls to the maximum weight data structure.*

Proof. Recall that throughout the algorithm set S only increases and, thus, the weight of each element only decreases. Thus, by discretizing the weights, the weight of an element can only be changed at most $O(\varepsilon^{-1}\log r)$ times and the total number of weight changes is at most $O(n\varepsilon^{-1}\log(r/\varepsilon))$. When all weights are at the lowest class, no element in B contributes more than $O(\varepsilon f(OPT)/r)$. With enough weight classes (e.g. more than $10\log_{1-\varepsilon}(\varepsilon/r)$ classes), the constant on the big-O is less than 1. Thus $\sum_{e \in B} \tilde{w}_e < \varepsilon f(OPT) < f(OPT)/\varepsilon$, so the while loop must terminate.

The most costly part of the algorithm is line 2. Each time line 2 executes, $O(\log n)$ weight changes and calls to f are made. Thus the overall cost of line 2 throughout the entire algorithm is at most $O(n\varepsilon^{-1}\log(r/\varepsilon))$. $\square$

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

Observation 3.1. *With high probability, S at the end of the algorithm has at most $\varepsilon r/2$ elements.*

Proof. Observe that whenever we add an element to S, it is a random element from a set $B \setminus S$ satisfying $\sum_{e \in B \setminus S} \geq \frac{4}{\varepsilon} f(OPT)$ with high probability (Lemma 3.1). Hence, whenever we add an element to S, $f(S)$ increases by at least $(4 - o(1))f(OPT)/(\varepsilon r)$ in expectation, and this is true conditioned on any prior history. Hence, by martingale concentration, the value after including t elements is with high probability at least $2t \cdot f(OPT)/(\varepsilon r)$. If there was non-negligible probability that the algorithm includes more than $\varepsilon r/2$ elements, then with some probability the value of $f(S)$ in theses cases would exceed $f(OPT)$, which is impossible (as S is always a feasible solution). Hence, with high probability, the algorithm does not include more than $\varepsilon r/2$ elements. $\square$

Theorem 3.3. *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:*

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

Proof. By Observation 3.1, we can assume that the algorithm includes at most $\varepsilon r/2$ elements in S. Let $r = \text{rank}(\mathcal{M})$, and $\{s_1, s_2, \ldots, s_t\}$ be the sequence of elements we add to S. Let OPT be the basis maximizing f and order $OPT = \{o_1, o_2, \ldots, o_r\}$ such that $\{s_1, \ldots, s_i\} \cup \{o_{i+1}, \ldots, o_r\}$ is independent for all i. We can arrange this ordering so that in each step, o_i is a uniformly random one of the remaining element of OPT: Given a base $\{s_1, \ldots, s_{i-1}, o_i, \ldots, o_r\}$, $S_{i-1} = \{s_1, \ldots, s_{i-1}\}$, and another base B containing S_{i-1}, there is a matching between $B \setminus S_{i-1}$ and $\{o_i, \ldots, o_n\}$ such that for any $e \in B \setminus S_{i-1}$, it is possible to add e to S_{i-1} and remove its matching optimal element (assume o_i), so that $\{s_1, \ldots, s_i, o_{i+1}, \ldots, o_n\}$ is still a base. Given that we choose $e \in B \setminus S_{i-1}$ uniformly at random, the choice of an optimal element to remove is also uniformly random.

To simplify the notation, we define $S_i := \{s_1, \ldots, s_i\}$ and $O_i := \{o_i, \ldots, o_r\}$, with the convention that $S_0 = O_{r+1} = \emptyset$ and use $f(s_i|S_{i-1})$ (resp. $f(o_i|O_{i-1})$) to denote $f(S_i) - f(S_{i-1})$ (resp. $f(O_i) - f(O_{i-1})$). Note that it follows that $f(s_i|S_{i-1}) = w_{s_i}(S_{i-1})$.

Following [EN19], we will show that

$$\mathbf{E}[f(s_i|S_{i-1})] \geq \frac{1}{2\varepsilon}\mathbf{E}[f(o_i|O_{i+1})]. \tag{1}$$

Adding the inequality Equation (1) for $i = 1, \ldots, t$, we have

$$\mathbf{E}[f(S_t) - f(\emptyset)] = E[f(S)] \geq \frac{1}{2\varepsilon}\mathbf{E}[f(O_1) - f(O_t)] = \frac{1}{2\varepsilon}\mathbf{E}[f(OPT) - f(O_t)].$$

Since $OPT^* \cup S$ and $O_t \cup S$ are both independent, we would then have

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

We now show Equation (1). Fix an iteration i and all random choices $s_1, s_2, \ldots, s_{i-1}$ up to iteration i. By Lemma 3.1, w.h.p. $\sum_{e \in B \setminus S_{i-1}} w_e(S_{i-1}) \geq \frac{4}{\varepsilon} f(OPT)$ on line 3 for all i. Thus choosing a random $e \in B \setminus S_{i-1}$ as s_i yields

$$\mathbf{E}[f(s_i|S_{i-1})] \geq \frac{f(OPT)}{\varepsilon|B \setminus S_{i-1}|} \geq \frac{2f(OPT)}{\varepsilon(r - i)}. \tag{2}$$

As discussed above, since s_i is chosen uniformly at random from $B \setminus S_{i-1}$, one of the $|B \setminus S_{i-1}|$ elements in O_i is chosen as o_i. By our assumptions on c and d for the (c,d)-approximate maximum weight oracle, $|B| \geq r/2$. Furthermore, $i \leq \varepsilon r$ due to Observation 3.1. Thus $|B \setminus S_{i-1}| \geq r/2 - i \geq (r-i)/4$ for $\varepsilon < 1/3$. This means that any element in O_i has probability at most $4/(r-i)$ chance of being chosen as o_i. Thus

$$\mathbf{E}[f(o_i|O_{i+1})] \leq \sum_{o \in O_i} \frac{4}{r-i} f(o|O_i - o) \tag{3}$$

$$\leq \frac{4}{r-i} \sum_{j=i}^{r} f(o_j|O_{j+1}) \tag{4}$$

$$\leq \frac{4f(OPT)}{r-i}. \tag{5}$$

Combining Equation (2) and Equation (5) yields $\mathbf{E}[f(s_i|S_{i-1})] \geq \frac{1}{2\varepsilon} \mathbf{E}[f(o_i|O_{i+1})]$. $\qquad \square$

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 [BV14], 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.

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 $\text{weight}(B^\star) = O(\frac{1}{\varepsilon} f(OPT))$.

Let $\tilde{f}(T) = f(T|S)$, where S is the set output at the termination of LAZYSAMPLINGGREEDY+. 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, [BFS17] (Corollary 3.2) gives the following result:

Lemma 3.4 ([BFS17]). 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 [BV14], which also appears in the algorithm of [BFS17]. 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 [BV14], 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 3.2. *Only $O(n/\varepsilon)$ independence oracle queries are required. Furthermore, it is sufficient to use an **incremental** independence oracle.*

Proof. DESCENDINGTHRESHOLD goes over the elements of $\mathcal{M}$ repeatedly in different orders, adding elements incrementally to a solution if they are independent from the current solution. Once an element is determined to be incompatible with the current solution, it remains incompatible for the duration of the algorithm. This is because the solution we build is incremental, and incompatibility with the solution implies incompatibility with all supersets of that solution due to matroid properties. Thus the independence oracle is only called $O(n)$ times per run of the DESCENDINGTHRESHOLD algorithm. Furthermore, the solution of descending threshold greedy is built incrementally, so only an incremental independence oracle is need.

Since DESCENDINGTHRESHOLD is executed ε^{-1} times, this implies that $O(n/\varepsilon)$ independence oracle queries are required in total. $\qquad\square$

Thus, we can modify the DESCENDINGTHRESHOLD of [BV14] by simply ignoring elements that have been previously rejected within the descending threshold greedy subprocedure (see Algorithm 3.4). This yields the following:

Lemma 3.5. 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 $\mathcal{D}.\text{TEST}(e)$ returns YES and $w_e(B) \geq \tau$, call $\mathcal{D}.\text{INSERT}(e)$ 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. $B^+ \leftarrow \mathcal{D}.\text{BATCH-INSERT}(E_\tau)$
>
> 3. While $B^+ \neq \emptyset$:
>
>   (a) Get any $e \in B^+$. If $w_e(B) < \tau$, $D \leftarrow \mathcal{D}.\text{DELETE}(e)$ and set $B^+ \leftarrow B^+ \cup D$.
>   (b) Remove e from B^+.
>
> 4. $\tau \leftarrow (1 - \varepsilon)\tau$
>
> Return B

Observation 3.3. *Instead of an incremental independence oracle,* CONTINUOUSGREEDY *can be implemented with a stable approximate maximum basis data structure. Furthermore,* CONTINUOUS-GREEDY *will only make* $O(\varepsilon^{-1} \log r)$ *calls to* BATCH-INSERT *and* $O(n\varepsilon^{-1} \log r)$ *calls to* DELETE.

Proof. In DESCENDINGTHRESHOLD, weights are placed into buckets of geometrically decreasing weights in $(1 - \varepsilon)$. For the highest non-empty bucket, we can build a $(1 - \varepsilon)$-approximate decremental independent set oracle data structure (with BATCH-INSERT). If there is any element in the

approximate maximal independent set of the current bucket which we have not added to our solution yet, we can add it to our solution. However, when we recompute the new value of the element, it may have shifted to a lower bucket. If this is the case, we remove the element from our data structure (with DELETE), move the element to a lower bucket and continue. When we are finished with a bucket, we initialize the data structure for the next bucket with the subset of elements in the solution so far. At the end we may miss adding at most ε items from each bucket, so this results in a solution of $(1 - \varepsilon)$ of the total weight. □

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 [BV14]. This fractional solution then needs to be rounded to an integral solution efficiently. In Appendix B, 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 3.6. *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:

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

Observation 3.4. *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 BATCH-INSERT 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 3.5), as LAZYSAMPLINGGREEDY+ only uses $O(n\varepsilon^{-1} \log(r/\varepsilon))$ oracle calls to f, where r is the rank of the matroid (Lemma 3.2).

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 Continuous-Greedy 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 Appendix C.

4.1 Laminar matroids

Laminar matroids generalize uniform and partition matroids. In Section 5 we present a data structure $\mathcal{D}$ using top trees [AHdLT05] 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. By Appendix A, deleting and inserting the 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 [FT87]. 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 [Tar75].

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 $|R| = \mathrm{rank}(\mathcal{M})$. 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 DECRE-MENT(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 Section 6.

Theorem 4.1. *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. 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 [BLSZ14] which achieves amortized $O(\varepsilon^{-1})$ time per delete operation. 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 Section 6 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 4.2. *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.*

We show here how to slightly modify the algorithm of [BLSZ14] so that it fulfills the following condition (C): *Given an initial bipartite graph $G_0 = ((L, R), E)$ and a partial matching B in G_0 the algorithm guarantees that the initial $(1 - \varepsilon)$-approximate matching computed for G_0 matches all vertices of $B \cap L$.*

The algorithm for the deletions of vertices in L of [BLSZ14] is actually based on an algorithm for insertions of vertices in R that guarantees that there is no augmenting path of length $k := 2 + 2/\varepsilon > 2$. Whenever a vertex l of L is supposed to be deleted, the algorithm inserts a new vertex $r \in R$ with a single edge, namely to l. Let us call the set of vertices inserted into R for this purpose R'. As the algorithm guarantees that no augmenting path of length 1 exists and l is unmatched, it follows that after the insertion l will be matched to r. Let us denote by G the initial graph without the deleted vertices and without the vertices in R' and by G' the initial graph with the vertices in R'. Lemma VI.5 in [BLSZ14] shows that the following inequality holds:

$$|M(G')| - |R'| \geq (1 - \varepsilon)|M(G)|,$$

i.e. even when not counting the matched edges incident to the $|R'|$ deleted edges, the matching in G' is a $(1 - \varepsilon)$-approximation of the optimal matching in G.

We now describe how to modify the algorithm to fulfill condition (C). Recall that in the decremental algorithm R is given initially and never changes, only in the implementation of the decremental algorithm by an incremental algorithm R is augmented by R'. Let us denote the initial set R by R_0 to avoid confusion. The incremental algorithm in [BLSZ14] starts with an empty set R. Thus, in the implementation of the decremental all vertices of R_0 must be inserted during the preprocessing step to compute an initial approximate matching. We will fix the order in which the vertices of R_0 are inserted as follows. First all the vertices that are endpoints of an edge in B are inserted and afterwards all remaining vertices of R_0 are inserted.

Furthermore, the incremental algorithm maintains a value for every left vertex l, called $rank(l)$, which is initially 0. Whenever a vertex is part of an augmenting path, its rank is incremented by 1. After the insertion of a vertex $r \in R$ the algorithm determines a lowest-ranked neighbor l of r and if it is unmatched, matches l with r. In the original algorithm if there are multiple neighbors with identical lowest rank, r is free to choose an arbitrary one. To guarantee condition (C) we modify this choice during preprocessing as follows. Order the edges in B in some arbitrary order and process them one after the other in this order. Let (l, r) be the next edge with $l \in L$ and $r \in R_0$. Insert r into the data structure. Note that the rank of l, and potentially also of the other neighbors of r equals 0. Choose l as the lowest-ranked neighbor of r, match l and r, and increment the rank of l (and of no other vertex) by 1. After all the edges of B have been processed, insert all remaining vertices of R_0 in arbitrary order, matching them as suitable. As the algorithm never unmatches a previously matched edge, all vertices of $B \cap L$ are matched at the end of the preprocessing step, fulfilling condition (C).

5 Dynamic maximum weight basis for laminar matroids

In this section, we describe a dynamic data structure to maintain a maximum weight basis for a laminar matroid. The data structure supports insertions and deletions of elements from the matroid and their weights to the matroid in $O(\log n)$ time, where n is the size of the matroid's ground set.

A laminar matroid $\mathcal{M}$ on a ground set $\mathcal{N} = \{1, 2, \ldots, n\}$ can be described by a tree $\mathcal{T}$ where:

- Each node v in the tree has an associated set $\mathcal{S}_v$ and a "capacity" C_v.

- The root node has the set $\mathcal{N}$ and the n leaf nodes have the singleton sets $\{i\}$ for $i = 1, 2, \ldots, n$. We will slightly abuse notation and refer to a leaf representing the singleton set $\{e\}$ for $e \in \mathcal{N}$ as the leaf node e and consider the ground set $\mathcal{N}$ to be the set of leaves.

- The sets of the children of a node v form a partition of $\mathcal{S}_v$.

A set $\mathcal{S} \subseteq \mathcal{N}$ is independent if $|\mathcal{S} \cap \mathcal{S}_v| \leq C_v$ for every vertex $v \in \mathcal{T}$. In the case of equality, that is when $|\mathcal{S} \cap \mathcal{S}_v| = C_v$, we say that the constraint of v is *tight*. Furthermore for each element $i \in \mathcal{N}$, we assume that there is a weight $w_i \geq 0$. The maximum weight basis of a laminar matroid is the independent set with the largest sum of weights.

5.1 Operations supported by the data structure

Our data structure aims to support the following operations:

- INSERT(u, v, w): Inserts a new element u to existing (non-leaf) v with weight w. Note that this will insert u into all sets $\mathcal{S}_w$ for all vertices w that is an ancestor of u in the tree. If the maximum weight basis changed, report the change.

- DELETE(u): Delete of a leaf u. If the maximum weight basis changed, report the change.

- QUERY(): Query for the maximum weight basis B.

For our applications, we know the underlying matroid structure of all elements that would be inserted into the data structure. Thus we can initialize our tree with all non-leaf nodes and their corresponding capacities.

The data structures we present will maintain an independent set B of maximum weight, and will be *stable*, meaning that the basis B will change by at most one element per insertion or deletion. This is guaranteed by Appendix A.

The main challenge for our data structures is that finding replacement elements when we delete an element involves both path queries to find when tight capacity constraints that loosened, and subtree queries to find the replacement element. Adding and removing elements also changes capacities along a path. Handling both subtree and path queries in tandem is a non-trivial task.

It is easy to devise slow algorithm running in $O(n)$ per operation which we do in Section 5.2. We improve this algorithm with heavy-light decomposition by showing how to change the subtree queries into path queries for $O(\log^2 n)$ per operation in Section 5.3. Finally we use top trees to handle both path and subtree operations and prove the following theorem in Section 5.4.

Theorem 5.1. *There exists a data structure that maintains the maximum weight independent set of a laminar matroid under insertion and deletions of elements in $O(\log n)$ worst-case time per operation.*

A simple lower bound. We briefly remark that this problem is at least as hard as sorting. To sort n elements, one can construct a cardinality matroid (which is a laminar matroid whose tree only consists of the leafs and the root) on the n elements with root cardinality constraint 1. Then sorting can be emulated by using n delete element operations. Since sorting takes $\Omega(n \log n)$, each query must take at least $\Omega(\log n)$ time amortized throughout all operations.

Helpful operations Our data structure we will implement the some additional operations:

- ADD(u): Add the element corresponding to leaf node u to the independent set B. This element must be able to be added to u, i.e. adding it does not violate any laminar constraints.

- REMOVE(u): Remove the element corresponding to leaf node u from B. This element must already be in B.

- LOWESTTIGHTCONSTRAINT(u): For a node u, return the node v closest to u on the path from the root of the tree to u with C_v is exactly equal to the number of elements in B that lie in the subtree rooted at v.

- INITIALIZE(u, v, w): Create a new node u that's a child of v with weight w. This only initializes variables in the new node u and may cause B to no longer be the maximum weight basis.

- DESTROY(u): Destroy a leaf node $u \notin B$.

- QUERYMIN(u): Return the minimum weight element in the subtree rooted at u that lies in the basis B or return that none exist.

- QUERYMAX(u): Return the maximum weight element in the subtree rooted at u not in the basis B that can be added to B without violating any capacity constraint, or return that none exist.

5.2   A slow dynamic laminar matroid data structure

For the sake of exposition, we first describe a slower data structure that has $O(n)$ update time. Throughout all the updates, we maintain an independent set B. Assume that the laminar matroid $\mathcal{M}$ initially starts with no elements (and $B = \emptyset$), and $\mathcal{M}$ constructed be iteratively adding or removing elements.

For each node $v \in \mathcal{T}$ we maintain the following variables:

- $\max_v :=$ maximum weight $e \in \mathcal{S}_v \setminus B$ that can be added to B (i.e. satisfies matroid constraints) as well as a pointer to an element with that weight in $\mathcal{S}_v$. If no elements can be added, set $\max_v = -\infty$.

- $\min_v :=$ minimum weight $e \in B \cap \mathcal{S}_v$ as well as a pointer to an element with that weight in $\mathcal{S}_v$. If $B \cap \mathcal{S}_v = \emptyset$, set $\min_v = \infty$.

- $c_v :=$ the residual capacity of a node v, i.e. $C_v - |B \cap \mathcal{S}_v|$.

For $\min_v$ and $\max_v$, we assume that the elements achieving those values are stored as well (with appropriate sentinel values when they don't exist).

The properties of a matroid guarantees that any addition or deletion of a new matroid element will only change B by at most one element by Appendix A. We'll use the variables above to find the correct element in B to replace. We first describe how to update the variables upon matroid insertions / deletions, and the later describe how to update B under these changes. We first describe the basic operation of updating B.

Implementing QUERYMIN(u) and QUERYMAX(u) These values are maintained by $\min_u$ and $\max_u$.

Implementing LOWESTTIGHTCONSTRAINT(u) We can walk up the tree from u to find the lowest constraint that is tight.

Implementing REMOVE(u) / ADD(u) We assume that B stays an independent set after removing / adding the element u. To remove / insert an element u from B, we simply increment / decrement C_v by 1 for every node v on the path P from u to the root and update the min and max values bottom up on P as follows: Let x be the current node on P. If $C_x > 0$, $\max_x$ can be computed by taking the maximum of $\max_y$ and $\max_z$ of the values of the children y and z of x. Otherwise set $\max_x = -\infty$; $\min_x$ can be computed by simply taking the minimum of a node's children. Then the step is repeated with x being set to the parent of x until the root has been processed.

Implementing INITIALIZE(u, v, w) We create a new leaf node v as a child of u. We set $\max_v$ to w, $\min_v$ to ∞, and $C_v = 1$. This returns u.

Implementing DESTROY(u) We simply delete the node u.

Implementing INSERT(u, v, w) We begin by creating a new leaf node u with INITIALIZE(u, v, w). We check whether $C_x > 0$ for all nodes x on the path P from v to the root. If so, we need to add v to B. If, however, C_x is not positive for all nodes x on P, there is at least one node w on P for which $C_w = 0$. Thus we need to determine whether the maximum weight basis needs to change, i.e. whether there exists an element v' in B that needs to be swapped out for v. To find the element v swaps out with, we begin with finding $y = \text{LOWESTTIGHTCONSTRAINT}(v)$. If $\min_y < w$, then we (1) remove the element e corresponding to $\min_y$ from B by calling REMOVE(e). (2) call ADD(v) to add v to the tree. Note that after (1) it is guaranteed that every node x on P has a positive c_x and, thus, adding e to B and decrementing the c_x values on P accordingly will not create an nodes x with negative c_x value.

Implementing DELETE(u) If the element u is in B, we call REMOVE(u). Now we may delete u from the tree with DELETE(u). Finally, we need to find an additional element to replace u and restore B to a maximum weight basis. We first find $v = \text{LOWESTTIGHTCONSTRAINT}(u)$. Now we remove u completely from the tree by deleting the leaf node u. Finally we may add the element given by $z = \text{QUERYMAX}(v)$ if it exists, and adding that element to B with ADD(z).

Remark In order to maintain the maximum weight base upon INSERT(u, v, w) and DELETE(u) depend entirely on other functions. For our later data structures for this we will omit implementing these functions.

Running time analysis The time to execute an element insertion or deletion is $O(h)$, where h is the height of the laminar matroid structure. One issue with this naive data structure is that the laminar matroid structure could be highly unbalanced, i.e. $O(n)$ in height. When this is the case, each update described above may take $O(n)$ time to execute.

5.3 Accelerating laminar matroid queries through a heavy light decomposition

To achieve $O(\log^2 n)$ query time, we will use the heavy-light decomposition of a tree defined by Sleater and Tarjan [ST83]. Define the size(v) for a node v in the tree to be $|\mathcal{S}_v|$. For a node u with children v and w, we call the tree edge from u to v is *heavy* if size(v) > size(w) or if size(v) = size(w) and $v < w$. Otherwise, we call the edge *light*. We will also refer to v as a heavy (light) child of u if the edge from v to its parent is heavy (light). Note that the binarization procedure from the previous sections guarantee that any non-leaf node has exactly two children with one being heavy and the other being light.

Lemma 5.2. *[ST83] For any vertex v in tree, there is at most one heavy edge to a child of v, and there are $O(\log n)$ light edges on the path from v to the root.*

For each node $v \in \mathcal{T}$ we maintain c_v as before, but instead of storing $\max_v$, and $\min_v$, we store:

- $\text{maxlight}_v := \max_u$ where u is the light child of v (or $w(v)$ if v is a leaf and $v \notin B$).

- $\text{minlight}_v := \min_u$ where u is the light child of v (or $w(v)$ if v is a leaf and $v \in B$).

For every heavy chain H in the tree, we store these values in an auxiliary balanced binary tree that supports the following operations in $O(\log n)$ time:

1. Range increments and decrements for c_v for $v \in H$ (i.e. given two nodes $u, v \in H$, increment or decrement all c_v on the path between those nodes by 1).

2. Reporting the highest / lowest depth node $v \in H$ with $c_v = 0$.

3. Updating the value of maxlight_v or minlight_v for a single $v \in H$.

4. Query for the maximum of maxlight_v or minimum of minlight_v in a contiguous range of the chain.

Note that this allows us to preform the above operations on an arbitrary (not necessarily heavy) path from u to v in $O(\log^2 n)$ time. This can be done by first finding w the lowest common ancestor (LCA) of u and v, then updating the u to w path and the v to w path. By Lemma 5.2, there are at most $O(\log n)$ light edges and $O(\log n)$ heavy paths to update.

Implementing QUERYMIN(u) $\min_u$ can be computed as the minimum value of minlight_y for any node y on the heavy chain that u is on. This is supported by operation 4.

Implementing QUERYMAX(u) We need to first find on the heavy chain of u the first descendent z with $c_z = 0$, which we can do by operation 2. Then $\max_v$ is the maximum of maxlight_y for y on the path from v to z excluding z. Since the path from v to z lies completely in a heavy chain, we can use operation 4 to do so in $O(\log n)$ time.

Implementing LowestTightConstraint(u) This is operation 2. However since u may not be on the same heavy chain as the root, this operation may take $O(\log^2 n)$ time.

Implementing Remove(u) / **Add**(u) We assume that $\mathcal{B}$ stays an independent set after removing / adding the element u. To remove / insert an element e from $\mathcal{B}$, we simply increment / decrement c_v by 1 for every node from e to the root as range updates (operation 1) and update the minlight$_v$ and maxlight$_v$ values on all higher end points of light edges encountered from the path from u up to the root of the tree (operation 3). This takes $O(\log^2 n)$ time overall.

Implementing Initialize(u, v, w) We create a new leaf node v as a child of u. We set maxlight$_v$ to w, minlight$_v$ to ∞, and $C_v = 1$.

Implementing Destroy(u) We delete the node u. Since we assume $u \notin B$, this changes values of maxlight$_v$ potentially on upper endpoints of light edges on the path from u to the root. We can update these values in $O(\log^2 n)$ time.

5.4   Using top trees for $O(\log n)$ time updates

The issue with the naive approach described above is that updating paths from a leaf to the root may be very costly when the tree is deep. When this happens, many max and min values may change in the tree. To get around this we describe a way to accelerate the query times to $O(\log^2 n)$ in Section 5.3 with heavy-light decomposition. The main bottleneck of the heavy-light approach is that we need to perform path range query and updates to maintain certain auxiliary values that take $O(\log^2 n)$ time to perform. To improve the update and query time to $O(\log n)$ time, we use the top tree interface of Alstrup, Holm, de Lichtenberg, and Thorup [AHdLT05] to avoid needing the path range query operation by implicitly storing the query values. To improve path range updates implicitly necessary for the Add and Remove operations, we us a lazy propagation trick to support such updates quickly.

5.4.1   Rooted top trees

Typically top trees are an interface for unrooted forests [AHdLT05, TW05], but we will only describe top trees for rooted trees to simplify some of the notation. Let T be a rooted tree with root r. A top tree over T is a data structure that represents the tree T by *clusters* that represent both a subtree (connected subgraph) and a path of the original tree. We denote a cluster representing the path from u to v and subtree T' of T by (u, v, T'). All edges from vertices of $T \setminus T'$ to a vertex in T' must be incident to u or v which are *boundary nodes*. We will allow leaf nodes to be special boundary nodes and require that each cluster has two boundary points with one of the boundary points as the descendent of the other. We will maintain for the sake of notation that for a top tree cluster $L = (u, v, T')$ we will always have v an ancestor of u (and thus v is the highest node in $T' \cap T$). The original edges of the T are *base clusters*. Top trees contract pairs of clusters to form new clusters, starting from the base clusters, until only one tree clusters. There are two types of valid contractions that we call *join* operations that create a new cluster, and one operation called *split* that deletes a cluster and returns the two child clusters that the cluster was a join of.

- JoinCompressCluster(L_1, L_2) – Given two clusters $L_1 = (u, v, T_1)$ and $L_2 = (v, w, T_2)$ with v having degree two, combine to form a new cluster $(u, w, T_1 \cup T_2)$.

- JOINRAKECLUSTER(L_1, L_2) – Given two clusters $L_1 = (u, x, T_1)$ and $L_2 = (v, x, T_2)$ with v being a leaf of T, combine to form a new cluster $(u, x, T_1 \cup T_2)$.

- SPLIT(L) – Delete the non-base cluster L and returns the two clusters L_1 and L_2 that L was a join of.

The underlying data structure decides when and what types of join and splits are done to maintain a balanced binary tree. The top tree interface takes in functions to change internal values of clusters when join and splits are performed, and only allows a user to have access to clusters through the functions:

- $expose(u, v)$ – Modifies the internal structure of the tree with joins and splits to return a root cluster having u and v as the boundary vertices.

- $cut(u, v)$ – Remove the edge (u, v) and return two new root clusters. Since the structure of T never changes, we will always call $cut(u, parent(u))$, so we will write $cut(u)$ for brevity.

- $link(u, v)$ – Add the edge (u, v), similarly v will always be the parent of u in T, so we will write $link(u)$ for brevity.

The top tree interface guarantees that at most $O(\log n)$ splits and joins are used in its implementation of these user facing functions. For details on how this is done we refer to the implementation of this top trees interface by Tarjan and Warneck [TW05].

5.4.2 Rooted top trees for laminar matroids

Let T be the tree representation for the laminar family with root r. For each top tree cluster $L = (u, v, T_L)$, we will additionally store the following:

- $\text{minc}_L :=$ The minimum c_v value on the path from u to v. Note that this value is only correct if L is an exposed cluster.

- $\Delta_L :=$ The change in c_v values on the path from u to v that needs to be lazily propagate to child clusters of L representing a subpath from u to v.

- $\text{arg minc}_L :=$ The node closest to u on the path from u to v that has $C_w = \text{minc}_L$.

- $\text{maxe}_L :=$ The maximum valued leaf of T' reachable from v that is not a descendent of a node with $x \in T'$ with $c_x = 0$. This value is only stored implicitly and will be determined by the following two variables.

- $\text{maxe1}_L :=$ The maximum valued leaf of T' that is a descendent of a node on the path from u to v and not a descendent of a node $x \in T'$ with $c_x = 0$. Note that this value does not care if a node w from u to v has $c_w = 0$.

- $\text{maxe0}_L :=$ The maximum valued leaf of T' that is a descendent of a node on the path from u to arg minc_L (not including arg minc_L) and not a descendent of a node $x \in T'$ with $c_x = 0$.

- $\text{mine}_L :=$ The minimum valued leaf of T' that is in the independent set B.

Our definitions of maxe0_L and maxe1_L are specifically tailored to maintain the value of maxe_L regardless of whether the value of minc_L has changed due to a lazy update.

$$\text{maxe}_L = \begin{cases} \text{maxe0}_L & \text{if } \text{minc}_L = 0 \\ \text{maxe1}_L & \text{if } \text{minc}_L > 0 \end{cases}$$

We would want that whenever $expose(u, v)$ is called, the value of minc_L is updated correctly accurate and thus we can compute maxe_L. This handles the main issue that when a path update occurs, the value of maxe_L can change for all clusters representing part of the path that gets updated.

INITIALIZATION

For every edge $(u, v) \in T$ with v a parent of u, initialize a base clusters $L = (u, v, \{(u, v)\})$ with the following initial values:

- $\text{minc}_L \leftarrow \min\{c_u, c_v\}$.

- $\Delta_L \leftarrow 0$.

- $\arg\text{minc}_L \leftarrow \arg\min\{c_u, c_v\}$ If there is a tie, set to c_u.

- $\text{maxe1}_L \leftarrow w_u$ if u is a leaf and null otherwise.

- $\text{maxe0}_L \leftarrow w_u$ if u is a leaf and null otherwise.

- $\text{mine}_L \leftarrow u$ if u is a leaf corresponding to an element in B and null otherwise.

JOINCOMPRESSCLUSTER$(L_1 = (u, v, T_1), L_2 = (v, w, T_2))$

Initialize and return a new cluster to $L = (u, w, T_1 \cup T_2)$ with the following values:

- $\text{minc}_L \leftarrow \min\{\text{minc}_{L_1}, \text{minc}_{L_2}\}$.

- $\Delta_L \leftarrow 0$

- $\arg\text{minc}_L \leftarrow \begin{cases} \arg\text{minc}_{L_1} & \text{if } \text{minc}_{L_1} \leq \text{minc}_{L_2} \\ \arg\text{minc}_{L_2} & \text{if } \text{minc}_{L_1} > \text{minc}_{L_2} \end{cases}$

- $\text{maxe0}_L \leftarrow$ maximum valued leaf between maxe0_{L_1} and maxe0_{L_2} if $\text{minc}_{L_1} > \text{minc}_{L_2}$, otherwise maxe0_{L_1}.

- $\text{maxe1}_L \leftarrow$ maximum valued leaf between maxe1_{L_1} and maxe1_{L_2}.

- $\text{mine}_L \leftarrow$ minimum valued leaf in B between mine_{L_1} and mine_{L_2}.

> **JoinRakeCluster($L_1 = (u, x, T_1), L_2 = (v, x, T_2)$)**
>
> Initialize and return a new cluster $L = (u, x, T_1 \cup T_2)$ with the values:
>
> - $\text{minc}_L \leftarrow \text{minc}_{L_1}$.
>
> - $\Delta_L \leftarrow 0$.
>
> - $\text{arg minc}_L \leftarrow \text{arg min}_{L_1}$.
>
> - $\text{maxe0}_L \leftarrow$ maximum valued leaf between maxe0_{L_1} and maxe0_{L_2} if $\text{arg minc}_L \neq x$ otherwise $\text{maxe0}_L \leftarrow \text{maxe0}_{L_1} \leftarrow \infty$.
>
> - $\text{maxe1}_L \leftarrow$ maximum valued leaf between maxe1_{L_1} and maxe1_{L_2}.
>
> - $\text{mine}_L \leftarrow$ minimum valued leaf in B between mine_{L_1} and mine_{L_2}.

> **Split(L)**
>
> Get the two child clusters L_1 and L_2 that L was a join of.
> For $i \in \{1, 2\}$:
>
> - $\delta_{L_i} \leftarrow \Delta_{L_i} + \Delta_L$
>
> - $\text{minc}_{L_i} \leftarrow \text{minc}_{L_i} + \Delta_L$
>
> Delete L and return L_1 and L_2.

We will say a cluster L is *touched* if it was considered in a join or one of the two children of a split. The following lemma is crucial to the correctness of our data structure and so that maxe_L can be computed in $O(1)$ time for any cluster returned from an *expose*, *cut*, or, *link*.

Correctness of the data structure. The correctness of the data structure hinges on the following lemma.

Lemma 5.3. *Whenever a cluster L is considered for an operation, it has the correct value for all values stored in L.*

Proof of Lemma 5.3. Consider the first time t a node L is considered by the algorithm for a split or join since the last time this occurred at the earlier time $t' < t$. We must have propagated any lazy updates that occurred between time t' and t from its parent. Furthermore, there can't have been any changes to descendants of L (with insertions or deletions), since any changes would involve splitting L first. Since that structure beneath L did not change since t', and all of L's ancestors have propagated down their lazy updates, we conclude that the values stored at L are correct. $\square$

We can do our desired updates by first exposing the root and the leaf we are updating. As the implementations of the changes to the auxiliary variables on splits and joins can be done in $O(1)$ time, this results in an $O(\log n)$ time operations for *expose*, *cut*, and *link*.

Implementing QUERYMIN(u) and QUERYMAX(u)   Calling call $cut(u)$ will return a cluster L. By Lemma 5.3, all values stored in L are correct. Since the values are correct, we can compute maxe_L to give the answer for QUERYMAX(u), and mine_L gives QUERYMIN(u).

Implementing LOWESTTIGHTCONSTRAINT(u)   We can begin by letting $L = expose(r, u)$. Now the answer is given by $\arg \text{minc}_L$.

Implementing REMOVE(u) / ADD(u)   When we've chosen an element u that is a leaf of the tree T to remove from / insert into our basis B, we can increment / decrement c_v by 1 for all v on the root to leaf path by calling $expose(r, u)$ and updating Δ_L to update the c_v values lazily. In addition, we need to call $cut(u)$ and $link(u)$ in order to update mine value at u.

Implementing INITIALIZE(u, v, w):   When we insert an element u, we create a new base cluster for the edge from u to its parent v and join it to the tree. The initialization of the edge (u, v) is described earlier.

Implementing DESTROY(u):   We simply need to call $cut(e)$.

6   Transversal matroids

Dynamic $(1 - \varepsilon, 1/2)$-approximate maximum weight oracle.   For every node $\ell \in L$ we will also store a *virtual weight* which we denote by $vw(\ell)$. Initially $vw(\ell) = w(\ell)$. Every time ℓ becomes matched, we will decrease the virtual weight by a factor of $(1 + \varepsilon)^{-1}$. Once its virtual weight is below 1, it will not become unmatched anymore. If a DECREMENT(ℓ, w) operation happens, we will set $vw(\ell)$ to w if it was larger than w, so we maintain that $vw(\ell) \leq w(\ell)$ and unmatch ℓ if it was matched. Observe that the algorithm maintains the following invariant for every vertex ℓ. That the invariant holds can be shown by a simple induction on the number of (re-)match, un-match, and decrement operations of ℓ:

Invariant 6.1. *For every node $\ell \in L$, ℓ is unmatched iff $vw(\ell) = w(\ell)$. Otherwise $vw(\ell) \leq w(\ell)/(1 + \varepsilon)$.*

Furthermore the algorithm will maintain the invariant that every node $r \in R$ is matched to the neighbor with the highest virtual weight. Formally we state this as follows:

Invariant 6.2. *For every node $r \in R$, if r is matched to ℓ with $vw(\ell) = (1+\varepsilon)^j$ for some $j \geq -\lfloor 1/\varepsilon \rfloor$, then for all other $(\ell', r) \in E$, $vw(\ell') \leq (1 + \varepsilon)^{j+1} = vw(\ell)(1 + \varepsilon)$.*

Data structure. We maintain the weight of all the matched vertices of L. Additionally, for each vertex r of R we keep the following data structure:

1. a list N_r of neighbors of r,

2. a pointer p_r into N_r which points to the first element of N_r that has not yet been processed, and

3. a value j_r indicating that r is currently looking for matches with virtual weight at least $(1+\varepsilon)^{j_r}$, which is initialized to $k := \lfloor n/\varepsilon \rfloor$.

The reason for maintaining p_r and j_r is as follows: r greedily tries to match itself with a neighbor of highest virtual weight, i.e. of weight $(1+\varepsilon)^{j_r}$ with $j_r = k$. To do so it traverses its neighbor list to find the first such neighbor. If none exists, it decreases j_r by one and restarts the traversal from the beginning of its adjacency list, i.e. it now checks for a neighbor with second highest virtual weight, and this is repeated until a match is found or j_r becomes negative. The next time that r is looking for a match, r does not have to recheck the neighbors that it already has checked during this traversal as virtual weights only decrease. Thus, it remembers the next neighbor to be checked with the pointer p_r and restarts its traversal from that point. As the traversals stop when j_r drops below a threshold of $-\lfloor 1/\varepsilon \rfloor$, this guarantees that the total work (during the whole algorithm) performed by r while looking for a match is $O(|N_r|(k + 1/\varepsilon))$.

We also assume that each vertex has a non-negative identifier and, thus, the condition that no suitable neighbor has been found so far is checked by testing whether the variable $unmatchedR$, which is initialized to -1, is still negative. If no match is found after j_r drops below the threshold, then we match r to any unmatched neighbor (which must have weight 0).

We initialize our matching with the following static algorithm, FINDMATCHING, that guarantees that the invariants hold after the initialization. This is necessary to guarantee the maximality of the matching.

Algorithm 6.1: FINDMATCHING$(G = ((L, R), E)$

$M \leftarrow \emptyset$.
$vw(\ell) \leftarrow w(\ell)$ for all $\ell \in L$.
$j_r \leftarrow k$ for all $r \in R$.
$N_r \leftarrow$ list of all neighbors of r, for all $r \in R$.
$p_r \leftarrow$ first element N_r.
For $r \in R$:

    1. MATCHR(r).

> **MATCHR(r)**
>
> 1. $unmatchedR = $ -1
>
> 2. While $j_r \geq -\lfloor 1/\varepsilon \rfloor$ and $unmatchedR$ is negative:
>
>   (a) While p_r is not NIL and $unmatchedR$ is negative:
>   i. Set ℓ to the element of N_r pointed to by p_r.
>   ii. Set p_r to the next element of N_r.
>   iii. If $vw(\ell) \geq (1 + \varepsilon)^{j_r}$:
>   A. $vw(\ell) \leftarrow vw(\ell)/(1 + \varepsilon)$
>   B. If ℓ was matched to r' in M:
>   - Remove (ℓ, r') from M
>   - Add (ℓ, r) to M
>   - $unmatchedR = r'$
>   C. Else:
>   - Add (ℓ, r) to M
>   - Return
>
>   (b) If p_r is NIL:
>   i. $j_r \leftarrow j_r - 1$
>   ii. Set p_r to the first element of N_r
>
> 3. If r is unmatched, and there is an unmatched neighbor $\ell \in N_r$, match r to ℓ.
>
> 4. If $unmatchedR$ is not -1: MATCHR($unmatchedR$)

Observe that MATCHR(r) will only match r to nodes that have virtual weight at least $(1+\varepsilon)^{-\lfloor 1/\varepsilon \rfloor}$.

When DECREMENT(ℓ, w) is called, we may assume we either can round w to the form $(1 + \varepsilon)^j$ for some integer $-\lfloor 1/\varepsilon \rfloor \leq j \leq k$ or is 0, because the weight would not matter. The DECREMENT operation is implemented as follows.

> **DECREMENT(ℓ, w)**
>
> Let j be the value such that $vw(\ell) = (1 + \varepsilon)^j$
>
> If $w < (1 + \varepsilon)^j$:
>
> - $vw(\ell) \leftarrow w$
>
> - If ℓ is matched to $r \in R$:
>
>   1. Remove the edge (ℓ, r) from M
>   2. MATCHR(r)
>   3. Return all changes to the set of matched nodes of L.

We next show the desired properties of the algorithm:

Lemma 6.1. *The algorithm is L-stable.*

Proof. Note that once a vertex l of L is matched, it remains matched until the next DECREMENT$(l, \cdot)$ operation, as a MATCHR operation does not un-match any vertex of L. It is only during a DECREMENT$(l, \cdot)$ operation that l is un-matched and it might become matched again during the subsequence MATCHR operations. As a DECREMENT operation was executed with parameter l, it follows that l is not frozen and, thus, the un-matching of l does not violate the L-stability of the algorithm. $\square$

Lemma 6.2. *The algorithm maintains a maximal cardinality matching.*

Proof. By our algorithm either every node of r of R will either be matched, or all neighbors are already matched. $\square$

Lemma 6.3. *The algorithm maintains Invariant 6.2.*

Proof. As the virtual weight of nodes never increases and it remains unchanged while it is matched, it suffices to show that the invariant holds for r immediately after running MATCHR(r) before running MATCHR on the next unmatched node. After MATCHR(r) has been executed, suppose r is matched to ℓ with virtual weight $(1 + \varepsilon)^j$.

There are two cases to consider.

Case 1: $j \geq -\lfloor 1/\varepsilon \rfloor$. Then in Line 2.(a)iii. in MATCHR(r) it must have been true that $vw(\ell) \geq (1 + \varepsilon)^{j+1}$. This implies that $j_r = j + 1$. It follows that r has checked at some earlier point in time that no other neighboring nodes have virtual weight $(1 + \varepsilon)^{j_r+1}$ or larger. As virtual weights never increase, this means that no neighboring node with virtual weight $(1 + \varepsilon)^{j_r+1}$ or larger can exist. Thus, all neighbors must have virtual weight at most $(1 + \varepsilon)^{j_r} = (1 + \varepsilon)^{j+1}$.

Case 2: $j < -\lfloor 1/\varepsilon \rfloor$. Then Invariant 6.2 does not apply to r, i.e., it holds trivially.

Since at initialization we call MATCHR(r) on all nodes $r \in R$, the invariant holds for all nodes any time after initialization. $\square$

For the running time analysis, we only need to inspect how much work MATCHR does for any particular $r \in R$. It will scan through incident edges at most $\lfloor 1/\varepsilon \rfloor + 1 + k = O(1/\varepsilon + \log n)$ times. Thus the total work done by the algorithm throughout the algorithm is at most:

$$\sum_{r \in R} \sum_{(\ell, r) \in E} O(1/\varepsilon + \log n) = O(m(1/\varepsilon + \log n))$$

As we actually maintain the matching explicitely the running time bound also gives an upper bound on the total number of changes to the base.

Next we show an approximation ratio of $(1 + \varepsilon)^{-1}$. The approximation algorithm can be analyzed by inspecting the maximum weight basis B^* and the matching M^* that witnesses the basis, and comparing it to the M we maintain. We consider the symmetric difference $M \oplus M^* = (M - M^*) \cup (M^* - M)$, consisting of cycles, even length paths, and odd length paths. It suffices to analyze the worst case ratio of the weight M^* attains versus the weight M attains on each of these cycles and paths.

The easiest case is for the cycles. Since the graph is bipartite, the cycle is even, and both M and M^* must match the same endpoints of the matching. This means both must match the same set of vertices in L and get the same weight.

For paths we will use the following lemma:

Lemma 6.4. *Let P be a path of $M \oplus M^*$ starting at an unmatched (in M) node $\ell^* \in L$ of weight $(1+\varepsilon)^{j^*}$. If P has a node $\ell \in L$ with $vw(\ell) = (1+\varepsilon)^j$ then*

$$w(P) - w(\ell^*) \geq \sum_{i=j}^{j^*-1} (1+\varepsilon)^i = \frac{(1+\varepsilon)^{j^*} - (1+\varepsilon)^j}{\varepsilon}.$$

For even length paths, if M and M^* share the same set of matched vertices of L on the path, they get the same weight. Otherwise, for even length paths P that start at some vertex $\ell^* \in M^*$ with weight $(1+\varepsilon)^{j^*}$ and end at $\ell \in M$ with weight $(1+\varepsilon)^j$, we can compute the ratio $w(P \cap M^*)/w(P \cap M)$ by applying Lemma 6.4.

$$\frac{w(P \cap M^*)}{w(P \cap M)} = \frac{w(P) - w(\ell)}{w(P) - w(\ell^*)} = 1 + \frac{w(\ell^*) - w(\ell)}{w(P) - w(\ell^*)} \leq 1 + \varepsilon \frac{(1+\varepsilon)^{j^*} - (1+\varepsilon)^j}{(1+\varepsilon)^{j^*} - (1+\varepsilon)^j} = 1 + \varepsilon$$

For an odd length path P, by the optimality of M^*, the path must start at an unmatched (in M) node $\ell^* \in L$ with weight $(1 + \varepsilon)^{j^*}$ and end at an unmatched (in M) node $r \in R$. This means that M^* must contain an edge (ℓ, r) for some $\ell \in L$. Since r is unmatched in M, we must have $vw(\ell) < (1+\varepsilon)^{-\lfloor 1/\varepsilon \rfloor}$, i.e. $j < -\lfloor 1/\varepsilon \rfloor$. As ℓ^* has weight $(1+\varepsilon)^{j^*}$, it holds that $j^* \geq 0$, which implies $j^* - j \geq 1/\varepsilon$. We can apply Lemma 6.4 to P and get:

$$\frac{w(P \cap M^*)}{w(P \cap M)} = \frac{w(P)}{w(P) - w(\ell^*)} = 1 + \frac{w(\ell^*)}{w(P) - w(\ell^*)} \leq 1 + \frac{\varepsilon(1+\varepsilon)^{j^*-j}}{(1+\varepsilon)^{j^*-j} - 1}$$

Claim 6.1 below shows that $1 + \frac{\varepsilon(1+\varepsilon)^{j^*-j}}{(1+\varepsilon)^{j^*-j}-1} \leq 1 + \varepsilon + \frac{1}{j^*-j} \leq 1 + O(\varepsilon)$, where the last inequality follow from the fact that $j^* - j = \Omega(1/\varepsilon)$

Claim 6.1. *For all integer $k > 0$ and any $\varepsilon \geq 0$ it holds that $\frac{\varepsilon(1+\varepsilon)^k}{(1+\varepsilon)^k-1} \leq \varepsilon + \frac{1}{k}$*

Proof. To show the claim it suffices to show that $\varepsilon(1+\varepsilon)^k \leq (\varepsilon + \frac{1}{k})((1+\varepsilon)^k - 1)$, or equivalently that $\varepsilon(1+\varepsilon)^k + (\varepsilon + \frac{1}{k}) \leq (\varepsilon + \frac{1}{k})(1+\varepsilon)^k$, i.e., $\varepsilon + \frac{1}{k} \leq \frac{1}{k}(1+\varepsilon)^k$.

Note that this inequality holds with equality for $\varepsilon = 0$. To finish the proof we differentiate both sides by ε and show that the right side grows as least as fast with ε as the left side. This implies that the inequality holds for all $\varepsilon \geq 0$. Differentiating the left side by ε gives 1, while differentiating the right side gives $(1 + \varepsilon)^{k-1}$, which is at least 1 as $k \geq 1$. Thus the claim follows. $\square$

Proof of Lemma 6.4. Let $\ell_0, r_1, \ell_1, r_2 ..., \ell_s, r_s$ if P is odd or $\ell_0, r_1, \ell_1, r_2 ..., \ell_s$ if P is even be the nodes of P in order of the path with $\ell_0 = \ell^*$. Consider the node $\ell = \ell_{s-1}$ and let j be such that $vw(\ell) = (1+\varepsilon)^j$. Since ℓ_0 is unmatched in M, we know that $vw(\ell_0) = w(\ell_0) = (1+\varepsilon)^{j^*}$. By invariant 6.2 for every i with $1 \geq i \geq s$ on P it holds that $vw(\ell_i) \geq vw(\ell_{i-1})/(1+\varepsilon)$. Thus by induction it follows that that $vw(\ell_i) \geq (1+\varepsilon)^{j^*-i}$. Since $vw(\ell_{s-1}) = (1+\varepsilon)^j$, we must have $s \geq j^* - j$. Thus:

$$w(P) - w(\ell^*) \geq \sum_{i=1}^{s} w(\ell_i) \geq \sum_{i=1}^{s} vw(\ell_i) \geq \sum_{i=j}^{j^*-1} (1+\varepsilon)^i$$

$\square$

Appendix

A Insertions and deletions change at most one maximum weight basis element

In this section, we show that insertions and deletions of weighted elements to a matroid changes the maximum weight basis by at most one element. This lemma is folklore, but we provide a proof here for completeness.

Let $\mathcal{M} = (E, \mathcal{I})$ be a matroid where every element $e \in E$ has a non-negative weight $w(e)$. For sets $S \subseteq E$, we denote the sum of weights in the set by $w(S)$. Let B be the maximum weight basis in $\mathcal{M}$. An insertion (or deletion) is from $\mathcal{M}$ involves adding (or removing) an element from E, and adding to (removing from) $\mathcal{I}$ independent sets containing the newly inserted (deleted) element.

Lemma A.1. *Insertion / deletion of an element e changes the maximum-weight basis by at most one element: Either it stays the same, or an element gets replaced by e (in the case of insertion), or e gets replaced by another element (in the case of deletion).*

Proof. Without loss of generality, assume all elements in $\mathcal{M}$ and the newly inserted element have distinct weights. This guarantees that the maximum-weight basis is unique. If some weights are equal, we can break ties arbitrarily but consistently.[4]

Let B' be the maximum weight basis after e is inserted (deleted). If $B = B'$ we are done, so assume this is not the case. We separate the proof into two cases.

Inserting e into $\mathcal{M}$: e must be in $B' \setminus B$, otherwise we have $B' = B$ (since the maximum-weight basis is unique). Suppose for the sake of contradiction that there exists $b' \in B' \setminus B$ such that $b' \neq e$. Hence, b' was in the matroid prior to the insertion of e.

By the strong exchange property, there exists $b \in B \setminus B'$ (which was also in the matroid originally) such that $B - b + b'$ and $B' - b' + b$ are both bases. Since the matroid elements have distinct weights, we must have either $w(B) < w(B - b + b')$ or $w(B') < w(B' - b' + b)$. In either case, we get a contradiction: Either B was not a maximum-weight basis originally, or B' is not a maximum-weight basis after the insertion of e.

Deleting e from $\mathcal{M}$: If $e \notin B$, then clearly $B' = B$, so assume that $e \in B \setminus B'$. By the strong exchange property, there is $e' \in B' \setminus B$ such that $B - e + e'$ and $B' + e - e'$ are both bases. We have $w(B - e + e') + w(B' + e - e') = w(B) + w(B')$, so it must be the case that either $w(B' + e - e') \geq w(B)$ or $w(B - e + e') \geq w(B')$. Unless $B' = B - e + e'$, whichever inequality holds must be strict, by the distinctness of weights. So either B was not a maximum-weight basis originally (since $B' + e - e'$ is a valid basis before deleting e) or B' is not a maximum-weight basis after the deletion (since $B - e + e'$ is a valid basis after the deletion). Hence it must be the case that $B' = B - e + e'$. $\square$

B Rounding the fractional solutions

In this section, we describe procedures to round the fractional solution output by CONTINUOUS-GREEDY into an integral one. The output of CONTINUOUSGREEDY is a convex combination of

---

[4]For example, by associating the elements $e \in \mathcal{M}$ with a unique index $i_e \in \mathbb{Z}^+$ and adding ε^{i_e} to each elements weight for some small enough ε.

$O(1/\varepsilon)$ bases of the matroid. To round the fractional solution into an integral one, we use the SwapRounding algorithm of Chekuri, Vondrák and Zenklusen [CVZ10]. This algorithm can be implemented in $O(b \cdot \alpha(\mathcal{M}))$ time, where b is the number of bases in the fractional solution, and $\alpha(\mathcal{M})$ is the time it takes to run the MergeBases subroutine (see Algorithm B.1). Given two bases B_1 and B_2, the MergeBases subroutine creates a new base B_{12} from the elements in $B_1 \cup B_2$.

Algorithm B.1: MergeBases($\alpha_1, B_1, \alpha_2, B_2$)

$\mathcal{S} \leftarrow B_1 \setminus B_2$
For each element i in $\mathcal{S}$:

- Find $j \in B_2 \setminus B_1$ such that $B_1 - i + j$ and $B_2 - j + i$ are both bases.

- With probability $\alpha_2/(\alpha_1 + \alpha_2)$, set $B_1 \leftarrow B_1 - i + j$. Otherwise, set $B_2 \leftarrow B_2 - j + i$.

Return B_1. (At the end of this routine, $B_1 = B_2$.)

For ContinuousGreedy, $b = O(1/\varepsilon)$, so it does not contribute significantly to the running time. Our goal in this section is to show that that there exists data structures for which $\alpha(\mathcal{M})$ is small enough for the rounding phase to not be a bottleneck in the overall algorithm.

Rounding with a transversal matroid constraint For a transversal matroid, our prior algorithms provides a matching certifying the bases provided by ContinuousGreedy. Given these matchings, $\alpha(\mathcal{M})$ can be implemented in $O(\text{rank}^2(\mathcal{M}))$ time by doing the following:

1. Given the matchings M_1 and M_2 associated with B_1 and B_2 respectively, form the graph with the edges $M_1 \cup M_2$. This graph consists paths and even length cycles on $O(\text{rank}(\mathcal{M}))$ edges.

2. Find any path from a vertex in $B_1 \setminus B_2$ to $B_2 \setminus B_1$. One is guaranteed to exist due to the matroid basis exchange property.

3. Update B_1 or B_2 respectively according to the MergeBases.

Each augmenting path takes $O(\text{rank}(\mathcal{M}))$ time to find as there are $O(\text{rank}(\mathcal{M}))$ edges. This is repeated up to $O(\text{rank}(\mathcal{M}))$ times, resulting in MergeBases running in $O(\text{rank}^2(\mathcal{M}))$ time.

Rounding with a laminar matroid constraint Using the data structures developed in Section 5, we can implement the choosing of i and j in MergeBases in $O(\log n)$ time by doing the following:

1. Initialize two copies of the laminar matroid data structure $\mathcal{D}_1$ and $\mathcal{D}_2$ with the laminar sets of $\mathcal{M}$ of Section 5, and call Insert on all elements of $B_1 \cup B_2$ into with weight 1 (we will use both as unweighted data structures). In $\mathcal{D}_1$, for every $e \in B_1$ call Add(e) to update constraints, and also call $cut(e)$ if $e \in B_1 \cap B_2$. In $\mathcal{D}_2$, do the same but for elements $e' \in B_2$.

2. Iterate through the elements of $B_1 \setminus B_2$ one by one, as in the MergeBases algorithm. Let the leaf u_i correspond to $i \in B_1$ and $i \notin B_2$ that we are considering in the loop of of the algorithm. Let $v = $ LowestTightConstraint(u_i) in $\mathcal{D}_2$, or the root node if there are no tight constraints.

Call REMOVE(u_i) in $\mathcal{D}_1$ and $cut(u_i)$. Let j be the matroid element corresponding to leaf $u_j = \text{QUERYMAX}(v)$ in $\mathcal{D}_1$ which guarantees that if j exists $j \in B_2 \setminus B_1$ as we only have elements of $B_1 \cup B_2$ inserted in the data structure and QUERYMAX guarantees j is not in B_1. We guarantee that $i \neq j$ since we cut off u_i. This element is guaranteed to exist by the matroid basis exchange property.

$B_1 - i + j$ is an independent set since calling REMOVE(i) in $\mathcal{D}_1$ means no nodes on the path from u_i to the root is tight, and QUERYMAX guarantees the path from v to u_j is not tight. $B_2 - j + i$ is an independent set since calling REMOVE(j) from $\mathcal{D}_2$ would make the path from v to the root have no tight nodes, and since v was the lowest tight constraint of u_i, this allows us to add i to $B_2 - j$.

3. After doing the possible swap (adding u_i back in if we don't swap), we can call $cut(u_i)$ and $cut(u_j)$ to remove i and j from consideration, without changing the c_v values of the intermediate nodes. i and j will still be in their respective (possibly swapped) basis. By the matroid basis exchange theorem, they never need to be considered in the future.

Rounding with a graphic matroid constraint   For a graphic matroid constraint, Ene and Nguyen [EN19] showed that the MERGEBASES can be implemented in $O(r \log^2 r)$ time with a red-black tree where $r = \text{rank}(\mathcal{M})$.

C $B \setminus S$ sampling data structure

Here we describe a data structure $\mathcal{S}$ used for sampling that we can augment any data structure that maintains an independent set I of a matroid when the weights lie in k distinct buckets $\mathcal{B}^{(j)}$ for $j = 1, ..., k$. For each bucket we will store a list L_j of the elements in that bucket. For each list L_j, we will also store a counter $|L_j|$ to keep track of the number of elements in the list. In addition we will also keep track of $w(B \setminus S)$.

This data structure supports the following operations:

- ADD(e, j) - Add e into L_j. It is guaranteed that e is not in some list L_i before this is called.

- REMOVE(e) - Remove e from the list L_{j_e} it is currently stored in.

- DECREMENT(e, j) - Move e from the current list that it is in to L_j. e must be already be in some list L_i for $i > j$.

- SAMPLE(t) - Return a subset of $B \setminus S$ with each $e \in B \setminus S$ sampled with probability $p_e = \min(1, \frac{t \cdot w_e}{w(B \setminus S)})$.

- UNIFORMSAMPLE() - Return a uniformly random element of $B \setminus S$.

Implementation of the data structure   The operations ADD, REMOVE, and DECREMENT, only manipulate the lists, and can be done in $O(1)$ time. To implement SAMPLE(), for each list L_j, we need to sample each element with probability $p_j = \min(1, \frac{t \cdot w_e}{w(B \setminus S)})$ for any $e \in L_j$. If $p_j < 1$, we first sample t_j from the binomial distribution $B(|L_j|, p_j)$, then sample t_j elements from L_j without replacement. Otherwise, when $p_j = 1$, we simply take all $t_j = |L_j|$ elements. This whole operation can be done in $O(k + \sum_j t_j)$ time. Implementing UNIFORMSAMPLE() can easily be done in $O(1)$.

References

[AHdLT05] 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.

[AW14] 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.

[BFNS14] 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.

[BFS17] 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.

[BHK08] Moshe Babaioff, Jason Hartline, and Robert Kleinberg. Selling banner ads: Online algorithms with buyback. In *Fourth workshop on ad auctions*, 2008.

[BIK07] 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.

[Bil22] Jeff A. Bilmes. Submodularity in machine learning and artificial intelligence. *CoRR*, abs/2202.00132, 2022.

[BLSZ14] 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.

[BMKK14] 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.

[BV14] 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.

[CCPV11] 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.

[CVZ10] Chandra Chekuri, Jan Vondrák, and Rico Zenklusen. Dependent randomized rounding via exchange properties of combinatorial structures. In *51th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2010, October 23-26, 2010, Las Vegas, Nevada, USA*, pages 575–584. IEEE Computer Society, 2010.

[EN19] 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.

[Fei98] Uriel Feige. A threshold of ln n for approximating set cover. *J. ACM*, 45(4):634–652, 1998.

[FT87] Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34(3):596–615, 1987.

[GP13] 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.

[HKNS15] 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.

[HPS21] 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.

[KMZ+19] 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.

[LFKK22] Wenxin Li, Moran Feldman, Ehsan Kazemi, and Amin Karbasi. Submodular maximization in clean linear time. *CoRR*, abs/2006.09327, 2022.

[LMSW22] 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.

[LV19] 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.

[NW78] 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.

[Sch03] Alexander Schrijver. *Combinatorial optimization: polyhedra and efficiency*, volume 24. Springer, 2003.

[ST83] Daniel Dominic Sleator and Robert Endre Tarjan. A data structure for dynamic trees. *J. Comput. Syst. Sci.*, 26(3):362–391, 1983.

[Tar75] Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *J. ACM*, 22(2):215–225, 1975.

[TW05] Robert Endre Tarjan and Renato Fonseca F. Werneck. Self-adjusting top trees. In *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2005, Vancouver, British Columbia, Canada, January 23-25, 2005*, pages 813–822. SIAM, 2005.

[WRB+18] 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.

[XR15] 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.

[ZH23] Da Wei Zheng and Monika Henzinger. Multiplicative auction algorithm for approximate maximum weight bipartite matching. *CoRR*, abs/2301.09217, 2023.