Near-Linear Time Approximation Schemes for some Implicit Fractional Packing Problems*

Chandra Chekuri Kent Quanrud

Abstract

We consider several *implicit* fractional packing problems and obtain faster implementations of approximation schemes based on multiplicative-weight updates. This leads to new algorithms with near-linear running times for some fundamental problems in combinatorial optimization. We highlight two concrete applications. The first is to find the maximum fractional packing of spanning trees in a capacitated graph; we obtain a $(1-\epsilon)$-approximation in $\tilde{O}(m/\epsilon^2)$ time, where m is the number of edges in the graph. Second, we consider the LP relaxation of the weighted unsplittable flow problem on a path and obtain a $(1-\epsilon)$-approximation in $\tilde{O}(n/\epsilon^2)$ time, where n is the number of demands.

1 Introduction

Packing, covering, and mixed packing and covering problems are ubiquitous in combinatorial and discrete optimization with several important applications. We are interested in solving *fractional* problems that arise directly or indirectly via linear-programming relaxations of the underlying discrete formulations. A pure packing problem is the form $\max\{\langle v, x\rangle : Ax \leq \mathbb{1}, x \geq \mathbb{0}\}$, where $x \in \mathbb{R}^n$, and $v \in \mathbb{R}^n_{\geq 0}$ and $A \in \mathbb{R}^{m\times n}_{\geq 0}$ are non-negative. A pure covering problem is of the form $\min\{\langle v, x\rangle : Bx \geq \mathbb{1}, x \geq \mathbb{0}\}$, where $v \in \mathbb{R}^n_{\geq 0}$ and $B \in \mathbb{R}^{m\times n}_{\geq 0}$ are non-negative. A mixed packing and covering problem is of the form $Ax \leq \mathbb{1}, Bx \geq \mathbb{1}, x \geq \mathbb{0}$ where both A and B are non-negative matrices. There is a large literature on *fast approximation* algorithms for solving fractional packing and covering problems starting with the influential works of Shahrokhi and Matula [32], Klein et al. [23], Plotkin, Shmoys, and Tardos [30], Grigoriadis and Khachiyan [18], Young [37], Garg and Könemann [16] and many others including recent advances [2]. These algorithms output a $(1\pm\epsilon)$-approximation with running times significantly faster than the exact algorithms obtained by linear programming. The earlier algorithms are largely based on Lagrangian relaxations with exponential penalty functions that can be interpreted in the multiplicative-weight update framework. They result in running times of the form $O(\text{poly}(n)/\epsilon^2)$, where n is the input size. The $1/\epsilon^2$ dependence is necessary for algorithms within this framework [22]. Methods that depend on accelerated gradient techniques, such as those of Nesterov [28], can yield a $1/\epsilon$ dependence. Until recently, those methods had a worse dependence on other parameters of the problem.

We distinguish *explicit* problems where v and A are given explicitly as part of the input from *implicit* problems where v and A are defined implicitly by combinatorial objects such as graphs, set systems, and geometric objects. In an explicit problem, the primary parameters are the dimension n, the number of constraints m, and the number of nonzeroes N in the matrix A. For explicit packing and covering problems, Koufogiannakis and Young [24] showed that one can obtain a $(1\pm\epsilon)$-approximation in $O(N + (m+n)\log N/\epsilon^2)$ time. Young [38] obtained a running time of $\tilde{O}(N/\epsilon^2)$ for mixed packing and covering problems (we use $\tilde{O}$ to hide terms that are poly-logarithmic in the input size and $1/\epsilon$ throughout this paper). In a recent breakthrough, Allen-Zhu and Orecchia [2] obtained an $\tilde{O}(N/\epsilon)$ randomized running-time for pure packing problems based on several new ideas. The same paper obtained a runtime of $\tilde{O}(N/\epsilon^{1.5})$ for pure covering problems, which was reduced to $\tilde{O}(N/\epsilon)$ in subsequent work by Wang et al. [35].

The focus in this paper is on *implicit* problems where the matrix A is not specified explicitly. Multicommodity flow problems are a classic example. The path formulation of flows, which has a variable for every routable path and a constraint for each capacitated edge, is defined implicitly by the underlying graph and demand pairs. For most graphs the dimension is exponential in the input size. Fast approximations for the path formulation via multiplicative-weight updates have been studied extensively in the literature [32, 23, 30, 16, 26], the latest of which, by Mądry [26], approximates the maximum multicommodity flow in

*Department of Computer Science, University of Illinois, Urbana, IL 61820. {chekuri,quanrud2}@illinois.edu. Work on this paper partially supported by NSF grant CCF-1526799.

time $\tilde{O}(mn/\epsilon^2)$. The dependence on ϵ can be decreased to $1/\epsilon$ at the cost of a higher dependence on the input size [8, 29].

The simplicity of multiplicative weight update algorithms makes them amenable to the use of efficient data structures to speed up the computation in each iteration. Partly motivated by a deterministic weight update technique from Young [38], and the use of dynamic data structures by Mądry [26], we consider some classes of implicit fractional packing problems and obtain significantly faster algorithms than previously known. Although we believe that the framework applies to several other problems including pure covering problems and mixed packing and covering problems, we focus on a few well-known packing problems to illustrate the central ideas and the necessary details.

Packing spanning trees. Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be an undirected graph with non-negative edge capacities $c : \mathcal{E} \rightarrow \mathbb{R}_{\geq 0}$. We consider the problem of computing a maximum fractional packing of spanning trees of $\mathcal{G}$. Letting $\mathcal{T}$ be the family of spanning trees of $\mathcal{G}$, we can express this as maximizing $\sum_{T \in \mathcal{T}} x_T$ subject to $x_T \geq 0$ for all $T \in \mathcal{T}$ and $\sum_{T \ni e} x_T \leq c_e$ for all edges $e \in \mathcal{E}$. By a classical theorem of Tutte [34] and Nash-Williams [27], the fractional spanning tree packing number is equal to the *strength* of $\mathcal{G}$ which is defined as $\min_{E' \subseteq \mathcal{E}} \frac{\bar{c}(E')}{\pi(E')-1}$ where $\pi(E')$ is the number of connected components in $\mathcal{G} - E'$. Network strength has several applications and efficient algorithms for it have been well studied [12, 33, 6, 15]. In this paper, we give the following nearly-linear time approximation scheme.

THEOREM 1.1. *There is a deterministic $\tilde{O}(m/\epsilon^2)$-time alogrithm that gives a $(1-\epsilon)$-approximation for fractional packing of spanning trees in a capacitated undirected graph with m edges (and hence also to network strength). This implies that a $(1-\epsilon)$-approximate packing can be described in $\tilde{O}(m/\epsilon^2)$ space.*

The best exact algorithm for fractionally packing spanning trees comes from Gabow's work [15] via his algorithm for packing arboresences, and runs in $O(n^3 m \log(n^2/m))$ time. As far as we are aware, the best previously known running time for a $(1-\epsilon)$-approximation in the capacitated setting is $\tilde{O}(mn/\epsilon^2)$. For uncapacitated graphs a $(1-\epsilon)$-approximation can be obtained in $\tilde{O}(mc/\epsilon^2)$ where c is the minimum cut value using standard multiplicative weights methods. Easy examples show that, even for a $(1-\epsilon)$-approximation, an explicit listing of the spanning trees in the decomposition requires $\Omega(n^2)$ space for a graph with $O(n)$ edges. Hence an implicit representation is necessary to obtain a near-linear running time. Using cut-preserving sampling techniques [7], we can further improve the dependence of the running time on m while worsening the dependence on ϵ if we only want an estimate of the fractional spanning tree number.

COROLLARY 1.1. *There is a randomized $\tilde{O}(m + n/\epsilon^4)$-time algorithm that outputs a $(1-\epsilon)$-approximation to the fractional packing number (and network strength) with high probability.*

Previously, Karger [20] showed that network strength can be estimated in $\tilde{O}(m + n^{3/2}/\epsilon^4)$-time.

Given an undirected graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, the spanning tree polytope is the convex hull of the chracteristic vectors of the spanning trees of $\mathcal{G}$ in the hypercube $[0,1]^{\mathcal{E}}$. The packing algorithm for spanning tree implies the following approximate separation oracle for the dominant of the spanning tree polytope.

COROLLARY 1.2. *Given a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ with spanning tree polytope $\mathcal{P}(\mathcal{T})$, and a vector $z \in [0,1]^{\mathcal{E}}$ there is an $\tilde{O}(m/\epsilon^2)$-time algorithm that either correctly outputs that $z \notin \mathcal{P}(\mathcal{T})$, or outputs a packing of spanning trees into z of value between $(1-\epsilon)$ and 1.*

Algorithms for decomposing a point in the spanning tree polytope into a convex combination of spanning trees have been used in recent algorithms for approximating TSP in both undirected and directed graphs [4, 17] (and several subsequent papers).

Connections to computing minimum cuts: The first step in the near-linear-time randomized global minimum cut algorithm of Karger [21] is to find a $\frac{1-\epsilon}{2}$-approximate spanning tree packing with $O(\log n)$ trees in the packing. This step is the only randomized aspect in his algorithm. We obtain a deterministic near-linear-time $(1-\epsilon)$-approximate packing, however the number of trees in our decomposition can be large even though the overall representation size is $\tilde{O}(m/\epsilon^2)$. It may be possible to use some data structures to build upon our results to obtain a fast deterministic linear-time algorithm for minimum cuts. We also mention that our result yields a deterministic near-linear time $(2+\epsilon)$-approximation for estimating the minimum cut of a graph. There is already such an algorithm due to Matula [25] which is simpler and has a better running time; however, our algorithm is conceptually very different.

The spanning tree packing problem is a special case of the more general problem of packing bases in a matroid. We obtain the following approximation algorithms for packing bases in uncapacitated and capacitated matroids in the oracle model.

THEOREM 1.2. *Let $\mathcal{M} = (\mathcal{N}, \mathcal{I})$ be a matroid with n elements and rank k, accessed by an independence*

oracle that runs in time Q. There is an algorithm that outputs a $(1-\epsilon)$-approximation for fractionally packing disjoint bases of $\mathcal{M} = (\mathcal{N}, \mathcal{I})$ in time $\tilde{O}(nQ/\epsilon^2)$. In the capacitated setting a $(1-\epsilon)$-approximation for fractionally packing bases can be found in $\tilde{O}(nkQ/\epsilon^2)$ time.

Karger [20] considered $(1-\epsilon)$-approximate packings of bases in matroids. He obtained running times of the form $\tilde{O}((n + k^3/\epsilon^5)Q)$ for estimating the value of the packing.

Packing intervals and paths. Suppose we are given n closed intervals $I_1, \ldots, I_n$ on the real line specified by their endpoints $I_i = [a_i, b_i]$. Each interval has a nonnegative value v_i and a non-negative size d_i. We are also given m points $p_1, \ldots, p_m \in \mathbb{R}$ on the real line, and each point p_j has a capacity $c_j > 0$. The goal is to choose a subset of the intervals of maximum value such that the total size of chosen intervals at any point is at most the capacity of the point. This is equivalent to the well-studied unsplittable flow problem (UFP) on paths. These problems and their variants have been well-studied in a variety of contexts and have numerous applications in resource allocation, optical networks, and routing problems, to name a few [5, 14, 36].

The underlying discrete optimization problem is NP-Hard if the demands d_i are different. However, the special case with unit demands ($d_i = 1$ for all i) can be solved in polynomial time. To see this we can consider the linear relaxation of the natural integer program formulation. The LP is of the form maximize $\langle v, x \rangle$ subject to $Ax \leq c$ and $0 \leq x \leq 1$, where the vector x has a coordinate x_i for each I_i, each row corresponds to a point p_j, and $A_{ij} = d_i$ if $p_j \in I_i$ and 0 otherwise. In the unit demand case, A is totally unimodular and hence the polyhedron is an integer polyhedron. Arkin and Silverberg [3] showed that this special case can be solved in $O\big((m+n)^2 \log(m+n)\big)$ time by a reduction to minimum-cost flow. This gives an $O\big(n^2 \log n\big)$ time algorithm for the problem of finding a maximum-weight subset of intervals that is k colorable, which corresponds to scheduling intervals on multiple machines. Borodin [9, slide 35] explicitly asked whether there is a FPTAS for this problem that runs in near-linear time. The input consists of only $O(m+n)$ numbers while the matrix A can have $\Omega(mn)$ nonzeroes. Existing methods that depend linearly on the number of non-zeroes in A take quadratic time for even a $(1-\epsilon)$ approximation. In contrast, we show the following result answering Borodin's question in the affirmative.

THEOREM 1.3. *There is an $\tilde{O}\big((m+n)/\epsilon^2\big)$-time $(1-\epsilon)$-approximation for the fractional interval packing problem.*

We obtain similar time bounds even when there are additional explicit packing constraints on the intervals of the form $Bx \leq \mathbb{1}$. Our running time in this case also depends near-linearly on the number of non-zeroes in B. This is particularly useful when imposing side constraints which occur frequently in applications. We can substantially improve the running times of several LP based approximation algorithms for interval packing and unsplittable flow problems that have been considered in the literature; we omit these details in this version.

Overview of techniques. Our results are built on three known techniques. At a high-level, we use a generic multiplicative-weight-based methodology that builds upon the non-uniform increments idea of Garg and Könemann [16] which yields a width-independent running time. The MWU framework reduces the original problem to implementing an appropriate oracle for a simpler subproblem, along with computing the weight updates for each constraint. For example, the oracle for packing spanning trees computes the minimum weight spanning tree in a weighted graph. The second and third techniques improve the efficiency of the implementation via appropriate dynamic data structures. The idea of using dynamic data structures to improve the efficiency of MWU based algorithms is well-known and has been used for both implicit and explicit packing problems. For packing spanning trees, we use dynamic MST data structures by Holm et al. [19] which maintain the minimum spanning tree in polylogarithmic time per update. The third and less well-known ingredient is a data structure for maintaining weights that has been used effectively by Young [38]. This data structure maintains the weights of the constraints in the MWU algorithm in a *lazy and approximate* fashion such that the amortized update time is small. In the context of minimum spanning trees, suppose we add a fractional spanning tree to the current solution. The MWU framework calls for the weights of each selected edge to be incremented. Naively, this update takes $O(n)$ time and would be the bottleneck of the algorithm, but can be reduced to $\tilde{O}(1/\epsilon^2)$ total time per edge. To achieve this, we borrow the deterministic update scheme for explicit problems by Young [38] and demonstrate its wider applicability to implicit problems where the dimension and/or the number of non-zeroes can be large. A key to the applicability is that the matrix in many implicit problems is *column-restricted*: all the non-zero values in a column are the same.

In retrospect, our overall scheme is surprisingly simple. It brings together and clarifies the applicability of some known high-level ideas to yield new and interesting results for a variety of basic combinatorial optimization

problems. We believe that our specific results and the scheme will lead to further applications.

Other related work. There is a large amount of literature on Lagrangian relaxation methods for obtaining fast approximation schemes to special cases of linear programming problems. More recent techniques employ a variety of powerful techniques from convex optimization, interior point methods, and data structures yielding surprising and powerful improvements to classical problems such as network flow. It is infeasible to do justice to this literature in this extended abstract. This paper is in the line of work that uses multiplicative weight update style algorithms and speeds up the implementation via data structures and approximate oracles. Two recent examples are the paper of Mądry [26] on speeding up multicommodity flow algorithms via improved shortest path data structures, and Agarwal and Pan [1] on geometric covering problems via range tree data structures. We defer a more detailed description to a longer version of the paper.

Extensions and related problems. Our key observation is that the amortized time to update weights in the MWU framework can be made small in various problems of interest that come from combinatorial optimization. This observation applies not only to packing but also to covering and mixed packing and covering problems. There are some natural implicit covering problems such as fractional arboricity (covering edges of a graph by the smallest number of forests), and covering points by intervals that we believe can be tackled by the same methods. Packing intervals is a simple example where geometric data structures are useful. The techniques should apply to other classes of geometric objects in low-dimensional settings such as boxes, triangles and disks, and to UFP in trees. Finally, there are several other implicit packing and covering problems in combinatorial optimization such as fractional packing of arborescences, solving the Held-Karp TSP relaxation, packing subject to matroid and knapsack constraints, covering cuts by spanning trees, etc., which are amenable to some of the ideas here. We plan to address these in future work. We believe that we can also obtain speedups for maximizing the multilinear relaxation of submodular functions building on [11]. It is an interesting question as to whether the dependence of the running time on ϵ can be improved to $1/\epsilon$ for some of the implicit problems we consider while retaining a near-linear dependence on the input size.

2 MWU framework

We outline the MWU-based algorithmic framework for optimization from Chekuri, Jayram, and Vondrák [11]. The framework in [11] is based on previous ideas and cleanly isolates the necessary features for dealing with more complex objectives such as concave and submodular functions. Although this paper only deals with the simpler setting of linear packing constraints, some of our ideas extend to these more sophisticated settings. That said, consider a pure packing problem of the form

$$\text{maximize } \langle v, x\rangle \text{ where } Ax \leq c \text{ and } x \geq \mathbb{0}. \tag{2.1}$$

Roughly speaking, the MWU framework reduces the packing problem to iteratively solving a simpler optimization problem where there is only one constraint. The single constraint is a weighted sum of the m given constraints, and the subproblems are of the form

$$\max \langle v, y\rangle \text{ where } \langle w, Ay\rangle \leq \langle w, c\rangle \text{ and } y \geq \mathbb{0}. \tag{2.2a}$$

The weights $w \in [1, \infty)^m$, initialized to $1/c$ and only increasing thereafter, can be interpreted as dual or Lagrangian variables.

The subproblem (2.2) maximizes a linear objective with a single packing constraint and no other upper bounds on the variables. This is a fractional knapsack problem and an optimum solution is to choose the best bang-for-buck coordinate

$$j = \arg\max_{\hat{\jmath} \in [n]} \frac{v_{\hat{\jmath}}}{\langle w, Ae_{\hat{\jmath}}\rangle} \quad \text{and set} \quad y = \frac{\langle w, c\rangle}{\langle w, Ae_j\rangle} \cdot e_j.$$

Since we are solving a relaxation of (2.1), it is clear that $\langle v, y\rangle \geq \text{OPT}$, where OPT is the value of an optimum solution to (2.1). The advantage of pure packing (or pure covering or even mixed packing and covering) is that the optimum solution y to the relaxed problem (2.2) can be assumed to have only one non-zero entry, which allows for significantly faster implementations.

The MWU method adds a fraction of the solution to the relaxation (2.2), then uses the solution to update the weights for the next iteration. The process is governed by two input parameters ϵ and η. The overall implementation, given as `mwu-template` and taken from Chekuri et al. [11], yields a width-independent running time by choosing time steps in a careful fashion based on the idea of non-uniform increments, originally due to Garg and Könemann [16]. The algorithm tracks progress by a "time" variable t, which increases from 0 to 1. Time is updated in discrete steps via non-uniform increments. The weight at time t for constraint i is denoted $w_i^{(t)}$.

The following theorem restricts the results of Chekuri et al. [11] to the simpler linear setting.

THEOREM 2.1. *If $\eta = \ln m/\epsilon$ and $\epsilon < 1/2$, the algorithm terminates in $O(m \ln(m)/\epsilon^2)$ iterations and outputs a point x such that $\langle v, x\rangle \geq \text{OPT}$ and $Ax \leq$*

```
mwu-template(v,A,c,ε,η)
  w ← 1/c,  x ← 𝟘,  t ← 0
  while t < 1
    j ← arg max_{ĵ∈[m]} v_ĵ / ⟨w^(t), Ae_ĵ⟩   // find the best bang-for-buck coordinate
    y ← ⟨w^(t), c⟩ / ⟨w^(t), Ae_j⟩ · e_j   // y is an optimum solution to the relaxed problem (2.2)
    δ ← min{ min_i (ε/η) · c_i/⟨e_i, Ay⟩, 1 − t }   // such that δAy ≤ (ε/η)c
    x ← x + δy   // add δy to the running solution
    for all i ∈ [m]   // update weights
      w_i^(t+δ) ← w^(t) exp(δη⟨e_i, Ay⟩/c_i)
    t ← t + δ
  end while
  return x
```

$(1 + O(\epsilon))\mathbb{1}$. *The point* $x' = x/(1 + O(\epsilon))$ *satisfies* $Ax' \le c$ *and* $\langle v, x'\rangle \ge (1 - O(\epsilon))\,\mathrm{OPT}$.

The number of iterations depends on m, the number of constraints, and not on n, the dimension of the problem. The proof of the preceding theorem is centered on the sum of weights $\langle w^{(t)}, c\rangle$. By choosing the step size δ to be sufficiently small, we ensure that the sum of weights $\langle w^{(t)}, c\rangle$ grows slowly and in particular that $\langle w^{(t)}, c\rangle \le \langle w^{(0)}, c\rangle \exp((1+\epsilon)\eta t) = m\exp((1+\epsilon)\eta t)$. Hence $\langle w^{(t)}, \mathbb{1}\rangle \le m\exp((1+\epsilon)\eta)$ at the end of the algorithm. Each weight $w_i^{(t)}$ is non-negative, so we also have $w_i^{(t)} \le m\exp((1+\epsilon)\eta)$ for each constraint i. These bounds can then be used to argue that the final solution x does not violate the constraints by more than a multiplicative factor of $(1 + O(\epsilon))$. To bound the number of iterations, we observe that the choice of step size guarantees that the weight of at least one constraint increases by a multiplicative factor of $\exp(\epsilon)$. Each constraint can only increase by such a multiplicative factor $O\big(\ln m/\epsilon^2\big)$ times before reaching the upper bound of $m\exp((1+\epsilon)\eta)$, and there are m constraints, so there are at most $O\big(m \ln m/\epsilon^2\big)$ iterations total[1].

Having bounded the total number of iterations, the key per-iteration steps that determine the final running time are (i) finding the best coordinate j, and (ii) updating the weights of all the constraints. The MWU framework is robust enough for us to approximate these steps. It suffices to pick a $(1 + O(\epsilon))$-approximation for the best coordinate j, and to approximate each weight $w_i^{(t)}$ to within a $(1 \pm \epsilon)$-factor of its true value. Taking advantage of this slack leads to running times of the form $\tilde{O}\big(N/\epsilon^2\big)$ [38], where N is the total number of nonzeroes in A.

In our applications, A is implicitly defined by a combinatorial optimization problem, and the dimension and the number of non-zeroes of A may be too large to apply an algorithm with running time $\tilde{O}\big(N/\epsilon^2\big)$ directly. We employ dynamic data structures that take advantage of the combinatorial construction per the needs of the MWU framework.

3 Packing spanning trees and bases of a matroid

We consider the problem of packing bases in a matroid $\mathcal{M}$ defined over a ground set $\mathcal{N}$ with n elements. Let $\mathcal{B}$ be the set of bases of $\mathcal{M}$ and k the rank of $\mathcal{M}$. Each base $b \in \mathcal{B}$ has cardinality k. In the *disjoint base packing* problem, we want to find the largest subcollection $S \subseteq \mathcal{B}$ of pairwise disjoint bases. In the *capacitated base packing problem*, we equip the ground set $\mathcal{N}$ with positive *capacities* $c : \mathcal{N} \to \mathbb{R}_{>0}$, and want to find the largest collection $b_1, \ldots, b_M \in \mathcal{B}$ of bases, possibly repeating, such that each element $e \in \mathcal{N}$ is contained in at most c_e of the bases.

In this section, we consider linear relaxations of packing bases. As a linear program, the capacitated base packing problem can be expressed as

$$\begin{aligned}
&\text{(3.3a)} && \text{maximize} && \langle \mathbb{1}, x\rangle && \\
&\text{(3.3b)} && \text{over} && x : \mathcal{B} \to \mathbb{R} && \\
&\text{(3.3c)} && \text{where} && \sum_{b : e \in b} x_b \le c_e && \text{for all } e \in \mathcal{N}, \\
&\text{(3.3d)} && \text{and} && x_b \ge 0 && \text{for all } b \in \mathcal{B}.
\end{aligned}$$

[1] One can also bound the number of iterations by $O(n \ln(m)/\epsilon^2)$. To this end, observe that for a fixed choice of coordinate $j \in [n]$, the constraints that increase by $\exp(\epsilon)$ are always the same. Therefore, a coordinate $j \in [n]$ can only be chosen $O\big(\ln m/\epsilon^2\big)$ times before its "bottleneck" constraints reach the upper bound of $m\exp((1+\epsilon)\eta)$.

```
capacitated-bases(𝒩,ℬ,c,ϵ,η)
  w ← 𝟙, x ← 𝟘, t ← 0
  while t < 1
    b ← arg min{w̄(b') : b' ∈ ℬ}
    γ ← w̄(𝒩) / w̄(b)
    δ ← ϵ min_{e∈b} c_e / ηγ
    x ← x + δγb
    for all e ∈ b
      w_e ← exp( ϵ min_{e'∈b} c_{e'} / c_e ) w_e
    t ← t + δ
  end while
  return x
```

This is a packing LP where A is a $0,1$ matrix and the number of variables $|\mathcal{B}|$ may be exponential. It is well-known that the above LP can be solved exactly in polynomial time. One way to see this is via the ellipsoid method, by observing that the separation oracle for the dual LP is the problem of finding a minimum weight base in $\mathcal{M}$, which admits a polynomial time algorithm. Strongly polynomial-time combinatorial algorithms are known for both fractional and integer packing of bases as well [31]. However, the running times of these algorithms are rather high.

Our goal is to find a $(1-\epsilon)$-approximation to the fractional base packing problem. The special case of packing spanning trees in a graph is a particular focus because it has several important applications. The capacitated case requires more machinery and outlines the need for a combination of data structures to improve the running time of an MWU-based implementation. We treat it here. Appendix B considers the unit capacity setting.

Packing in the capacitated setting. We adapt the MWU algorithm of Section 2 to the fractional base packing LP (3.3) in the algorithm `capacitated-bases`. There is a variable x_b for each base $b \in \mathcal{B}$. The number of constraints is n, corresponding to the capacities on the elements. The implicit constraint matrix is a $\{0,1\}$-matrix, where the column corresponding to a base b has 1's for each element $e \in b$. The number of nonzeroes in each column is the rank of $\mathcal{M}$, k. Recall that in each iteration the MWU algorithm picks a single variable and takes a small step of that variable solution times δ. This corresponds to picking a base b, and in particular, the minimum-weight base w/r/t the current weights on $\mathcal{N}$. Although the number of variables (bases) may be exponential, the number of constraints is n, so the MWU algorithm terminates in $O(n\ln(n)/\epsilon^2)$ iterations.

We first analyze the naive implementation of the algorithm. The algorithm maintains n weights, one for each constraint (i.e., for each element in $\mathcal{N}$). In each iteration it does following.

1. Compute a minimum weight base b with respect to the weights $w : \mathcal{N} \to [1,\infty)$.
2. Update the current solution by adding some multiple of b.
3. Update the weight of each of the k elements in b.

Updating the weights takes $O(k)$ time. Computing a minimum weight base takes $O(nQ + n\log n)$ time via the greedy algorithm, where Q is the cost of a call to an independence oracle for $\mathcal{M}$. Thus, the overall running time is $O\big((nQ+n\log n)n/\epsilon^2\big) = \tilde{O}\big(n^2Q/\epsilon^2\big)$.

The first observation to improve the running time is to update the weights *lazily*. Suppose w_e is the weight of an element e in the current iteration, and the new weight after the iteration is w'_e. Suppose $w'_e < (1+\epsilon)w_e$ and say we do not update update the weight as we are supposed to. This affects the minimum weight base in the next iteration, but only by a $(1+\epsilon)$-factor, and the resulting approximate minimum-weight base can be absorbed into the overall approximation factor. How does this help us? If we only maintain a weight w_e approximately to within a $(1+\epsilon)$-factor, then the total number of weight changes of e is $O(\ln(n)/\epsilon^2)$. The total number of weight changes of all elements is $O(n\ln(n)/\epsilon^2)$. If we also had a dynamic data structure for maintaining the minimum weight base, then the number of update operations for that data structure is only $O(n\ln(n)/\epsilon^2)$! There are already very efficient data structures for maintaining the minimum spanning tree of a graph [19]. In Appendix A, we develop a data structure for maintaining the minimum weight base in a matroid.

By rounding down each weight w_e to powers of $(1+\epsilon)$, we can limit the number of update requests to the data structure, but we still need to keep track of the true weights w. In the capacitated setting, the weights of the updated elements change at different rates, and visiting each weight requires $O(k)$ time. The overall running time just to update the weights would be $O(nk\ln(n)/\epsilon^2)$; in the case of packing spanning trees in a graph with m edges and n nodes, this is $\tilde{O}\big(mn/\epsilon^2\big)$. To overcome this barrier, we need to have a data structure that maintains the weights in amortized polylogarithmic time per update. We show that this is feasible in several settings when the matrix A has a column-restricted structure, such as the one for packing bases. To make the discussion concrete we focus on the problem of packing spanning trees. The case of general matroids is deferred to Appendix A.

Packing Spanning trees. Let $\mathcal{G} = (\mathcal{V},\mathcal{E})$ be a capacitated graph with m edges and n nodes. Let $c_e > 0$

denote the capacity of an edge e. Let $\mathcal{T}$ be the set of spanning trees of $\mathcal{G}$. The spanning trees $\mathcal{T}$ are the bases of the graphic matroid $\mathcal{M}_{\mathcal{G}}$ and the edges $\mathcal{E}$ form the ground set. Given positive edges weights $w : \mathcal{E} \to \mathbb{R}_{>0}$, computing a minimum weight base $\mathcal{M}_{\mathcal{G}}$ is precisely the task of computing the minimum weight spanning tree of $\mathcal{G}$.

Per the preceding discussion, a fast MWU implementation requires a dynamic data structure to maintain the minimum spanning tree. The data structure of Holm et al. [19] can maintain the MST under edge insertions and deletions in $O(\log^4 n)$ time per update[2]. Recall that the total number of data updates is $O(m \ln m/\epsilon^2)$ if we only update the data structure when the weight of an edges crosses a power of $(1+\epsilon)$. Then the total work over the entire course of the algorithm to find the MST in each iteration is $O(m \log^5 n/\epsilon^2)$.

The only remaining bottleneck to obtain a nearly linear running time is the need to update $n-1$ edge weights at the end of each of $\tilde{O}(m/\epsilon^2)$ iterations. We alleviate this bottleneck by extending the techniques of Young [38]. To motivate the details of the data structure, we first walk through the details of how the weights w change in each iteration.

Suppose in the current iteration we select a tree T. Let c be the minimum capacity of any edge in T. We first add a $(\epsilon c/\eta)$-fraction of T to the current solution. Then the weight of each edge $e \in T$ needs to be increased to $\exp(\epsilon c/c_e)w_e$. In particular, the weight update depends on the ratio of c/c_e, and larger capacity edges growing much more slowly than small capacity edges. As the tree changes in each iteration, the minimum capacity changes and with it so does the *rate* at which each weight is updated. The key observation is the following. Suppose we change from tree T to a new tree T'. The rates of all the edges common to both T and T' change, but uniformly by the same factor. This is precisely due the fact that the nonzero coefficients in each column of the packing matrix A are identical. Of course, in this case it is even simpler because A is a $0,1$ matrix, but column-restrictedness suffices and will be used in the next section. In contrast, if A is arbitrary, then when the selected column changes to j, the rate at which the weight of a constraint for row i changes will also depend on the coefficient A_{ij}.

[2]This bound can be improved slightly because we only increase weights of edges to powers of $(1+\epsilon)$ from 1 to $O(m^{1/\epsilon})$. More precisely, one can apply the decremental data structure of Holm et al. [19] to $O(\log(n)/\epsilon^2)$ copies of each edge. Each copy of an edge has its weight set to a power of $(1+\epsilon)$. When the weight of an edge e (in the MWU framework) increases from $(1+\epsilon)^i$ to $(1+\epsilon)^{i+1}$, we delete the copy of e with weight $(1+\epsilon)^i$ from the data structure. This brings the running time down to $O(\log^2 n)$ per weight update, and $O(m \log^3(n)/\epsilon^2)$ overall.

Our goal is to build a data structure that adaptively maintains the weights of the edges as they grow at different rates. A key feature is that it should enable an efficient way to change the rate of all the edges currently in the data structure by a common scaling factor, without individually examining all of them. The data structure should respond to an increment by outputing all edges whose weights have increased by more than (roughly) a $(1+\epsilon)$-factor. One can do this in amortized poly-logarithmic time per update. We describe below the high-level interface of the data structure and the simple idea behind its implementation, and leave the finer details to the appendix. After that we describe the algorithm for packing spanning trees by combining the use of the two data structures.

A data structure for lazy weight increments. The `lazy-incs` data structure approximates a dynamic set of counters that increment concurrently at different rates[3]. The interface for `lazy-incs` is given on page 8. In the applications of this paper, we use `lazy-incs` to track the "additive part" $v_i = \frac{1}{\epsilon}\ln(w_i)$ for each constraint i. The rate of each counter i is stored as $\texttt{rate}(i)$. The primary operation of `lazy-incs` is $\texttt{inc}(\rho)$, which simulates the increments for one increment at the rate ρ. For each i tracked by the data structure, $\texttt{inc}(\rho)$ (approximately) adds $\texttt{rate}(i)/\rho$ to the counter for i[4]. The salient point of `lazy-incs` is that it makes the increments in an amortized fashion, where the total work is proportional to the sum of all increments. In exchange for better amortized efficiency, the counters are approximated to a constant additive factor of their true values. Note that if we can estimate $v_i = \ln(w_i)/\epsilon$ to an additive factor of $O(1)$, then we can estimate $w_i = \exp(\epsilon v_i)$ to a $(1 \pm O(\epsilon))$-multiplicative factor, which suffices for our applications.

At a high level, `lazy-incs` buckets the counters by rounding up each rate to the nearest power of 2. For each power of 2, an auxiliary counter is maintained (more or less) exactly and efficiently. The auxiliary increments at rates that are powers of 2 can be maintained in constant amortized time for the same reason that a binary number can be incremented in constant amortized time. Whenever an auxiliary counter increases by a whole counter, the tracked counters in the corresponding bucket are increased proportionately. Up to

[3]The data structure we describe is essentially borrowed from the scheme in [38]. We isolate a clean interface so that it can used effectively in a modular fashion. The interval packing application also demonstrates the utility of a unified interface.

[4]Technically, $\texttt{inc}(\rho)$ tries to increase counter i only if $\lceil \log \texttt{rate}(i) \rceil \leq \lceil \log \rho \rceil$. In all applications in this paper, ρ is always greater than or equal to $\texttt{rate}(i)$ for any constraint i being tracked.

Lazy Increments: Interface

Operations	Description
`lazy-incs(`ζ`)`	Initializes the `lazy-incs` data structure with error parameter $\zeta \in (0,1)$.
`insert(`i`,`ρ`)`	Given an integer $i \in \mathbb{N}$ identifying a weight and a positive rate $\rho \in \mathbb{R}_{>0}$, begins tracking increments to the weight i at the rate ρ.
`delete(`i`)`	Given an identifying integer $i \in \mathbb{N}$, stops tracking increases to weight i. Returns a left over increment $\delta \in \mathbb{R}_{\geq 0}$ to be committed to weight i.
`inc(`ρ`)`	Given a value ρ, simulates an increase at the rate of the fastest weight being tracked. Returns a list of couples (i, δ), where $i \in \mathbb{N}$ identifies a weight and $\delta > 0$ is an positive increment to commit to weight i.

accounting details, this tracks the increments to each counter to within a constant additive factor at all times. A detailed implementation of the operations of `lazy-incs` and a full proof of the following theorem are deferred to Section 5.

Theorem 3.1. *Consider an instance of* `lazy-incs(`ζ`)` *over a sequence calls to* `inc`*,* `insert`*, and* `delete`*. Let* $M \in \mathbb{N}$ *be the total number of calls to* `inc`*,* $I \in \mathbb{N}$ *be the number of calls to* `insert`*, and* $D \in \mathbb{N}$ *the total number calls to* `delete`*. For each constraint* i*, let* $\widetilde{v}_i$ *be sum of increments for constraint* i *confirmed in the return values of calls to* `inc`*, and let* D_i *be the number of times constraint is deleted. Then* `lazy-incs(`ζ`)` *maintains* $\widetilde{v}_i$ *to within a* $O(1+D_i\zeta)$ *additive factor of the true number of increments for constraint* i*. The total time over all operations is* $\tilde{O}(M + (I+D)\log(1/\zeta) + \sum_i V_i)$*, where* $\tilde{O}$ *hides logarithmic factors in* $M + I + D$*.*

Putting it all together. With `lazy-incs` in tow, we can assemble the complete algorithm `capacitated-spanning-trees`, given on page 9. Recall that by dynamically maintaining the minimum weight spanning tree with respect to an approximate set of weights that updates infrequently, the only obstacle is updating the weights for all $n-1$ edges of the selected tree in each of $\tilde{O}(m/\epsilon^2)$ iterations. At the beginning of the algorithm, we instantiate an instance of the `lazy-incs(`ζ`)` data structure with $\zeta \approx \epsilon^2$, and call `insert` on each edge of the first minimum spanning tree. The rate for each edge e, per the MWU template, is the reciprocal of its capacity, $1/c_e$. The data structure for the minimum spanning tree and the `lazy-incs` data structure mirror one another. When one edge is replaced by another in the minimum spanning tree, a matching `delete` and `insert` is made to `lazy-incs`. When `lazy-incs` commits a full increment to an edge e, the weight change is propagated to the dynamic minimum spanning tree, which may decide to replace that edge with another. The total number of updates to either data structure is $O(\log m/\epsilon^2)$ per edge. This gives Theorem 1.1.

The running time of Theorem 1.1 is approximately linear in the number of edges despite the fact that the algorithm returns a weighted combination of $\tilde{O}(m/\epsilon^2)$ spanning trees each consisting of $n-1$ edges. This output alone suggests a minimum running time of $\tilde{\Omega}(mn/\epsilon^2)$. `capacitated-spanning-trees` beats this lower bound by taking advantage of high overlap between one spanning tree and the next. Surprisingly, Theorem 1.1 implies that a $(1-\epsilon)$-approximation to fractionally packing spanning trees can be *described* in $\tilde{O}(m/\epsilon^2)$ bits (times the bit complexity of the edge weights).

4 Packing intervals

We consider the *fractional interval packing* problem defined as follows. Let $\mathcal{I}$ be a set of n intervals on the real line, where each interval $I \in \mathcal{I}$ is defined by its two endpoints $I = [\alpha, \beta]$. Let $\mathcal{P}$ be a set of m points on the real line. Each interval $I \in \mathcal{I}$ has a positive value $v_I > 0$ and a positive size $d_I > 0$. Each point $p \in \mathcal{P}$ has a positive capacity $c_p > 0$. We want to

$$\begin{aligned} &\text{(4.4a)} \quad \text{maximize} \quad \langle v, x \rangle \\ &\text{(4.4b)} \quad \text{over} \quad x : \mathcal{I} \to \mathbb{R}_{\geq 0} \\ &\text{(4.4c)} \quad \text{subject to} \quad \sum_{I \in \mathcal{I}: p \in I} d_I x_I \leq c_p \quad \text{for all } p \in \mathcal{P}. \end{aligned}$$

Several applications impose the constraint that $x_I \leq 1$ for each I. We discuss this extension later in Section 4.1 when we append an additional set of packing constraints $Bx \leq \mathbb{1}$ to the above LP (4.4).

The interval packing problem is different from the base packing problem (3.3) in that the number of dimensions/variables, n, is linear in the input size. In that sense the problem is explicit. The difficulty lies in the fact that each interval can contain many more points than its two end points. If we take the number of

```
capacitated-spanning-trees(G = (V, E), c, ε, η)
  w̃ ← 𝟙, x ← 𝟘, t ← 0
  T ← dynamic minimum weight spanning tree w/r/t w̃
  I ← lazy-inc(Θ(ε²/ log m))
  for all e ∈ T
    I.insert(e, 1/c_e)
  while t < 1
    γ ← (Σ_{e∈N} w̃_e) / (Σ_{e∈T} w̃_e)
    δ ← (ε min_{e∈T} c_e) / (ηγ)
    x ← x + δγT // add to the running solution
    Δ ← I.inc(1/ min_{e∈T} c_e) // increment all the edges in T
    for each increment (e, ξ) ∈ Δ
      w̃_e ← exp(εξ) w̃_e // apply the increment to w̃_e
      if T replaces e with an edge f
        ξ' ← I.delete(e) // remove e from the lazy-incs data structure
        w̃_e ← exp(εξ') w̃_e // apply any residual increment
        I.insert(f, 1/c_f) // insert f into the lazy-incs data structure
      end if
    end for
    t ← t + δ
  end while
  return x
```

points in each interval, and sum them up, then the total number of point-interval relations, which corresponds to the number of non-zeroes in the packing matrix A, may be $\Omega(mn)$. To obtain a nearly linear running time in the input - which has size $O(m+n)$ - we cannot afford to visit every point for every interval. That is to say that the implicit packing matrix A is too dense for a nearly-linear time algorithm to even write down explicitly.

If we apply the MWU framework to the fractional interval packing problem (4.4), we obtain the algorithm `intervals`. The MWU framework instantiates a weight w_p for each point $p \in \mathcal{P}$. For each interval I, let the weight of I be the sum of weights over the points contained in I, $\overline{w}(I) \stackrel{\text{def}}{=} \sum_{p \in \mathcal{P} \cap I} w_p$. The algorithm repeatedly finds an interval with approximately maximum ratio of value to size-times-weight. After choosing an interval I, it adds a fraction of I to the running solution and then increments the weight of every point $p \in \mathcal{P} \cap I$ in proportion to its capacity.

Per the MWU framework, `intervals` runs in $\tilde{O}(m/\epsilon^2)$ iterations. To keep our running time near-linear, we need to (approximately) compute the weight $\overline{w}(I)$ of an interval I, update the weight w_p of each point $p \in \mathcal{P} \cap I$, and find the best bang-for-buck interval $\arg\max_{I \in \mathcal{I}} v_I / \overline{w}(I)$ efficiently. By "efficiently", we mean that an operation should either run in $\tilde{O}(1)$ worst-case time, or $\tilde{O}((m+n)/\epsilon^2)$ total time over the course of the algorithm, where $\tilde{O}$ hides polylogarithmic factors in m, n, and $1/\epsilon$.

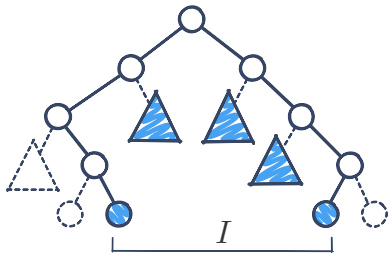
```
intervals(P, c, I, v, ε, η)
  w ← 𝟙, x ← 𝟘, t ← 𝟘
  while t < 1
    choose I ∈ I s.t.
      v_I / (d_I w̄(I)) ≥ (1 − ε) max_{J∈I} v_J / (d_J w̄(J))
    γ ← (Σ_{p∈P} w_p c_p) / (d_I w̄(I))
    δ ← (ε / (ηγ d_I)) · min_{p∈I∩P} c_p
    x ← x + δγI
    for all p ∈ P ∩ I
      w_p ← w_p exp( (ε min_{q∈I∩P} c_q) / c_p )
    t ← t + δ
  end while
  return x
```

We take advantage of the simple geometry of the problem *and* the column-restricted nature of the matrix. We integrate the `lazy-incs` data structure into a range tree over the points to maintain the weights efficiently. We also need another standard bucketing trick to choose the approximately best interval in each iteration.

The augmented range tree data structure. Let T

be a balanced binary tree whose leaves are the points $\mathcal{P}$ in sorted order. T has $O(\log m)$ levels. For each node $\nu \in T$, let T_ν denote the subtree of T rooted at ν and let $\mathcal{P}_\nu$ denote the points at the leaves of T_ν.

An input interval $I = [\alpha, \beta] \in \mathcal{I}$ traces two paths of length $O(\log m)$ from the root of T down to leaves (in the figure on the above, the edges on these paths are drawn solid). The two leaves may or may not be in I. Every other subtree T_ν that hangs off these paths, and lying between the paths, has all its points $\mathcal{P}_\nu$ in the interval I. That is, each interval is the disjoint union of the (leaves of the) maximal subtrees contained in I, and there are $O(\log m)$ such subtrees that can all be retrieved in $O(\log m)$ time (filled in in the picture below). When an interval contains all the points of such a subtree, we hope to perform a single aggregate operation at the root instead of visiting every leaf individually.



At each node $\nu \in T$ we store several quantities of interest. We store the minimum capacity among $\mathcal{P}_\nu$, and maintain a value $\widetilde{W}(\nu)$ that approximates the sum of weights of $\mathcal{P}_\nu$ to within a $(1\pm\epsilon)$-factor. We also store a `lazy-incs` data structure at each ν which is initialized statically with $\mathcal{P}_\nu$; in this application, there is no need to delete a point from the data structure. A point $p \in \mathcal{P}$ is inserted into $O(\log m)$ `lazy-incs` data structures, so we maintain a single weight $\widetilde{w}_p$ for each point $p \in \mathcal{P}$ to aggregate its increments. The exact interaction between $\widetilde{w}$ and the `lazy-incs` data structures requires a bit more care and will be explained later. The total storage for all the data structures is $O((m+n)\log^2(m+n))$ and initialization takes $O((m+n)\log^2(m+n))$ time.

We now describe how each of the crucial steps in the MWU framework can be implemented to obtain the desired nearly-linear running time.

Computing the weight of an interval. The weight of an interval I is the sum of weights of the leaves of the maximal subtrees contained in I. As mentioned above, for each node ν, the data structure maintains a value $\widetilde{W}(\nu)$ that approximates the sum of weights of $\mathcal{P}_\nu$. If $\widetilde{W}(\nu)$ is a $(1\pm O(\epsilon))$-approximation for the (true) sum of weights $\overline{w}(\mathcal{P})\eta$, then the sum of $\widetilde{W}(\nu)$ over all maximal subtrees T_ν contained in I is a $(1\pm O(\epsilon))$-approximation of the true weight of I. If each approximate sum $\widetilde{W}(\nu)$ is already computed, then the weight of I can be approximated in $O(\log m)$ time.

Whenever a weight $\widetilde{w}_p$ is increased by a `lazy-inc` data structure, we visit every subtree T_ν containing p and increase $\widetilde{W}(\nu)$ by the same amount. Since $\widetilde{w}_p$ is a $(1\pm O(\epsilon))$-approximation of w_p, and $\widetilde{W}(\nu)$ is the sum of $\widetilde{w}_p$ over $p \in \mathcal{P}_\nu$, $\widetilde{W}(\nu)$ is a $(1 \pm O(\epsilon))$-approximation of the sum of weights $\overline{w}(\mathcal{P}_\nu)$. Thus, we can charge the total cost of updating the $\widetilde{W}(\nu)$ values to the total number of weight updates of the points. As we have seen previously, the total number of weight updates is $\tilde{O}(m/\epsilon^2)$.

Incrementing the point weights of an interval. In each iteration we add a fraction of the chosen interval I to the running solution and need to update the weights of the points in $I \cap \mathcal{P}$. Recall that for every node $\nu \in T$, we maintain an instance of the `lazy-incs` data structure. At the start of the algorithm we instantiate it with `lazy-incs(0)` and for each point $p \in \mathcal{P}_\nu$ we call `insert(`p`,`$\log m/c_p$`)`. (When we insert before any increments, `insert` takes constant time.)

After adding some a fraction of I to the solution, we update the weights as follows. Let $c = \min_{p\in\mathcal{P}\cap I} c_p$. We visit the root ν of every maximal subtree contained in I and call `inc(`$\log m/c$`)` on its `lazy-incs` instance. For every pair (p, δ) returned in `inc`, where $p \in \mathcal{P}_\nu$ and $\delta > 0$ registers an increment to p, we increase $\widetilde{w}_p$ by a multiplicative factor of $\exp(\epsilon\delta/\log m)$.

The alert reader may wonder why we use a rate of $\log m/c_p$ for point p instead of $1/c_p$ and how it affects the running time. The reason is that each point $p \in \mathcal{P}$ has increments counted by $O(\log m)$ instances of `lazy-incs`. When we increment weights for an interval I containing p, an increment is registered in exactly one of these data structures; namely, that corresponding to the unique maximal subtree of I containing p. Since $O(\log m)$ different `lazy-incs` data structures contribute to the weight of a point p, each data structure needs to commit refined increments at a higher rate to ensure that the total error over all data structures for p does not exceeds an additive $O(1)$ factor. By scaling up the rates by $\log m$ and scaling down the increments by $\log m$, each data structure reports respective increments to p to within an additive factor of $O(1/\log m)$ (instead of $O(1)$). Since there are $O(\log m)$ such data structures, the sum of reported increments is within an $O(1)$ additive factor of the true total. It follows that $\widetilde{w}_p$, which maps the sum of reported increments Δ to $\exp(\epsilon\Delta)$, is within a $(1\pm O(\epsilon))$-multiplicative factor of the true weights.

By increasing the sensitivity of the `lazy-incs` data structures by $\log m$, each "full" increment to $\widetilde{w}$ may

be as small as $1/\log m$. Consequently we examine the weight of each point p an $O(\log m)$ factor more times, increasing the running time by an $O(\log m)$ factor.

Lazily finding the maximum ratio interval. The final issue to be dealt with is approximating best interval in each iteration. Recall that we want to pick the interval J with the maximum ratio $\frac{v_J}{d_J \overline{w}(J)}$. Let α_J denote this ratio for interval J, and note that α_J changes only because of changes to $\overline{w}(J)$. We use a lazy bucketing scheme for this purpose which amortizes the cost of updating α_J. The intervals are placed in buckets. Bucket i has all intervals whose α_J value is between $(1+\epsilon)^i$ and $(1+\epsilon)^{i+1}$. The non-empty buckets are maintained in a linked list. In each iteration, we take an arbitrary interval I from the bucket i with largest index, and recompute the weight of I. If the ratio for I still lies in the range of the bucket, then we take I as an approximately maximum ratio interval. Otherwise, we reassign I to a lower bucket according to the newly computed weight. This way, each computation of the weight of an interval can either be charged to an iteration, or charged to an interval going to a lower-index bucket. As the weight of an interval ranges from 1 to $m^{O(1/\epsilon)}$, each interval only goes through about $\tilde{O}\big(1/\epsilon^2\big)$ buckets. Thus, the total number of times we need to recompute the weight of an interval over the course of the algorithm is $\tilde{O}\big((m+n)/\epsilon^2\big)$, and each such computation takes $O(\log m)$ time.

4.1 Additional packing constraints for intervals

The interval packing problem considered above had constraints only based on the capacities c of the points. The formulation did not allow simple upper bounds on x_I. It is not hard to extend the analysis to handle such simple constraints. Our goal here is to show that one can achieve more, and point out some applications. We consider the interval packing problem where, in addition to the capacity constraints imposed by the points, we have an additional set of packing constraints on the x_I variables in the form of an explicit matrix $Bx \leq \mathbb{1}$. We give one motivating application. In resource allocation problems [5], a job/task $\mathcal{J}$ consists of several intervals, each of which may have a different profit. A job requires only one of its intervals to be done; equivalently, the profit obtained from a task is the maximum of the profits of the intervals in the task that are scheduled. This can be seen as a partition matroid constraint on the set of intervals. A natural way to model this problem is to add, for each job/task $\mathcal{J}$, the constraint $\sum_{I \in \mathcal{J}} x_I \leq 1$. Generalizations have been studied under the label bag-constrained UFP in paths and trees [13, 10]. Scheduling interval jobs on multiple unrelated machines can also be modeled in this fashion.

```
intervals'(P,c,I,d,B,ε,η)
  w ← 𝟙, w' ← 𝟙, x ← 𝟘, t ← 0
  while t < 1
    I ← arg max_{J∈I} v_J / (d_J w̄(J) + ⟨w', BJ⟩)
    γ ← (⟨w, 𝟙⟩ + ⟨w', 𝟙⟩) / (d_I w̄(I) + ⟨w', BI⟩)
    κ ← min{ min_{p∈I∩P} c_p/d_I , min_{i'∈[m]} 1/⟨e_i', BI⟩ }
    δ ← εκ/(ηγ)
    x ← x + δγI
    for all p ∈ P ∩ I
      w_p ← w_p · exp(εκd_I / c_p)
    for all i' ∈ [m']
      w'_{i'} ← w'_{i'} · exp(εκ⟨e_{i'}, BI⟩)
    t ← t + δ
  end while
  return x
```

Formally, the problem we have is of the form

$$\text{maximize } \langle v, x\rangle \tag{4.5a}$$
$$\text{over } x : \mathcal{I} \to \mathbb{R}_{\geq 0} \tag{4.5b}$$
$$\text{subject to } \sum_{I : p \in I} d_I x_I \leq c_p \text{ for all } p \in \mathcal{P} \tag{4.5c}$$
$$\text{and } Bx \leq \mathbb{1}, \tag{4.5d}$$

where $B \in \mathbb{R}^{m' \times \mathcal{I}}$ is an arbitrary non-negative matrix. Let N' be the number of non-zeroes in the matrix B. In the motivating example above, $N' = O(n)$. The algorithm `intervals` easily extends to obtain a $(1-\epsilon)$-approximation to this fractional packing problem in $\tilde{O}\big((N'+m+n)/\epsilon^2\big)$ time.

A direct implementation of the MWU framework is given on the right as `intervals'`; a detailed implementation is given as `intervals''` at the end of the section. As `intervals''` is more complicated than `intervals`, we go through the details gently at the cost of some redundancy. As we describe how to manage the additional packing constraints B, this will also give us a chance to review how to manage explicit packing constraints deterministically in time proportional to the nonzeroes.

We maintain two weight vectors. The first vector, $w : \mathcal{P} \to [1, \infty)$, has one weight for each point capacity constraints. The second, $w' : [m'] \to [1, \infty)$, has one weight for each constraint in B. The point weights w are managed by an augmented range tree as in Section 4. The weights w' are managed in a straightforward

```
intervals''(P,c,I,d,B,ε,η)
  w̃ ← 1, w' ← 1, x ← 0, t ← 0
  let R be a range tree over P
  for each node ν in R
    let P_ν be the points in the leaves of ν
    W̃(ν) ← |P_ν|
    ν.L ← lazy-inc(ζ)
      for ζ = ε² / ((m+m') log(m+m'))
    for all p ∈ P_ν
      ν.L.insert(p, 1/c_p)
  end for
  while t < 1
    I ← interval approx. maximizing the ratio
        v_I / (d_I Σ_{p∈P∩I} w̃_p + ⟨w', BI⟩)
    let V be the roots of the O(log m) maximal
      subtrees contained in I
    γ ← (Σ_{p∈P} w̃_e + Σ_{i'∈[m']} w̃_{i'}) / (Σ_{ν∈V} W̃(ν) + ⟨w', BI⟩)
    κ ← min{ min_{p∈I∩P} c_p/d_I , min_{i'∈[m]} 1/⟨e_{i'}, BI⟩ }
    δ ← εκ/(ηγ)
    x ← x + δγI
    for each root ν of a maximal subtree of R
        contained in I
      Δ ← ν.L.inc(log m/(κ d_I))
      for each increment (p, ξ) ∈ Δ
        w̃_p ← exp(εξ/log m) w̃_p
        for all ν' ∈ R with p ∈ P_{ν'}
          update W̃(ν')
      end for
    end for
    for all i' ∈ [m'] with B_{i',I} ≠ 0
      w'_{i'} ← w'_{i'} · exp(εκ⟨e_{i'}, BI⟩)
    t ← t + δ
  end while
  return x
```

fashion with no data structures[5]. Each iteration, we select an interval I that (approximately) maximizes the ratio $v_I/(d_I\overline{w}(I) + \langle w', BI\rangle)$. The selection is done by lazy bucketing. We add a fraction of I to the running solution and need to update the weights w and w'. To update the weights for the point capacities, we visit the root ν of each of $O(\log m)$ maximal subtrees contained in I, and call `inc` on the `lazy-inc` data structure stored there. The remaining weights w' are updated directly by incrementing the weight of each constraint i' with $B_{i',I} \neq 0$ by the exact amount dictated by the MWU framework.

Between the point capacities and the packing constraints B, there are now $m + m'$ constraints, hence $\tilde{O}\big((m+m')/\epsilon^2\big)$ iterations total. The total number of increments committed by `inc` is $\tilde{O}\big((m+m')/\epsilon^2\big)$, hence so is the total amount of time spent incrementing weights by Theorem 3.1. As noted in the footnote in Section 2, each interval I can only be selected $O\big(\ln(m+m')/\epsilon^2\big)$ times. A weight $w'_{i'}$ corresponding to the i'th row of B is incremented only when an interval I with $B_{i',I} \neq 0$ is visited. Thus, the total time spent incrementing the weights w' is at most $\tilde{O}\big(N'/\epsilon^2\big)$.

The total work spent selecting the best interval in each iteration is amortized by lazy bucketing. In this setting, we draw an interval I from the highest bucket, and we have to recompute the ratio $v_I/(d_I\overline{w}(I) + \langle w', BI\rangle)$. We extract the point weight $\overline{w}(I)$ from the augmented range tree in $O(\log m)$ time, as in Section 4. Since the total number of times we need to recompute the ratio of an interval is $\tilde{O}\big((m+m'+n)/\epsilon^2\big)$, the total time spent collecting values from the augmented range tree is $\tilde{O}\big((m+m'+n)/\epsilon^2\big)$. To compute $\langle w', BI\rangle$, we simply visit every nonzero entry $B_{i',I} \neq 0$ in I's column in B to compute the weighted sum $\langle w', BI\rangle$. For a fixed interval I, we only select I or demote I to a lower bucket $O\big(\ln(m+m')/\epsilon^2\big)$ times over the entire algorithm. Consequently, we only visit a nonzero entry $B_{i',I} \neq 0$ in I's column at most $\tilde{O}\big(1/\epsilon^2\big)$ times. Summed over all intervals $I \in \mathcal{I}$, the total time spent computing weight sums of the form $\langle w', BI\rangle$ is $\tilde{O}\big(N/\epsilon^2\big)$. Thus, the total time spent recomputing ratios for the lazy bucketing scheme is $\tilde{O}\big((m+n+N')/\epsilon^2\big)$. This gives the following overall running time.

THEOREM 4.1. *Let $\mathcal{P}$ be a set of m points with positive capacities $c \in \mathbb{R}^{\mathcal{P}}$, and $\mathcal{I}$ a set of n intervals with positive sizes $d \in \mathbb{R}^{\mathcal{I}}_{>0}$ and values $v \in \mathbb{R}^{\mathcal{I}}_{>0}$. Let $B \in \mathbb{R}^{m'\times\mathcal{I}}_{\geq 0}$ be a positive matrix with N' nonzeroes. Then there is an $\tilde{O}\big((N'+m+n)/\epsilon^2\big)$-time $(1-\epsilon)$-approximation for the fractional augmented interval packing problem (4.5a).*

5 Lazy increments

In this section, we give implementation details for the `lazy-incs` data structure and prove the bounds of Theorem 3.1. The data structure is based on the deterministic update scheme of Young [38]. In particular, we implement a clean and isolated interface that allows for more dynamic settings and simplifies the reasoning when used in more complex combinations, such as the augmented range tree data structure of Section 4. A brief introduction that discusses the

[5] One can also apply techniques by Young [38] to manage w'.

```
lazy-incs(ζ)
  incs ← 𝕆^ℤ (allocated lazily)
  rem ← 𝕆^ℤ (allocated lazily)
  depth ← ⌈log 1/ζ⌉

insert(i, ρ)
  rate(i) ← ρ
  ℓ ← ⌈log ρ⌉
  W(ℓ) ← W(ℓ) ∪ {i}
  frac-incs(i) ← −apx-rem(ℓ)

delete(i)
  ℓ ← ⌈log rate(i)⌉
  δ ← (frac-incs(i) + apx-rem(ℓ)) · rate(i)/2^ℓ
  rate(i) ← 0
  W(ℓ) ← W \ {i}
  return δ

inc(ρ)
  ℓ ← ⌈log ρ⌉
  rem(ℓ) ← rem(ℓ) + ρ/2^ℓ
  return flush(ℓ)

apx-rem(ℓ)
  return Σ_{λ=ℓ}^{λ=ℓ+depth} rem(λ) · 2^{ℓ−λ}
```

```
flush(ℓ)
  Δ ← ∅
  if rem(ℓ) ≥ 1
    δ ← ⌊rem(ℓ)⌋
    rem(ℓ) ← rem(ℓ) − δ
    incs(ℓ) ← incs(ℓ) + δ
    for i ∈ W(ℓ)
      frac-incs(i) ← frac-incs(i) + δ
      if frac-incs(i) ≥ 1
        δ′ ← ⌊frac-incs(i)⌋
        frac-incs(i) ← frac-incs(i) − δ′
        Δ ← Δ ∪ {(i, δ′ · rate(i)/2^ℓ)}
      end if
    end for
    rem(ℓ − 1) ← rem(ℓ − 1) + δ/2
    Δ ← Δ ∪ (flush(ℓ − 1))
  end if
  return Δ

full-rem(ℓ)
  return Σ_{λ≥ℓ} rem(λ) · 2^{ℓ−λ}

full-incs(ℓ)
  return incs(ℓ) + full-rem(ℓ)
```

intuition of the `lazy-incs` data structure is provided in Section 3.

The careful reader might notice that two functions, `full-rem(`ℓ`)` and `full-incs(`ℓ`)`, are not needed to implement the API. `full-rem(`ℓ`)` gives an exact account of the total number of fractional increments for level ℓ, where a fractional increment in the larger level $\ell + 1$ contributes half of a fractional increment to the total for level ℓ, and so on. `full-incs(`ℓ`)` then counts the exact total number of increments to level ℓ by adding `incs(`ℓ`)` to `full-rem(`ℓ`)`. Although immaterial to the implementation, the values of `full-incs(`ℓ`)` and `full-rem(`ℓ`)` are relatively stable and easy to analyze. The actual operations are then analyzed relative to these values.

The first lemma untangles `flush` by showing that it preserves `full-incs(`ℓ`)` for all ℓ.

LEMMA 5.1. *For any two levels $\ell_1, \ell_2 \in \mathbb{Z}$,* `flush(`$\ell_1$`)` *does not change the value of* `full-incs(`ℓ_2`)`.

Proof. `flush(`ℓ_1`)` only modifies `rem(`λ`)` and `incs(`λ`)` for $\lambda \le \ell_1$, and `full-incs(`ℓ_2`)` is a function of `rem(`λ`)` and `incs(`λ`)` for $\lambda \ge \ell_2$. If $\ell_2 > \ell_1$, then these sets of values are disjoint, and the claim holds.

Suppose $\ell_1 \ge \ell_2$. If `incs(`ℓ_1`)` ≥ 1, then `flush(`ℓ_1`)` transfers some $\delta > 0$ from `rem(`ℓ_1`)` to `incs(`ℓ_2`)`, adds half of δ to `rem(`$\ell_1 - 1$`)`, and then recursively calls `flush(`$\ell_1 - 1$`)`. By induction on $\ell_1 - \ell_2$, the recursive call to `flush(`$\ell_1 - 1$`)` preserves `full-incs(`ℓ_2`)`, and it suffices to analyze the changes before the recursive call.

Let `incs`$'$ and `rem`$'$ fix the values of `incs` and `rem` before the transfer of δ, and let `incs`$''$ and `rem`$''$ fix the values of `incs` and `rem` after the transfer. `incs`$'$ and `rem`$'$ equal `incs`$''$ and `rem`$''$ almost everywhere, except `incs`$''(\ell_1) =$ `incs`$'(\ell_1) + \delta$, `rem`$''(\ell_1) =$ `rem`$'(\ell_1) - \delta$, and `rem`$''(\ell - 1) =$ `rem`$'(\ell - 1) + \delta/2$. If $\ell_1 = \ell_2$, then before the transfer `full-incs(`ℓ_2`)` equals

$$\begin{aligned}
&\texttt{incs}'(\ell_2) + \texttt{rem}'(\ell_2) + \sum_{\lambda > \ell_2} \texttt{rem}'(\lambda) 2^{\ell_2 - \lambda} \\
&\quad = (\texttt{incs}''(\ell_2) - \delta) + (\texttt{rem}''(\ell_2) + \delta) + \sum_{\lambda > \ell_2} \texttt{rem}''(\lambda) 2^{\ell_2 - \lambda} \\
&\quad = \texttt{incs}''(\ell_2) + \texttt{rem}''(\ell_2) + \sum_{\lambda > \ell_2} \texttt{rem}''(\lambda) 2^{\ell_2 - \lambda},
\end{aligned}$$

which equals `full-incs(`ℓ_2`)` after the transfer of δ, as

desired. If $\ell_1 > \ell_2$, then before the transfer, we have,

$$\begin{aligned}
&\texttt{full-incs}(\ell_2)\\
&= \texttt{incs}'(\ell_2) + \texttt{rem}'(\ell_1 - 1)\cdot 2^{\ell_2-(\ell_1-1)} + \texttt{rem}'(\ell)\cdot 2^{\ell_2-\ell_1}\\
&\quad + \sum_{\substack{\lambda\geq\ell_2\\ \lambda\notin\{\ell_1,\ell_1-1\}}} \texttt{rem}'(\lambda)\cdot 2^{\ell_2-\lambda}\\
&= \texttt{incs}''(\ell_2) + \left(\texttt{rem}''(\ell_1-1) + \frac{\delta}{2}\right)\cdot 2^{\ell_2-(\ell_1-1)}\\
&\quad + (\texttt{rem}''(\ell)-\delta)\cdot 2^{\ell_2-\ell_1} + \sum_{\substack{\lambda\geq\ell_2\\ \lambda\notin\{\ell_1,\ell_1-1\}}} \texttt{rem}''(\lambda)\cdot 2^{\ell_2-\lambda}\\
&= \texttt{incs}''(\ell_2) + \texttt{rem}''(\ell_1-1)\cdot 2^{\ell_2-(\ell_1-1)}\\
&\quad + \texttt{rem}''(\ell)\cdot 2^{\ell_2-\ell_1} + \sum_{\substack{\lambda\geq\ell_2\\ \ell_2\notin\{\ell_1,\ell_1-1\}}} \texttt{rem}''(\lambda)\cdot 2^{\ell_2-\lambda}
\end{aligned}$$

which equals $\texttt{full-incs}(\ell_2)$ after the transfer, as desired. ∎

The next lemma shows that `inc` increases $\texttt{full-incs}(\ell)$ by the correct amount if $\texttt{full-incs}(\ell)$ counted the number of increments to a weight that incremented at the rate 2^ℓ.

LEMMA 5.2. *Let $\rho > 0$ and $\ell = \lceil \log \rho \rceil$. For all levels $\lambda \leq \ell$, $\texttt{inc}(\rho)$ increases $\texttt{full-incs}(\lambda)$ by $2^\lambda/\rho$.*

Proof. $\texttt{inc}(\rho)$ increases $\texttt{rem}(\ell)$ by $\rho/2^\ell$ and then calls $\texttt{flush}(\ell)$. By Lemma 5.1, $\texttt{flush}(\ell)$ does not affect $\texttt{full-incs}(\lambda)$, so it suffices to consider the increment to $\texttt{rem}(\ell)$. Let $\texttt{rem}'$ denote the value of `rem` before the increase and let $\texttt{rem}''$ denote the value of `rem` after. Before the increment, we have

$$\begin{aligned}
&\texttt{full-incs}(\lambda)\\
&= \texttt{incs}'(\lambda) + \sum_{\substack{\mu\geq\lambda\\ \mu\neq\ell}} \texttt{rem}'(\mu)2^{\lambda-\mu} + \texttt{rem}(\ell)2^{\lambda-\ell}\\
&= \texttt{incs}''(\lambda) + \sum_{\substack{\mu\geq\lambda\\ \mu\neq\ell}} \texttt{rem}''(\mu)2^{\lambda-\mu} + \left(\texttt{rem}''(\ell) - \frac{2^\ell}{\rho}\right)2^{\lambda-\ell}\\
&= \texttt{incs}''(\lambda) + \sum_{\mu\geq\lambda} \texttt{rem}''(\mu)2^{\lambda-\mu} - \frac{2^\lambda}{\rho},
\end{aligned}$$

which is the value of $\texttt{full-incs}(\lambda)$ after the update minus the claimed difference $2^\lambda/\rho$. ∎

The previous lemma shows that `inc` increments each $\texttt{full-incs}(\ell)$ by exactly the right amount. The following lemma argues that we approximately track the number of increments for each weight i, by arguing that the committed increments to weight i are closely coupled with the increments to $\texttt{full-incs}(\lceil \log \texttt{rate}(i) \rceil)$.

LEMMA 5.3. *Let weight i be tracked continuously by the data structure in level $\ell = \lceil \log \texttt{rate}(i) \rceil$ without being deleted. Consider the sum $\Phi + V$, where V is the total number of increments committed by the data structure to i (communicated in the return values of `inc`), and Φ is defined to be*

$$\Phi \stackrel{\text{def}}{=} (\texttt{frac-incs}(i) + \texttt{full-rem}(\ell)) \cdot \frac{\texttt{rate}(i)}{2^\ell}.$$

Then

(a) For any level ℓ', $\texttt{flush}(\ell')$ does not change the sum $V + \Phi$.

(b) A call to $\texttt{inc}(\rho)$ for $\rho \geq \texttt{rate}(i)$ increases the sum $V + \Phi$ by $\frac{\texttt{rate}(i)}{\rho}$.

(c) When i is inserted, we have $\Phi \leq \zeta$. After each `inc`, $\Phi \leq 3$. In particular, after a sequence of K increments at rates of $\rho_1, \ldots, \rho_K \geq \texttt{rate}(i)$, V is within an additive factor of 3 of $\sum_{k=1}^K \texttt{rate}(i)/\rho_k$.

(d) Suppose i is deleted after a sequence of K increments at rates of $\rho_1, \ldots, \rho_K$ with $\lceil \log \rho_k \rceil \geq \lceil \log \texttt{rate}(i) \rceil$ for each k, and the call to $\texttt{delete}(i)$ returns an increment of δ. Then $V + \delta$ is within an additive factor of ζ of $\sum_{k=1}^K \texttt{rate}(i)/\rho_k$.

Proof. (a) Consider the "immediate" part of $\texttt{flush}(\ell')$ executed before the recursive call to $\texttt{flush}(\ell - 1)$ (if any). It suffices to show that the immediate parts of each `flush` do not change the sum $V + \Phi$. If $\ell' < \ell$, then this work does not change commit any weight to i, and any change to $\texttt{rem}(\ell')$ is not included in the computation to $\texttt{full-rem}(\ell)$. Thus, the immediate part of $\texttt{flush}(\ell')$ has no impact on $V + \Phi$ for all $\ell' < \ell$.

If $\ell = \ell'$, then $\texttt{flush}(\ell')$ may delete $\delta > 0$ weight from $\texttt{rem}(\ell)$, hence decrease $\texttt{full-rem}(\ell)$ by δ. The same quantity δ is added to $\texttt{frac-incs}(i)$. Then $\texttt{flush}(\ell')$ may decrease $\texttt{frac-incs}(i)$ by a second quantity $\delta' > 0$, and in turn $\delta' \cdot \texttt{rate}(i)/2^\ell$ is added to V. Thus, $\texttt{full-rem}(\ell)$ decreases by δ, $\texttt{frac-incs}(i)$ increases by $\delta - \delta'$, and V increases by $\texttt{rate}(i)/2^\ell$. Altogether, some value may shift from Φ to V, but the sum $\Phi + V$ remains fixed.

Finally, if $\ell' > \ell$, then the immediate part of $\texttt{flush}(\ell')$ may decrease $\texttt{rem}(\ell')$ by some $\delta > 0$, but then it increases $\texttt{rem}(\ell' - 1)$ by $\delta/2$. Since $\ell' - 1 \geq \ell$, this results in no change in $\texttt{full-rem}(\ell)$, hence no change in the sum $\Phi + V$.

(b) Let $\ell' = \lceil \log \rho \rceil$. $\texttt{inc}()$ increases $\texttt{rem}(\ell')$ by $\rho/2^{\ell'}$ and then calls $\texttt{flush}(\ell')$. The addition to $\texttt{rem}(\ell')$ increases $\texttt{full-rem}(\ell)$ by $\left(2^{\ell'}/\rho\right) \cdot 2^{\ell-\ell'} = 2^\ell/\rho$, which in turn increases Φ by $\left(2^\ell/\rho\right) \cdot \left(\texttt{rate}(i)/2^\ell\right) = \texttt{rate}(i)/2^\ell$. By part (a), the remaining call to $\texttt{flush}(\ell')$ does not affect the sum $\Phi + V$. Thus, the total increase in $\Phi + V$ is $\texttt{rate}(i)/2^\ell$, as desired.

(c) Since $\texttt{apx-rem}(\ell) \le \texttt{full-rem}(\ell) \le \texttt{apx-rem}(\ell) + \zeta$, and $\texttt{frac-incs}(i)$ is initialized to $-\texttt{apx-rem}(\ell)$, we have $\Phi \in [0, \zeta]$ when i is inserted. By design, the data structure keeps $\texttt{rem}(\ell') \in [0, 1]$ for all levels ℓ', which implies that $\texttt{full-rem}(\ell') \in [0, 2)$ for all levels ℓ' as well. The data structure also keeps $\texttt{frac-incs}(i)$ in the range $-\texttt{full-rem}(\ell') \le -\texttt{apx-rem}(\ell') \le \texttt{frac-incs}(i) \le 1$. Thus, Φ always lies in the range $0 \le \Phi < 3 \cdot \left(\delta/2^{\ell}\right) \le 3$.

(d) Fix V and Φ to just before the call to $\texttt{delete}(i)$. By (b), we have $V + \Phi = \sum_{k=1}^{K} \texttt{rate}(i)/\rho_k + \eta$ for some $\eta \in [0, \zeta]$. Furthermore,

$$\begin{aligned}\Phi - \delta &= (\texttt{full-rem}(\ell) - \texttt{apx-rem}(\ell)) \cdot \frac{\texttt{rate}(i)}{2^{\ell}} \\ &\in [0, \zeta].\end{aligned}$$

Thus, $V + \delta = (V + \Phi) + (\delta - \Phi) = \sum_{k=1}^{K} \texttt{rate}(i)/\rho_k + \eta'$, for some $\eta' \in [-\zeta, \zeta]$, as desired. ■

In the final stage of our analysis, we bound the running time of the functions of the `lazy-incs` data structure. At the essence of the running times is that the increments for the levels can be maintained essentially for free, for the same reason that a binary number can be incremented in constant amortized.

LEMMA 5.4. *For a fixed instance of* $\texttt{lazy-incs}(\zeta)$*, let*

- $M \in \mathbb{N}$ *be the number of calls to* `inc`*,*
- I *the number of calls to* `insert`*, and*
- D *the number of calls to* `delete`*, and*
- *for each constraint* i*, let* V_i *be the sum of increments for constraint* i *confirmed in the return values of calls to* `inc` *and* `delete`*.*

Then `lazy-incs` *executes all* M *calls to* `inc`*,* I *calls to* `insert`*, and* D *calls to* `delete` *in total running time* $\tilde{O}(M + (I + D)\log(1/\zeta) + \sum_i V_i)$*, where* $\tilde{O}$ *hides logarithmic factors in* M*.*

Proof. It is easy to see that each `insert` and `delete` take $\tilde{O}(1)$ time. The work of each call to `inc` can be divided into updating counters for levels (`incs` and `rem`) and for the tracked weights (`frac-incs`). The work to maintain `incs` and `rem` take $\tilde{O}(1)$ amortized time per `inc` for the same reason that incrementing a binary integer takes constant amortized time: when we call $\texttt{inc}(\rho)$ with $\ell = \lceil \log \rho \rceil$, we put 1 credit on level ℓ, half a credit on level $\ell + 1$, one fourth a credit on level $\ell + 2$, and so on. These credits, for each ℓ, are at least as much as the increase to $\texttt{full-incs}(\ell)$, and the total number of credits dispersed is 2. Since we only execute the body of a $\texttt{flush}(\ell)$ when $\texttt{rem}(\ell)$, and $\texttt{incs}(\ell) + \texttt{rem}(\ell)$ is always with 1 of $\texttt{full-incs}(\ell)$, the number of times $\texttt{flush}(\ell)$ is executed is proportional to the increase in $\texttt{full-incs}(\ell)$, which in turn can be paid for by accrued tokens. As per maintaining $\texttt{frac-incs}(i)$, for each constraint i, $\texttt{frac-incs}(i)$ is increased by at least 1 whenever it is touched by $\texttt{flush}(\ell)$ for $\ell = \lceil \log \texttt{rate}(i) \rceil$. Every time $\texttt{frac-incs}(i)$ exceeds 1, at least one unit of weight is committed by the data structure and added to V_i. Since $\texttt{frac-incs}(i) \ge -2$ at all times, $\texttt{flush}(\ell)$ touches $\texttt{frac-incs}(i)$ at most a constant number of times before we either increase V_i by at least one or i is deleted. Thus, the work corresponding to maintaining $\texttt{frac-incs}(i)$ can be charged to V_i plus any calls to $\texttt{delete}(i)$. ■

This concludes the proof of Theorem 3.1.

Acknowledgments: We thank Neal Young and David Karger for helpful comments and pointers.

References

[1] P. K. Agarwal and J. Pan. Near-linear algorithms for geometric hitting sets and set covers. In *Proc. 30th Annu. Sympos. Comput. Geom.* (SoCG), page 271, 2014.

[2] Z. Allen-Zhu and L. Orecchia. Nearly-linear time positive LP solver with faster convergence rate. In *Proc. 47th Annu. ACM Sympos. Theory Comput.* (STOC), pages 229–236, 2015.

[3] E. M. Arkin and E. B. Silverberg. Scheduling jobs with fixed start and end times. *Discrete Appl. Math.*, 18(1):1–8, 1987.

[4] A. Asadpour, M. X. Goemans, A. Madry, S. O. Gharan, and A. Saberi. An $O(\log n/\log\log n)$-approximation algorithm for the asymmetric traveling salesman problem. In *Proc. 21st ACM-SIAM Sympos. Discrete Algs.* (SODA), pages 379–389, 2010.

[5] A. Bar-Noy, R. Bar-Yehuda, A. Freund, J. (Seffi) Naor, and B. Schieber. A unified approach to approximating resource allocation and scheduling. *J. Assoc. Comput. Mach.*, 48(5): 1069–1090, Sept. 2001.

[6] F. Barahona. Packing spanning trees. *Math. of Oper. Res.*, 20(1):104–115, Feb. 1995.

[7] A. A. Benczúr and D. R. Karger. Randomized approximation schemes for cuts and flows in capacitated graphs. *SIAM J. Comput.*, 44(2):290–319, 2015.

[8] D. Bienstock and G. Iyengar. Approximating fractional packings and coverings in O(1/epsilon) iterations. *SIAM J. Comput.*, 35(4):825–854, 2006.

[9] A. Borodin. Greedy algorithms and why simple algorithms can be complex. UC Irvine Comp. Sci. Dept. Seminar Series, Feb. 2009.

[10] V. T. Chakaravarthy, A. R. Choudhury, S. Gupta, S. Roy, and Y. Sabharwal. Improved algorithms for resource allocation under varying capacity. In *Proc. 22nd Annu. European Sympos. Algorithms* (ESA), pages 222–234. Springer, 2014.

[11] C. Chekuri, T. Jayram, and J. Vondrák. On multiplicative weight updates for concave and submodular function maximization. In *Proc. 6th Conf. Innov. Theoret. Comp. Sci.* (ITCS), pages 201–210, 2015.

[12] W. H. Cunningham. Optimal attack and reinforcement of a network. *J. Assoc. Comput. Mach.*, 32 (3):549–561, July 1985.

[13] K. Elbassioni, N. Garg, D. Gupta, A. Kumar, V. Narula, and A. Pal. Approximation algorithms for the unsplittable flow problem on paths and trees. In *LIPIcs-Leibniz Int. Proc. in Informatics*, volume 18. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2012.

[14] T. Erlebach and F. C. Spieksma. Interval selection: Applications, algorithms, and lower bounds. *J. Algorithms*, 46(1):27 – 53, 2003.

[15] H. N. Gabow and K. Manu. Packing algorithms for arborescences (and spanning trees) in capacitated graphs. *Math. Prog.*, 82(1-2):83–109, 1998.

[16] N. Garg and J. Könemann. Faster and simpler algorithms for multicommodity flow and other fractional packing problems. *SIAM J. Comput.*, 37 (2):630–652, 2007. Preliminary version in Proc. 39th Annu. IEEE Sympos. Found. Comput. Sci. *(FOCS)*(1998).

[17] S. O. Gharan, A. Saberi, and M. Singh. A randomized rounding approach to the traveling salesman problem. In *Proc. 52nd Annu. IEEE Sympos. Found. Comput. Sci.* (FOCS), pages 550–559, 2011.

[18] M. D. Grigoriadis and L. G. Khachiyan. Fast approximation schemes for convex programs with many blocks and coupling constraints. *SIAM J. Optim.*, 4(1):86–107, 1994.

[19] J. Holm, K. de Lichtenberg, and M. Thorup. Polylogarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *J. Assoc. Comput. Mach.*, 48 (4):723–760, 2001.

[20] D. R. Karger. Random sampling and greedy sparsification for matroid optimization problems. *Math. Program.*, 82:41–81, 1998. Preliminary version in Proc. 34th Annu. IEEE Sympos. Found. Comput. Sci. *(FOCS)*, 1993.

[21] D. R. Karger. Minimum cuts in near-linear time. *J. Assoc. Comput. Mach.*, 47(1):46–76, Jan. 2000.

[22] P. N. Klein and N. E. Young. On the number of iterations for Dantzig-Wolfe optimization and packing-covering approximation algorithms. *SIAM J. Comput.*, 44(4):1154–1172, 2015.

[23] P. N. Klein, S. A. Plotkin, C. Stein, and É. Tardos. Faster approximation algorithms for the unit capacity concurrent flow problem with applications to routing and finding sparse cuts. *SIAM J. Comput.*, 23(3):466–487, 1994.

[24] C. Koufogiannakis and N. E. Young. A nearly linear-time PTAS for explicit fractional packing and covering linear programs. *Algorithmica*, 70 (4):648–674, 2014. Preliminary version in Proc. 48th Annu. IEEE Sympos. Found. Comput. Sci. *(FOCS)*, (2007).

[25] W. D. Matula. A linear time $2 + \epsilon$ approximation algorithm for edge connectivity. In *Proc. 4th ACM-SIAM Sympos. Discrete Algs.* (SODA), pages 500–504, 1993.

[26] A. Mądry. Faster approximation schemes for fractional multicommodity flow problems via dynamic graph algorithms. In *Proc. 42nd Annu. ACM Sympos. Theory Comput.* (STOC), pages 121–130, 2010.

[27] C. S. J. A. Nash-Williams. Edge-disjoint spanning trees of finite graphs. *J. London Math. Soc.*, 36: 445–450, 1961.

[28] Y. Nesterov. *Introductory lectures on convex optimization*, volume 87 of *Applied Optimization*. Springer, 2004.

[29] Y. Nesterov. Fast gradient methods for network flow problems. 20th Internat. Symp. Math. Prog., 2009.

[30] S. A. Plotkin, D. B. Shmoys, and É. Tardos. Fast approximation algorithms for fractional packing and covering problems. *Math. of Oper. Res.*, 20 (2):257–301, 1995.

[31] A. Schrijver. *Combinatorial Optimization: Polyhedra and Efficiency*, volume 24 of *Algorithms and Combinatorics*. Springer, 2003.

[32] F. Shahrokhi and D. W. Matula. The maximum concurrent flow problem. *J. Assoc. Comput. Mach.*, 37(2):318–334, 1990.

[33] V. Trubin. Strength of a graph and packing of trees and branchings. *Cybernetics and Syst. Analysis*, 29 (3):379–384, 1993.

[34] W. T. Tutte. On the problem of decomposing a graph into n connected components. *J. London Math. Soc.*, 36:221–230, 1961.

[35] D. Wang, S. Rao, and M. W. Mahoney. Unified acceleration method for packing and covering problems via diameter reduction. In *Proc. 43rd Internat. Colloq. Automata Lang. Prog.* (ICALP), pages 50:1–50:13, 2016.

[36] P. Winkler and L. Zhang. Wavelength assignment and generalized interval graph coloring. In *Proc. 14th ACM-SIAM Sympos. Discrete Algs.* (SODA), pages 830–831, 2003.

[37] N. E. Young. Randomized rounding without solving the linear program. In *Proc. 6th ACM-SIAM Sympos. Discrete Algs.* (SODA), pages 170–178, 1995.

[38] N. E. Young. Nearly linear-time approximation schemes for mixed packing/covering and facility-location linear programs. *CoRR*, abs/1407.3015, 2014. URL `http://arxiv.org/abs/1407.3015`.

A Packing bases in a matroid

Consider the more general case where $\mathcal{B}$ is the set of bases of a matroid $\mathcal{M} = (\mathcal{N}, \mathcal{I})$ with n elements and rank k. In the case of packing spanning trees, dynamic maintenance of an MST is very efficient. The lazy-increment data structure solves the problem of maintaining weights. For general matroids the bottleneck is maintaining a minimum weight base during the sequence of $\tilde{O}(n/\epsilon^2)$ weight updates. If we rerun the greedy algorithm at each iteration, then the overall running time would be $\tilde{O}(n^2/\epsilon^2)$. To mimick the `capacitated-spanning-trees` algorithm of Section 3, we design a dynamic data structure `dynamic-min-base` that maintains the minimum weight base amidst a sequence of increments to the weights in order to replace one of the factors of n by the rank k. At the end of this section we state the running time that is achievable assuming a dynamic data structure for computing a minimum-weight base.

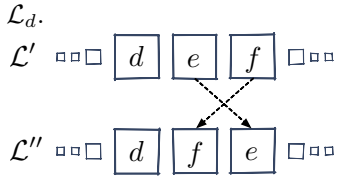
The data structure `dynamic-min-base` is initialized by a matroid $\mathcal{M} = (\mathcal{N}, \mathcal{I})$ and an initial set of weights $w : \mathcal{N} \to \mathbb{R}$ and supports two operations. The first, `min-base()`, returns the minimum weight base. The second, `inc(`e`,`δ`)`, takes an element $e \in \mathcal{N}$ and a positive increment $\delta > 0$, and increases the weight w_e by δ, adjust the minimum weight base as needed. Internally, `dynamic-min-base` maintains the ground set $\mathcal{N}$ in a list $\mathcal{L}$ sorted in nondecreasing order. The minimum weight independent set can always be computed from one pass of $\mathcal{L}$ by starting with an empty independent set $I = \emptyset$ and processing the elements in nondecreasing order of weight, adding any element e to I such that $I + e \in \mathcal{I}$. Of course, reading all of $\mathcal{N}$ takes $O(n)$ time. To reduce the running time, we avoid rebuilding the minimum weight independent set from scratch. When the weight of an element $e \in I$ is incremented, we remove e from I and move e down $\mathcal{L}$ to reflect its new weight. Then, starting from the former position of e in $\mathcal{L}$ rather than the beginning, we replace e by f by taking the first element such that $I + f \in \mathcal{I}$.

LEMMA A.1. *The dynamic data structure* `dynamic-min-base` *maintains the minimum weight independent set.*

Proof. Let $\mathcal{L}$ list the elements of $\mathcal{N}$ in nondecreasing order of weight. For each element $e \in \mathcal{N}$, let $\mathcal{L}_e$ be the prefix of $\mathcal{L}$ up to and including e, and let $I_e = I \cap \mathcal{L}_e$. For $I \in \mathcal{I}$ to be the minimum weight independent set, it suffices to show that for every element $e \in \mathcal{N}$, I_e is a base in $\mathcal{L}_e$. Certainly, this holds for our the initial independent set by construction. It remains that the subroutine `inc` preserves this invariant.

Suppose we increment the weight of an element e. If $e \notin I$, or the position of e does not change, then the claim is immediate. Suppose $e \in I$ and the weight increment demotes e to a new position further down the list $\mathcal{L}$. Breaking the weight increment into a sequence of small weight increments that push e down just one spot at a time, it suffices to assume e moves back one slot in $\mathcal{L}$.

Let d be the element immediately preceding e and f the element immediately after e just before the increment. Let $\mathcal{L}'$ and I' fix the value of $\mathcal{L}$ and I before the subroutine and let and $\mathcal{L}''$ and I'' be the value of $\mathcal{L}$ and I after. $\mathcal{L}'$ lists d, e, f in consecutive order and $\mathcal{L}''$ lists d, f, e in consecutive order. Since no elements in $\mathcal{L}_d$ change position or w/r/t membership in I (i.e., $I'_d = I''_d$ and $\mathcal{L}'_d = \mathcal{L}''_d$), the invariant still holds for all elements $e \in \mathcal{L}_d$.



Dynamic Minimum Weight Base

Operations	Description	Time
`dynamic-min-base(`$\mathcal{M}$`,`w`)`	Given a matroid $\mathcal{M} = (\mathcal{N}, \mathcal{I})$ with n elements and rank k, and an initial set of weights $w : \mathcal{N} \to \mathbb{R}$ over the groundset, initializes the data structure and computes the initial minimum weight base.	$O(n \log n + nkQ)$ amortized
`min-base()`	Returns the minimum weight base.	$O(1)$
`inc(`e`,`δ`)`	For an element $e \in \mathcal{N}$ and positive value $\delta > 0$, increases the weight w_e by δ.	$O(k)$ amortized

```
dynamic-min-base(M = (N, I), w : N → R)
  L ← sorted list of N increasing in w
  I ← ∅
  for e ∈ L in order
    if I + e ∈ I then I ← I + e

min-base()
  return I

inc(e, δ)
  w_e ← w_e + δ
  let e' be the element after e in L (if any)
  reinsert e into L w/r/t w_e
  if e ∈ I and the position of e in L changed
    I ← I − e
    for f ∈ L in order starting from e'
      if f ∉ I and I + f ∈ I
        I ← I + f
        break loop
    end for
  end if
```

The subroutine first considers f. In the simplest case, we have $f \in I'$ already. Here the subroutine leaves f in I and then adds e back in before exiting, resulting in the same independent set as we started with (that is, $I' = I''$). However, the invariants (that I''_e is a base in $\mathcal{L}''_e$ for all e) may be broken because $\mathcal{L}''$ is different. To this end, we observe that $|I''_d| = \text{rank}(\mathcal{L}''_d)$ because I''_d is a base in $\mathcal{L}''_d$, so

$$\begin{aligned}\left|I''_f\right| &= |I''_d| + 1 = \text{rank}(\mathcal{L}''_d) + 1 \\ &\geq \text{rank}(\mathcal{L}''_d + f) = \text{rank}\big(\mathcal{L}''_f\big),\end{aligned}$$

and

$$\begin{aligned}|I''_e| &= |I''_d| + 2 = \text{rank}(\mathcal{L}''_d) + 2 \\ &\geq \text{rank}(\mathcal{L}''_d + e + f) = \text{rank}\big(\mathcal{L}''_f\big),\end{aligned}$$

as desired.

Suppose $f \notin I'$, and I''_d spans f (i.e., $I''_d + f \notin \mathcal{I}$). Then the subroutine does not add f to I, and then adds e back in, resulting in the same independent set. Since $I''_f = I''_d$ also spans $\mathcal{L}''_d$ and $\mathcal{L}''_f = \mathcal{L}''_d + c$, we have that I''_f is a base in $\mathcal{L}''_f$. Since $I''_e = I''_f + e$, $\mathcal{L}''_e = \mathcal{L}''_f + e$, and I''_f spans $\mathcal{L}''_f$, we have that I''_e spans $\mathcal{L}''_e$, as desired.

In the final case, I'_d does not span f, so f is added to I'_d and the routine terminates. That is, $I'' = I' - e + f$. I''_f is a base in $\mathcal{L}''_f$ because

$$\begin{aligned}\left|I''_f\right| &= |I''_e + f| = |I''_e| + 1 = \text{rank}(\mathcal{L}''_d) + 1 \\ &\geq \text{rank}(\mathcal{L}''_d + f) = \text{rank}\big(\mathcal{L}''_f\big).\end{aligned}$$

I''_e is a base in $\mathcal{L}''_e$ because $|I''_e| = \left|I'_f\right| = \text{rank}\Big(\mathcal{L}'_f\Big) = \text{rank}(\mathcal{L}''_e)$.

For any remaining element g that comes after f in $\mathcal{L}''$, since we deleted exactly one element and added exactly one element, we have $\left|I'_g\right| = |I''_D|$, hence

$$\left|I''_g\right| = \left|I'_g\right| = \text{rank}\big(\mathcal{L}'_g\big) = \text{rank}\big(\mathcal{L}''_g\big),$$

and I''_g is a base in $\mathcal{L}''_g$, as desired. ■

Lemma A.2. *Initialization runs in $O(n \log n)$ time plus $O(nkQ)$ amortized time, and each increment* `inc(`e`,`δ`)` *runs in $O(\log n + kQ)$ amortized time, where k is the rank of the matroid and Q is the running time of a call to the independence oracle. Furthermore, ℓ consecutive calls* `inc(`e`,`δ_1`)`$, \ldots,$`inc(`e`,`δ_ℓ`)` *to* `inc` *with the same element e takes $O(\ell \log n + kQ)$ amortized time.*

Proof. Besides sorting and reinserting the elements by weight, the work is proportional to the number of calls to an independence oracle. Let us bound the number of independence calls for a fixed element $e \in \mathcal{N}$.

Suppose we test if $I + e \in \mathcal{I}$ and indeed $I + e$ is independent. Then e is added to our independent set, and we can charge the oracle call to the last time e was either ejected from I by a weight increment `inc(`e`,`δ`)`, or if this has never happened, when it was rejected by the initialization.

Otherwise, $I + e$ is dependent, and we do not add e to I. This only happens during the initialization phase or when an element d that was before e in $\mathcal{L}$ has its

weight incremented so much that d now comes after e in $\mathcal{L}$. In this case, even though e is not added to I, the rank of $\mathcal{L}_e$ has decreased by 1 because I_e is a base in $\mathcal{L}_e$ and the cardinality of I_e decreased by exactly 1 with the removal of d. In a sequence of independence queries for e without any calls to `inc(`e`,`δ`)`, each time we call $I+e$ and e is not added to I, the rank of $\mathcal{L}_e$ has decreased by 1. In particular, we can only test if $I+e$ is independent at most $k = \text{rank}(\mathcal{M})$ times before e is either added to I or its weight is increased by a call to `inc(`e`,`δ`)`.

Note that when an element e is incremented several times consecutively, we do not need to test if $I+e$ is independent in between these increments. Thus, for ℓ consecutive increments to e, we only pay kQ once, and otherwise pay for each binary search to reinsert e. ■

The data structure `dynamic-min-base` is a drop-in replacement for the dynamic MST data structure of Holm et al. [19]. The running time analysis is similar. The total number of iterations is $\tilde{O}\big(n/\epsilon^2\big)$. By using the `lazy-incs` data structure to maintain the weights, the total time spent updating weights is $\tilde{O}\big(n/\epsilon^2\big)$. `dynamic-min-base` takes $O(nkQ)$ amortized time to initialize, and each full weight increment returned by `lazy-incs` for an element e in the minimum weight base takes $O(k)$ amortized time for `dynamic-min-base` to process. Furthermore, the `lazy-incs` data structure increases the weight of an element e is increased at most $\tilde{O}\big(1/\epsilon^2\big)$ by `lazy-incs.inc` and any increments from `lazy-incs.delete` immediately proceed a `lazy-incs.inc`, so by Lemma A.2, `dynamic-min-base` spends $\tilde{O}\big(k/\epsilon^2\big)$ amortized time processing increments to the weight of e, and $\tilde{O}\big(nk/\epsilon^2\big)$ time processing increments over all elements in $\mathcal{N}$. This gives a total running time of $\tilde{O}\big(nk/\epsilon^2\big)$.

THEOREM A.1. *Let $\mathcal{M} = (\mathcal{N},\mathcal{I})$ be a matroid with n elements and rank k, accessed by an independence oracle that runs in time Q. Using the* `lazy-incs` *structure of Section 5 to approximately maintain edge weights and a dynamic data structure* `dynamic-min-base` *that maintains the minimum weight independent set w/r/t the approximate weights,* `capacitated-bases` *returns a $(1+\epsilon)$-approximation for fractionally packing bases of $\mathcal{M}$ in time $\tilde{O}\big(nkQ/\epsilon^2\big)$.*

The running time $\tilde{O}\big(nkQ/\epsilon^2\big)$ is a factor of $O(k)$ away from really being nearly-linear in the size of the input. Still it improves on the naive $\tilde{O}\big(n^2/\epsilon^2\big)$ running time in many applications where k is much smaller than n. This situation is rectified somewhat in the next section, where we obtain a nearly-linear running time in the unweighted setting that is essentially independent of k.

```
disjoint-bases'(N,B,ε,η)
  w ← 𝟙, x ← 𝟘, t ← 0
  while t < 1
    b ← arg min{w̄(b) : b ∈ B}
    γ ← w̄(N) / w̄(b)
    δ ← ε / (ηγ)
    x ← x + δγb
    for all e ∈ b
      w(e) ← exp(ε)w(e)
    end for
    t ← t + δ
  end while
  return x
```

For some matroids it is possible to implement an efficient dynamic data structure for computing a minimum weight base. We saw this for spanning trees. Another example is laminar matroids. Suppose the total time for h updates on a matroid with n elements and rank k is given by $T(n,k,h)$. Let $S(n,k)$ be the time to initialize the data structure. Here we are assuming that the data structure allows for increments to the element weights and that each operation outputs the one element change to the current minimum weight base. Then we can obtain a $(1-\epsilon)$-approximate fractional packing of bases in time $\tilde{O}(n/\epsilon^2 + S(n,k) + T(n,k,n\log n/\epsilon^2))$.

B Packing bases in the uncapacitated setting

The MWU algorithm simplifies considerably in the uncapacitated case, giving a much improved running time for the case of matroids. The direct implementation of the algorithm, given as `disjoint-bases'`, is essentially the same as the one for the capacitated setting, except all weights update at the same rate because all capacities are uniformly 1. `disjoint-bases'` has only $\tilde{O}\big(n/k\epsilon^2\big)$ iterations instead of $\tilde{O}\big(n/\epsilon^2\big)$ because each iteration increases k weights by the full $\exp(\epsilon)$-factor. If each minimum weight base in `disjoint-bases'` computed by the greedy algorithm, then each iteration is bounded by $O(n)$ calls to an independence oracle, and the total running time is bounded by an $\tilde{O}\big(n^2/k\epsilon^2\big)$ calls to the independence oracle. However, the combination of a simple weight update and the simplicity of the greedy algorithm makes the sequence of bases $\langle b^\ell \rangle$ chosen by `disjoint-bases'` predictable enough to precompute the entire sequence and make the final algorithm nearly linear in n.

Each iteration of `disjoint-bases'` computes the minimum cost base w/r/t the current weights w (initialized to $\mathbb{1}$), and increases the weight of each ele-

```
disjoint-bases(𝒩,ℬ,ϵ,η)
  // let Z = ⌊(ϵ + (1+ϵ)η + ln n)/ϵ⌋
  for ℓ = 1,...,⌊nZ/k⌋
    b^ℓ ← 0
  repeat ⌊Z⌋ times
    for e ∈ 𝒩
      search for first ℓ s.t.
        (a) e ∉ b^ℓ
        (b) b^ℓ + e ∈ ℐ
      b^ℓ ← b^ℓ + e
    end for
  end loop
  w ← 𝟙, x ← 𝟘, t ← 0, ℓ ← 1
  for ℓ = 1,2,... until t ≥ 1
    γ ← w̄(𝒩)/w̄(b^ℓ)
    δ ← ϵ/(ηγ)
    x ← x + δγb^ℓ
    t ← t + δ
  end for
  return x
```

ment in the base by an $\exp(\epsilon)$-factor. Every element $e \in \mathcal{N}$ has its weight that grow along the same sequence of 1, $\exp(\epsilon)$, $\exp(2\epsilon)$, ..., with a uniform upper bound of $\exp(\epsilon + (1+\epsilon)\eta + \ln n)$ for all weights. Let $Z = \lfloor(\epsilon + (1+\epsilon)\eta + \ln n)/\epsilon\rfloor$ denote the maximum number of times an element e can be selected and have its weight increased.

Consider the enlarged groundset $\mathcal{N}' = \mathcal{N} \times [Z]$, where we make Z copies of each element, and define the matroid $\mathcal{M}' = (\mathcal{N}', \mathcal{I}')$ by letting a set $S \subseteq \mathcal{N}'$ be independent iff it contains no two copies of the same element in $\mathcal{N}$ and the underlying elements are independent in $\mathcal{M}$. Define a set of weights $v : \mathcal{N}' \to \mathbb{R}_{>0}$ by $v(e,i) = \exp(\epsilon i)$. If we repeatedly find a minimum cost base b' in $\mathcal{M}'$ w/r/t v and remove all the elements of b' from $\mathcal{N}'$, then the underlying elements of the sequence of bases generated by this procedure is a possible sequence of bases enumerated by `disjoint-bases'`.

In the algorithm `disjoint-bases`, we precompute these bases ahead of time. We start with $\lfloor nZ/k \rfloor$ independent sets $b^1, b^2, \ldots$, each intialized to the empty set $\emptyset$. We implicitly process each element of $\mathcal{N}'$ in increasing order of weight, and use binary search to find the first independent set where the element can be inserted without breaking independence. It is easy to see that this simulates iteratively calling the greedy algorithm on $\mathcal{M} = (\mathcal{N}, \mathcal{I})$ with weights initialized to $\mathbb{1}$ and scaling up the weight of each selected element by $\exp(\epsilon)$ each time an element is selected. However, by using binary search over the $\lfloor nZ/k \rfloor$ sets, the number of independence oracle calls is cut down logarithmically, to $\log(nZ/k) = \tilde{O}(1)$ per element.

THEOREM B.1. *Let $\mathcal{M} = (\mathcal{N}, \mathcal{I})$ be a matroid with n elements and rank k, and let Q denote the cost of a call to an independence oracle for $\mathcal{I}$. Then* `disjoint-bases` *returns an ϵ-approximation for fractional packing disjoint bases in running time $\tilde{O}(nQ/\epsilon^2)$.*

Returning to packing spanning trees, in the uncapacitated setting, we can improve on the log factors of Theorem 1.1 by observing that only a disjoint set union data structure is required to implement the independence oracle.