Dynamic Algorithms for Packing-Covering LPs via Multiplicative Weight Updates

Sayan Bhattacharya*
University of Warwick
S.Bhattacharya@warwick.ac.uk

Peter Kiss
University of Warwick
Peter.Kiss@warwick.ac.uk

Thatchaphol Saranurak
University of Michigan
thsa@umich.edu

Abstract

In the *dynamic linear program* (LP) problem, we are given an LP undergoing updates and we need to maintain an approximately optimal solution. Recently, significant attention (e.g. [Gupta et al. STOC'17; Arar et al. ICALP'18, Wajc STOC'20]) has been devoted to the study of special cases of dynamic packing and covering LPs, such as the dynamic fractional matching and set cover problems. But until now, there is no non-trivial dynamic algorithm for general packing and covering LPs.

In this paper, we settle the complexity of dynamic packing and covering LPs, up to a polylogarithmic factor in update time. More precisely, in the *partially dynamic* setting (where updates can either only relax or only restrict the feasible region), we give near-optimal deterministic ϵ-approximation algorithms with polylogarithmic amortized update time. Then, we show that both partially dynamic updates and amortized update time are necessary; without any of these conditions, the trivial algorithm that recomputes the solution from scratch after every update is essentially the best possible, assuming SETH.

To obtain our results, we initiate a systematic study of the *multiplicative weights update* (MWU) method in the dynamic setting. As by-products of our techniques, we also obtain the first online $(1+\epsilon)$-competitive algorithms for both covering and packing LPs with polylogarithmic recourse, and the first streaming algorithms for covering and packing LPs with linear space and polylogarithmic passes.

Contents

1 Introduction

Packing and covering linear programs (LPs) are defined as follows:

$$\text{Covering LP } \mathbb{C}: \quad \min_{x \in \mathbb{R}^n_{\geq 0}} \{c^\top x \mid Ax \geq b\},$$

$$\text{Packing LP } \mathbb{P}: \quad \max_{y \in \mathbb{R}^m_{\geq 0}} \{b^\top y \mid A^\top y \leq c\},$$

where $A \in \mathbb{R}^{m \times n}_{\geq 0}, b \in \mathbb{R}^m_{\geq 0}, c \in \mathbb{R}^n_{\geq 0}$ contain only non-negative entries. The two LPs are duals to each other, so we let OPT denote their shared optimum. We say that x is an ϵ-*approximation* for $\mathbb{C}$ if $Ax \geq b$ and $c^\top x \leq (1 + \epsilon)$OPT. Similarly, y is an ϵ-*approximation* for $\mathbb{P}$ if $A^\top y \leq c$ and $b^\top y \geq \text{OPT}/(1 + \epsilon)$.

Packing-covering LPs have a wide range of applications in various different contexts, such as in approximation algorithms [LN93, Tre98], flow control [BBR04], scheduling [PST95], graph embedding [PST95] and auction mechanisms [ZN01]. Thus, a long line of work is concerned with the study of efficient algorithms for packing-covering LPs [ZO15, AZO19, WRM15, KY14, You14, CQ18, Qua20], and currently, it is known how to compute ϵ-approximation in $\tilde{O}(N/\epsilon)$ time [AZO19, WRM15]. Here, N denotes the total number of non-zero entries in the constraint matrix.[1]

In this paper, we consider a very natural dynamic setting, where an adversary can update any entry of A, b or c. After each update, the goal is to obtain a new ϵ-approximation quickly. Although this question has been studied since the early 80's [OvL80, OvL81, Epp91, AEM92, Mat93], all the existing dynamic algorithms can only handle a very small number of variables such as 2 or 3 (but they work for general LPs). In contrast, significant attention [ACC+18, BHN19, BK21, GKKP17, Waj20] has recently been devoted to a special case of dynamic packing and covering LPs: the dynamic fractional matching and set cover problems. These new algorithms, however, are highly specific and do not extend to general packing-covering LPs. For the latter, the current best dynamic algorithm just recomputes a solution from scratch after each update.

1.1 Our Results

We resolve the complexity of dynamic algorithms for general packing-covering LPs, up to a polylogarithmic factor in the update time. The following are the two key take-home messages from this paper. (1) In partially dynamic settings, there are deterministic ϵ-approximation algorithms for this problem that have polylogarithmic amortized update times. (2) Both partially dynamic updates and amortized update times are necessary: without any of these conditions, the trivial algorithm which recomputes an ϵ-approximation to the input LP from scratch is essentially the best possible. Below, we explain our results more formally.

We say an update is *relaxing* $\mathbb{C}$ (and *restricting* $\mathbb{P}$) if u increases an entry of A or decreases an entry of b, c. Note that OPT can only decrease due to a relaxing update. In contrast, an update is *restricting* $\mathbb{C}$ (and *relaxing* $\mathbb{P}$) if it decreases an entry of A or increases an entry of b, c. A sequence of updates to an LP is *partially dynamic* if either {all of them are relaxing} or {all of them are restricting}. Otherwise, the sequence of updates is *fully dynamic*. Let N denote the *maximum* number of non-zero entries in the input LP throughout the sequence of updates. Our main algorithmic result is the following:

Theorem 1.1. *We can deterministically maintain an ϵ-approximation to a packing-covering LP going through t partially dynamic updates in $\tilde{O}\left(N/\epsilon^3 + t/\epsilon\right)$ total update time. Thus, the amortized update time of our algorithm is $\tilde{O}(1/\epsilon^3)$.*[2]

---

[1] $\tilde{O}(\cdot)$ notation hides a $\text{poly}(\log(\frac{NW}{\epsilon}))$ factor, where W is the ratio between the largest and smallest nonzero entries in the LP.
[2] The total update time is defined to be the preprocessing time plus the total time taken to handle all updates. If an algorithm has total update time $O(N\tau_1 + t\tau_2)$, then its amortized update time is $O(\max\{\tau_1, \tau_2\})$.

By generality of packing and covering LPs, several applications immediately follow. For example, Theorem 1.1 implies the first decremental $(1 + \epsilon)$-approximation algorithm for bipartite maximum matching in *weighted* graphs in $\mathrm{poly}(\log(n)/\epsilon)$ amortized update time. Previously, the best known algorithms either took $\Omega(\sqrt{m})$ update time [GP13] or only worked for unweighted graphs [BGS20, JJST22, ABD22]. Theorem 1.1 also implies near-optimal incremental/decremental algorithms for maintaining near-optimal fractional solutions for set covers and dominating sets [GKKP17, HIPS19]. Generally, Theorem 1.1 reduces every dynamic problem that can be modeled as a packing/covering LP to a dynamic *rounding problem*. Since rounding LPs is one of the most successful paradigms in designing approximation algorithms in the static setting, we expect that our result will find many other future applications in dynamic algorithms.

We complement Theorem 1.1 by giving strong conditional lower bounds. Assuming SETH,[3] we show that a $N^{o(1)}$-approximation algorithm which either handles fully dynamic updates or guarantees worst-case update time, must have an update time of $\Omega(N^{1-o(1)})$. As we can solve the problem from scratch after every update in $\tilde{O}(N)$ time, our lower bound rules out any non-trivial dynamic algorithm in these settings.

Theorem 1.2. *Under SETH, there exists a $\beta = N^{o(1)}$ such that any dynamic algorithm $\mathcal{A}$ that maintains a β-approximation to the optimal objective of a packing or covering LP $\mathbb{P}$ satisfies the following conditions.*

- *If $\mathbb{P}$ undergoes fully dynamic updates, then $\mathcal{A}$ must have $\Omega(N^{1-o(1)})$ amortized update time.*

- *If $\mathbb{P}$ undergoes partially dynamic updates, then $\mathcal{A}$ must have $\Omega(N^{1-o(1)})$ worst-case update time.*

Our lower bounds hold even if the dynamic algorithm only *implicitly* maintains the solution, meaning that it only maintains an approximation to the value of OPT. Furthermore, they hold even if all the updates are on the entries of the constraint matrix A. Thus, Theorem 1.1 and Theorem 1.2 essentially settle the complexity of the dynamic packing-covering LP problem, up to a polylogarithmic factor in update time.

1.2 Techniques

To prove our main result (Theorem 1.1), we initiate a systematic study of the *multiplicative weight update* (MWU) framework in the dynamic setting. MWU is one the most versatile iterative optimization framework in theoretical computer science. There is a big conceptual barrier, however, in extending this framework to the dynamic setting. We start by explaining how to overcome this conceptual barrier.

1.2.1 MWU in the dynamic setting: The main challenge

Like other iterative methods, a MWU-based algorithm builds its solution x in iterations. At each iteration t, the solution $x^{(t)}$ heavily depends on the previous solution $x^{(t-1)}$. So when we update our original input, we expect that the update affects the solution $x^{(1)}$ from the first iteration. Then, the change in $x^{(1)}$ affects $x^{(2)}$, which in turn affects $x^{(3)}$ and so on. There are at least logarithmic many iterations and eventually the final solution might completely change. Intuitively, this propagation of changes suggests that using MWU in the dynamic setting is a dead end. How do we overcome this strong conceptual barrier?

As a take-home message, we show that for MWU there is *no* propagation of changes under partially dynamic updates! To be a bit more concrete, suppose that the current solution is $x^{(t)}$ and now the adversary updates the input. As long as the updates are partially dynamic, we can argue that all the previous solutions $x^{(1)}, \ldots, x^{(t-1)}, x^{(t)}$ remain valid with the new input. Our only task is to compute a valid $x^{(t+1)}$ for the updated input, which is much more manageable. To understand why this is possible, we now explain how MWU works at a high level.

---

[3]The Strong Exponential Time Hypothesis [IPZ01] says that there is no algorithm for solving k-CNF SAT with n variables for all k in $2^{n-\Omega(1)}$ time.

The explanation below will be generic (i.e., it will hold beyond packing-covering LPs). But for concreteness, consider a feasibility problem where given a matrix $C \in \{0,1\}^{m \times n}$, we need to find $x \in \mathbb{R}^n_{\geq 0}$ where $\mathbb{1}^\top x \leq 1$ and $Cx \geq \mathbb{1}$. Its dual problem is to find $y \in \mathbb{R}^m_{\geq 0}$ where $\mathbb{1}^\top y \geq 1$ and $C^\top y \leq \mathbb{1}$. MWU-based algorithms can be viewed as two-player games where the two players, which we call a *Whack-a-Mole* player and a *Greedy* player, together maintain a pair (x, y) using the following simple outline:

1. **Whack-a-Mole player builds a multiplicative solution:** Start with $x = \mathbb{1}/n$. If x violates some constraint $i \in [m]$ (i.e. $C_i x < 1$), then *multiplicatively* update x according to i (i.e. $x_j \leftarrow (1 + \epsilon)x_j$ for all j where $C_{ij} = 1$). We refer to this operation as *whacking* constraint i. Whenever x satisfies all constraints, terminate the game. Basically, this player keeps "whacking" violated constraints.

2. **Greedy player builds an additive solution:** Start with the zero vector $y = \mathbf{0}$. If there exists a *cheap* coordinate $i \in [m]$, then *additively* increment y_i by one. (In our case, coordinate i is cheap iff $C_i x < 1$.)[4] Terminate the game after T rounds, for some T, and return the average $y^* = y/T$. Basically, this player keeps greedily "incrementing" in a cheap direction.

The game terminates when: either (1) x satisfying all constraints, or (2) via the regret minimization property of MWU [AHK12], we can show that y^* is approximately feasible (i.e., $\mathbb{1}^\top y^* = 1$ and $C^\top y^* \leq (1 + \epsilon)\mathbb{1}$).

Coming back to the existing literature on MWU, we observe that the known algorithms can be classified into one of two categories. The algorithms in the first category take the greedy player's perspective and build their solutions y additively, while updating the weight x multiplicatively as a helper. These include algorithms for flow-related problems [Fle00, GK02, GK07], packing-covering LPs [PST95, KY14, You14, CQ18, Qua20], or constructions of pseudorandom objects [Bsh16]. In contrast, the algorithms in the second category take the whack-a-mole player's perspective and build their solutions x multiplicatively. These include algorithms for learning and boosting [Lit88, FS97, FS99]. The idea is also implicit in algorithms for geometric set covers [Cla93, BG95, AP14, CH20] and explicit in algorithms for online set covers [GKL21].

Now, the high-level explanation as to why MWU works under partially dynamic updates is simple. Suppose that the updates are restricting only. Then we take the whack-a-mole player's perspective. All we do is whacking violated constraints. Given an update, the constraints that we whacked in the past would still be violated at that time because the update is restricting. So our past actions are still valid. On the other hand, if the updates are relaxing only, then we take the greedy player's perspective. Given a relaxing update, all the cheap directions that we incremented in the past remain cheap at the present moment as well. This explains why all the past actions that we committed to are still valid. Hence, we only need to ensure that the next action is valid with respect to the updated input.

As the reasoning above is very generic, we expect that this insight will lead to further applications. For now, we discuss how we efficiently carry out this approach for packing-covering LPs.

1.2.2 Width-Independent Packing Covering LP Solvers via Black-box Regret Minimization

Existing static ϵ-approximation algorithms for packing-covering LPs can be grouped into two types: *width-dependent* and *width-independent* solvers. The width-dependent solvers have running time depending on $\lambda \cdot \text{OPT}$, where λ is the largest entry in the LP (e.g. $\tilde{O}(N(\frac{\lambda \text{OPT}}{\epsilon})^2)$ [PST95] and $\tilde{O}(N\frac{\lambda \text{OPT}}{\epsilon^2})$ [AHK12]). While the analysis in [PST95] is adhoc, [AHK12] gives a very insightful explanation as to how the guarantee of these solvers follows in a black-box manner from the well-established regret minimization property of MWU in the *experts setting*. This undesirable dependency on λOPT in the runtime led to an extensive line of work on width-independent solvers [ZO15, KY14, You14, CQ18, Qua20]. These stronger solvers run in $\tilde{O}(N/\text{poly}(\epsilon))$ time, which is independent of λOPT, and many of them are based on MWU.

---

[4]For readers familiar with the MWU literature, this is the "oracle" problem. Given weight x, we need to find a y' which satisfies the "average constraint" $x^\top C^\top y' \leq x^\top \mathbb{1}$. Then, we set $y \leftarrow y + y'$.

There is, however, one curious and unsatisfactory aspect in the literature on width-independent LP solvers. Namely, each of these MWU-based algorithms requires a separate and fine-tuned analysis involving calculations that resemble the proof of the regret minimization property of MWU. But can we just apply the regret bound of MWU in a blackbox manner while analyzing a width-indepdendent LP solver?

We design a new static algorithm for packing-covering LPs that answers this question in affirmative. To the best of our knowledge, it is the first width-independent near-linear-time LP solver whose analysis follows from a black-box application of the regret bound for MWU. This is explained in details in Section 2.1. Moreover, the algorithm is very intuitive and its high-level description is as follows:

> There are $O(\log(n)/\epsilon^2)$ phases. During each phase, we loop over the constraints: if constraint i is violated, then we "whack" it until it is satisfied and move on to the next constraint $i + 1$. Whenever $\mathbb{1}^\top x > (1 + \epsilon)$, we start the next phase and normalize x so that $\mathbb{1}^\top x = 1$. If we finish the loop, then we return x as an ϵ-approximation to the primal. After the last phase, we are guaranteed to obtain a dual ϵ-approximation y certifying that there is "almost" no feasible primal solution, i.e., no x where $\mathbb{1}^\top x \le 1$ and $Cx \ge (1 + \epsilon) \cdot \mathbb{1}$.

Henceforth, we refer to this algorithm as the *whack-a-mole MWU algorithm*.[5] The algorithm takes near-linear time because it has few phases and each phase requires only one-pass scan. In more details, we refer to the operation of "whacking constraint i until it is satisfied" as *enforceing* constraint i. Enforcing a constraint i can be done via binary search in $\tilde{O}(N_i)$ time, where N_i is the number of non-zero entries in constraint i. As each constraint is scanned and enforced once per phase, the total running time is $\tilde{O}(\sum N_i/\epsilon^2) = \tilde{O}(N/\epsilon^2)$.

The simplicity of this algorithm not only leads us to efficient dynamic implementations (see Section 1.2.3), but also efficient streaming algorithms and online algorithms with recourse (see Section 1.3).

1.2.3 Dynamic Implementation

We can extend our whack-a-mole MWU algorithm from Section 1.2.2 to handle restricting updates, in the following natural manner. Suppose that the algorithm obtained a feasible solution x (because x satisfies all constraints during the one-pass scan). Now, consider a restricting update to constraint i, say to an entry C_{ij}. We just need to check whether x violates constraint i (i.e., if $C_i x < 1$). If yes, then we enforce constraint i (i.e., whack it until it is satisfied). No new constraint gets violated due to this enforcement. As in the static setting, we start the next phase whenever $\mathbb{1}^\top x > (1 + \epsilon)$. We proceed like this until we obtain a feasible solution x. Once the algorithm concludes that there is almost no feasible solution, then we are done because this will remain the case after future restricting updates. The correctness follows from the static algorithm.

We have to overcome an issue while analyzing the total update time: Each constraint is *not* enforced once per phase anymore (because constraint i can be updated repeatedly within a phase). Accordingly, we strengthen our *analysis* to show that, even under restricting updates, each constraint is still enforced $\tilde{O}(1/\epsilon^2)$ times. But the algorithm remains the same. This leads to a total update time of $\tilde{O}(N/\epsilon^2 + t)$, for t updates.

Our dynamic whack-a-mole MWU algorithm applies to covering LPs under restricting updates, and by duality, to packing LPs under relaxing updates. There is a technical challenge, however, that prevents us from extending this approach to packing (resp. covering) LPs under restricting (resp. relaxing) updates.

To handle this case, we instead consider another static $\tilde{O}(N/\epsilon^2)$-time algorithm that take the greedy player's perspective [You14, Qua20]. We call it the *greedy MWU algorithm*. Given a covering LP (or even a mixed packing-covering LP), this greedy MWU algorithm can naturally handle relaxing updates (as

---

[5]Our algorithm resembles the whack-a-mole-based algorithms for computing geometric set covers [Cla93, BG95, AP14, CH20]. However, these previous algorithms give $O(1)$-approximation, are specifically described for set covers, and use adhoc calculation related to MWU. Thus, our whack-a-mole MWU algorithm refines their approximation, generalizes their applicability, and modularizes their analysis. Furthermore, for readers familiar with the algorithms of [You14, Qua20], our whack-a-mole MWU algorithms can be interpreted as implementing their algorithms from the whack-a-mole player's perspective.

discussed in Section 1.2.1). Using similar ideas, we show how to dynamize it with only polylogarithmic overhead in the total update time. Therefore, we obtain algorithms for covering LPs under relaxing updates, and by duality, for packing LPs under restricting update via duality.[6]

Discussion on the Simplicity of Our Techniques. We show how to seamlessly use the MWU framework to obtain near-optimal partially dynamic algorithms for packing-covering LPs. We view the simplicity of our approach as an important merit of this paper. Despite its simplicity, our result implies immediate applications such as a near-optimal decremental maximum *weight* matching algorithm, while previous algorithms [BGS20, JJST22] for this problem require more involved arguments and work only on unweighted graphs.

1.3 By-Products of Our Techniques

Our techniques turn out to have implications beyond dynamic algorithms for packing-covering LPs.

Online Algorithms with Recourse. The online covering LP problem was introduced in the influential work of Buchbinder and Naor [BN09a]. This is a fundamental problem in the literature on the online primal-dual method; with applications in online set cover, routing, ad-auctions, metrical task systems and many other settings (see the book [BN+09b]). Several variants of the problem are studied in [GN12, ABFP13, GKL21].

In this problem, we have a covering LP $\mathbb{C}$ with m constraints and n variables. The constraints of $\mathbb{C}$ arrive in an online fashion one after another, and we need to maintain a solution to $\mathbb{C}$. Crucially, the values of the variables can only increase through time (i.e., any decision made cannot be retracted). An online covering LP algorithm is α-competitive if it maintains a solution to $\mathbb{C}$ whose cost is always at most $\alpha \cdot \text{OPT}$. Buchbinder and Naor gave a tight $\Theta(\log n)$-competitive algorithm for this problem.

Can we beat the lower bound on competitive ratio for a given problem by allowing some amount of *recourse*? Here, recourse is defined to be the number of times the algorithm "retracts" its decision. An influential line of work in online algorithms [BHR18, GGK13, GK14, GKS14, GKKP17] has been devoted towards answering this question for various fundamental problems. For online covering LPs, recourse equals the number of times the value of any variable is decreased. Although covering LPs play a central role in online algorithms, until now it was not known whether we can beat the $\Omega(\log n)$ lower bound on the competitive ratio of this problem if we allow for small recourse. We answer this question in the affirmative.

Theorem 1.3. *There is a $(1+\epsilon)$-competitive algorithm for online covering LPs with $O(n \log(n) \log(nW)/\epsilon^3)$ total recourse. In fact, in this algorithm each variable incurs at most $O(\log(n) \log(nW)/\epsilon^3)$ recourse.*

Note that the recourse bound is completely independent of the number of constraints. Theorem 1.3 follows almost immediately from our whack-a-mole MWU algorithm from Section 1.2.2. Interestingly, our algorithm can be viewed as a small adjustment to the well-known $O(\log n)$-competitive algorithm of [BN09a]. To see this, suppose that we know OPT by guessing. Then here is the description of our algorithm.

> Run the algorithm of [BN09a]. Whenever the objective of the solution exceeds $(1 + \epsilon)\text{OPT}$, scale down the solution by a factor of $(1 + \epsilon)$ and continue.

Our argument shows that we will never scale down too many times. The algorithm is clearly $(1 + \epsilon)$-competitive. In the paper, we also show similar $(1 + \epsilon)$-competitive algorithm for online packing LPs. Curiously, online algorithms *without* recourse of this problem is not known.[7]

---

[6]Since the greedy MWU algorithm can handle mixed packing-covering LPs under relaxing updates, via duality this implies algorithms for handling restricting updates to both covering and packing LPs. So strictly speaking, our dynamic whack-a-mole MWU algorithm is subsumed by the dynamic greedy MWU algorithm. Nevertheless, we present the dynamic whack-a-mole MWU algorithm as it is simpler to understand and has more modular analysis.

[7]Many papers in online algorithms consider a problem where the *variables* of a packing LP are revealed one by one, which is a dual of the online covering LP problem. We consider a different setting, where the packing *constraints* arrive one after another.

Streaming algorithms. Our whack-a-mole MWU algorithm also works in the streaming setting. When the rows of the constraint matrix arrive one by one, we obtain an ϵ-approximation for both packing and covering LPs using $O(n)$ space and $\tilde{O}(1/\epsilon^3)$ passes. In contrast, when the columns of the constraint matrix arrive one by one, we get the same result but with a space complexity of $O(n + m)$. These give the first streaming algorithms for general packing covering LPs. Previous algorithms in the literature worked only for special cases such as fractional set cover [IMR+17] and fractional matching [AG11, AG18, AJJ+22].

Mixed Packing-Covering LPs. In a *mixed packing-covering* LP $\mathbb{P}$ (or *positive* LP for short), we are given $A_c \in \mathbb{R}_{\geq 0}^{m_c \times n}, b_c \in \mathbb{R}_{\geq 0}^{m_c}$ and $A_p \in \mathbb{R}_{\geq 0}^{m_p \times n}, b_p \in \mathbb{R}_{\geq 0}^{m_p}$, and we need to find $x \in \mathbb{R}_{\geq 0}^n$ such that $A_c x \geq b_c$ and $A_p x \leq b_p$. This is a generalization of both packing and covering LPs. An x is an ϵ-approximation to this positive LP if $A_c x \geq b_c/(1 + \epsilon)$ and $A_p x \leq (1 + \epsilon)b_p$. Our dynamic greedy MWU algorithm can handle positive LPs under relaxing updates.

Theorem 1.4. *We can deterministically maintain an ϵ-approximation to a positive LP undergoing t relaxing updates in $\tilde{O}\left(N/\epsilon^3 + t/\epsilon\right)$ total update time. Hence, the amortized update time of our algorithm is $\tilde{O}(1/\epsilon^3)$.*

Theorem 1.4 immediately gives the following natural applications captured by mixed packing-covering LPs. For the well-studied load balancing problem (e.g., [HLLT06, ABL20]), we obtain a near-optimal decremental algorithm for maintaining $(1 + \epsilon)$-approximate fractional assignments, where each update deletes a job. By duality, for the dynamic densest subhypergraph problem [HWC17, BBCG22], this implies a near-optimal decremental algorithm for maintaining $(1 + \epsilon)$-approximation, where each update deletes a hyperedge. The near-optimal algorithms for both problems were not known before.

Finally, we leave the question of designing a dynamic algorithm for positive LPs under restricting updates as an interesting open problem.

1.4 Roadmap for the Rest of the Paper

In Section 2, we present our whack-a-mole MWU algorithm and its extensions to dynamic, streaming and online settings. Section 3 gives an overview of the static greedy MWU algorithm for positive LPs [You14, Qua20], and then explains how we extend this algorithm to the setting where the input LP undergoes relaxing updates. Finally, in Section 4, we present our conditional lower bounds for dynamic packing-covering LPs.

If the reader wishes to treat this as an extended abstract, then we recommend reading until the end of Section 2.2 as that contains the main technical result of this paper.

2 The Whack-a-Mole MWU Algorithm

This section focuses on the whack-a-mole MWU algorithm. We start by considering the following problem.

Problem 2.1. *Given a matrix $C \in [0, \lambda]^{m \times n}$ where $\lambda > 0$, either return a vector $x \in \mathbb{R}_{\geq 0}^n$ with $\mathbb{1}^\top x \leq 1 + \Theta(\epsilon)$ and $Cx \geq (1 - \Theta(\epsilon)) \cdot \mathbb{1}$, or return a vector $y \in \mathbb{R}_{\geq 0}^m$ with $\mathbb{1}^\top y \geq 1 - \Theta(\epsilon)$ and $C^\top y \leq (1 + \Theta(\epsilon)) \cdot \mathbb{1}$.*

Problem 2.1 corresponds to the following covering LP: Minimize $\mathbb{1}^\top x$, s.t. $Cx \geq \mathbb{1}$ and $x \in \mathbb{R}_{\geq 0}^n$. We have to either return an approximately feasible solution to this covering LP with objective $\leq 1 + \Theta(\epsilon)$, or return an approximately feasible solution to the dual packing LP with objective $\geq 1 - \Theta(\epsilon)$. In Sections 2.1, 2.2, 2.3 and 2.4, we respectively show that our MWU algorithm solves Problem 2.1 in static, dynamic, streaming and online settings. Subsequently, in Section 2.5, we explain how we can compute an ϵ-approximate optimal solution to a general packing-covering LP if we have an algorithm for Problem 2.1.

Throughout this section, an index $i \in [m]$ (resp. $j \in [n]$) refers to a row (resp. column) of the matrix C, and $C_{ij} \in [0, \lambda]$ denotes the entry corresponding to the ith row and j^{th} column of C. Furthermore, the symbol v_k denotes the kth coordinate of a vector $v \in \mathbb{R}^n$, where $k \in [n]$.

2.1 Static Whack-a-Mole MWU Algorithm

We now present the whack-a-mole MWU algorithm for Problem 2.1 in the static setting. We describe the basic template behind the algorithm in Section 2.1.1. In Section 2.1.3, we show how to implement this basic template in near-linear time.
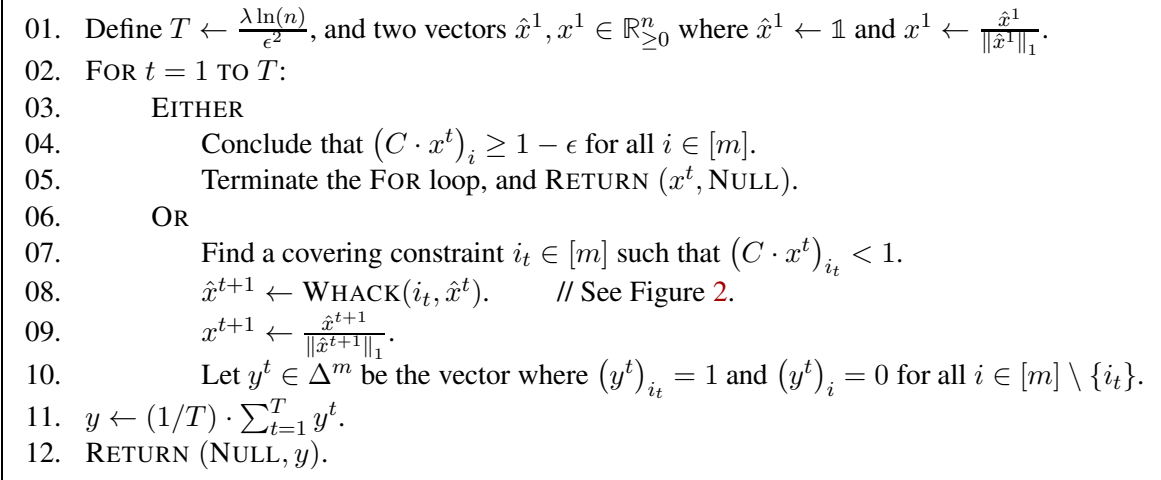
2.1.1 The Basic Template

---

01.   Define $T \leftarrow \frac{\lambda \ln(n)}{\epsilon^2}$, and two vectors $\hat{x}^1, x^1 \in \mathbb{R}_{\geq 0}^n$ where $\hat{x}^1 \leftarrow \mathbb{1}$ and $x^1 \leftarrow \frac{\hat{x}^1}{\|\hat{x}^1\|_1}$.
02.   FOR $t = 1$ TO T:
03.   EITHER
04.   Conclude that $(C \cdot x^t)_i \geq 1 - \epsilon$ for all $i \in [m]$.
05.   Terminate the FOR loop, and RETURN (x^t, NULL).
06.   OR
07.   Find a covering constraint $i_t \in [m]$ such that $(C \cdot x^t)_{i_t} < 1$.
08. $\hat{x}^{t+1} \leftarrow \text{WHACK}(i_t, \hat{x}^t)$.   // See Figure 2.
09. $x^{t+1} \leftarrow \frac{\hat{x}^{t+1}}{\|\hat{x}^{t+1}\|_1}$.
10.   Let $y^t \in \Delta^m$ be the vector where $(y^t)_{i_t} = 1$ and $(y^t)_i = 0$ for all $i \in [m] \setminus \{i_t\}$.
11. $y \leftarrow (1/T) \cdot \sum_{t=1}^{T} y^t$.
12.   RETURN (NULL, y).

Figure 1: The Whack-a-Mole MWU Algorithm

---

1. FOR ALL $j \in [n]$:
2. $\hat{z}_j \leftarrow \left(1 + \epsilon \cdot \frac{C_{ij}}{\lambda}\right) \cdot \hat{x}_j$.
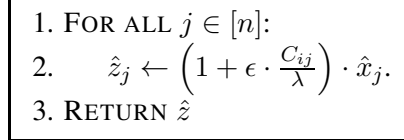3. RETURN $\hat{z}$

Figure 2: WHACK$(i, \hat{x})$.

The algorithm is described in Figure 1 and Figure 2. It maintains a vector $\hat{x} \in \mathbb{R}_{\geq 0}^n$, where $\hat{x}_j$ denotes the *weight* associated with a variable $j \in [n]$ in the covering LP, and the normalized vector $x := \hat{x}/\|\hat{x}\|_1$. This ensures that $\mathbb{1}^\top \cdot x = 1$. The algorithm runs in $T = \lambda \ln(n)/\epsilon^2$ iterations, where $0 < \epsilon < 1/2$.

Let $\hat{x}^t$ and x^t respectively denote the status of $\hat{x}$ and x at the start of iteration $t \in [T]$ of the main FOR loop in Figure 1. Before the very first iteration, we initialize $\hat{x}^1 \leftarrow \mathbb{1}$. Subsequently, during any given iteration $t \in [T]$, the algorithm branches into one of the following two cases.

Case (1). It observes that $Cx^t \geq (1-\epsilon) \cdot \mathbb{1}$ and returns (x^t, NULL). In this case, $x^t \in \mathbb{R}_{\geq 0}^n$ is an approximately feasible solution to the covering LP, with objective $\mathbb{1}^\top x^t = 1$.

Case (2). It identifies a violated covering constraint $i_t \in [m]$ with $(Cx^t)_{i_t} < 1$. It then *whacks* constraint i_t, by setting $\hat{x}_j \leftarrow \left(1 + \epsilon \cdot \frac{C_{i_t j}}{\lambda}\right) \cdot \hat{x}_j$ for all $j \in [n]$, and accordingly updates the normalized vector x. Note that this step increases the relative importance (in the solution x^t) of the coordinates $j \in [n]$ that have large $C_{i_t j}$ values. Thus, intuitively, whacking a violated covering constraint makes progress towards making the solution x^t feasible for the covering LP. We let $y^t \in \Delta^m$ denote the *indicator* vector for the covering constraint $i_t \in [m]$ that gets whacked.

7

After T iterations, the algorithm returns (NULL, y), where y is the average of the vectors $y^1, \cdots, y^T$. Clearly, we have $\mathbb{1}^\top y = 1$. The next lemma shows that y is an approximately feasible solution to the dual packing LP.[8] Its proof appears in Section 2.1.2.

Lemma 2.2. *Suppose that the algorithm in Figure 1 returns* (NULL, y) *in step (12). Then* $C^\top y \leq (1+4\epsilon)\cdot\mathbb{1}$.

Theorem 2.3. *The algorithm in Figure 1 either returns an* $x^t \in \mathbb{R}^n_{\geq 0}$ *with* $\mathbb{1}^\top x^t = 1$ *and* $Cx^t \geq (1 - \epsilon) \cdot \mathbb{1}$, *or it returns a* $y \in \mathbb{R}^m_{\geq 0}$ *with* $\mathbb{1}^\top y = 1$ *and* $C^\top y \leq (1 + 4\epsilon) \cdot \mathbb{1}$.

Proof. Follows from Lemma 2.2 and the preceding discussion. □

2.1.2 Proof of Lemma 2.2

We start by recalling the *experts* setting [AHK12]. We have n experts $\{1, \ldots, n\}$, who participate the following process that goes on for T rounds. At the start of round $t \in [T]$, we have a weight vector $w^t \in \mathbb{R}^n_{\geq 0}$, where $\left(w^t\right)_j$ denotes the weight associated with expert $j \in [n]$. Initially, before round 1 begins, we set $w^t = \mathbb{1}$. These weights define a distribution $\mathcal{D}^t$ over the experts, where the probability of picking expert $j \in [n]$ is given by $\frac{\left(w^t\right)_j}{\|w^t\|_1}$. At the start of round $t \in [T]$, our algorithm picks an expert $j \in [n]$ from this distribution $\mathcal{D}^t$. Subsequently, nature reveals a payoff vector $p^t \in [-1, 1]^n$, where $\left(p^t\right)_j \in [-1, 1]$ is the payoff for expert $j \in [n]$ in round t. Based on these payoffs, the algorithm then updates the weights of the experts for the next round, by setting $w^{t+1}_j \leftarrow \left(1 + \epsilon \cdot \left(p^t\right)_j\right) \cdot w^t_j$ for all $j \in [n]$.

The lemma below, which is identical to Theorem 2.5 in [AHK12], bounds the total expected payoff obtained by the algorithm in this setting, in terms of the total payoff obtained by any fixed expert.

Lemma 2.4. *For all experts* $j \in [n]$, *we have:*

$$\sum_{t=1}^T \left(p^t\right)^\top \cdot \frac{w^t}{\|w^t\|_1} \geq \sum_{t=1}^T \left(p^t\right)_j - \sum_{t=1}^T \epsilon \cdot \left|\left(p^t\right)_j\right| - \frac{\ln(n)}{\epsilon}.$$

We now map the whack-a-mole algorithm to the experts setting as follows. Each covering constraint $j \in [n]$ corresponds to an expert, each iteration of the FOR loop in Figure 1 corresponds to a round, the vector $\hat{x}^t$ corresponds to the weight vector w^t at the start of round $t \in [T]$, and finally, the payoff for an expert $j \in [n]$ in round $t \in [T]$ is given by $\left(p^t\right)_j := (1/\lambda) \cdot C_{i_t, j}$.

Since $C_{i_t, j} \in [0, \lambda]$, we have $\left|\left(p^t\right)_j\right| = \left(p^t\right)_j$ for all $j \in [n], t \in [T]$. Thus, Lemma 2.4 implies that:

$$\sum_{t=1}^T \left(p^t\right)^\top \cdot x^t \geq \sum_{t=1}^T (1 - \epsilon) \cdot \left(p^t\right)_j - \frac{\ln(n)}{\epsilon}, \text{ for all experts } j \in [n]. \tag{2.1}$$

Diving both sides of the above inequality by T, and then rearranging the terms, we get:

$$\frac{(1 - \epsilon)}{T} \cdot \sum_{t=1}^T \left(p^t\right)_j \leq \frac{1}{T} \cdot \sum_{t=1}^T \left(p^t\right)^\top \cdot x^t + \frac{\ln(n)}{\epsilon \cdot T}, \text{ for all experts } j \in [n]. \tag{2.2}$$

We next upper bound the right hand side (RHS) of (2.2). Since the algorithm picks a violated covering constraint to whack in each round, we have: $\left(p^t\right)^\top \cdot x^t = (1/\lambda) \cdot \left(Cx^t\right)_{i_t} \leq (1/\lambda)$. Taking the average of

---

[8]We remark here that this basic algorithm will work even if the matrix C had negative entries (i.e., if we had $C \in [-\lambda, \lambda]^{m \times n}$), provided we increase the number of iterations by setting $T = \frac{\lambda^2 \ln(n)}{\epsilon^2}$.

this inequality across all the T rounds, we get: $(1/T) \cdot \sum_{t=1}^{T} \left(p^t\right)^{\top} \cdot x^t \le (1/\lambda)$. Since $T = \lambda \ln(n)/\epsilon^2$, we derive the following upper bound on the RHS of (2.2).

$$\frac{1}{T} \cdot \sum_{t=1}^{T} \left(p^t\right)^{\top} \cdot x^t + \frac{\ln(n)}{\epsilon \cdot T} \le \frac{1}{\lambda} \cdot (1 + \epsilon). \tag{2.3}$$

We now focus our attention on the left hand side (LHS) of (2.2). Fix any expert $j \in [n]$. We first express the payoff obtained by this expert at round $t \in [T]$ in terms of the vector y^t, and get: $\left(p^t\right)_j = \frac{1}{\lambda} \cdot C_{i_t, j} = \frac{1}{\lambda} \cdot \left(C^{\top} y^t\right)_j$. Since $y = \frac{1}{T} \cdot \sum_{t=1}^{T} y^t$, the average payoff for the expert j across all the T rounds is given by:

$$\frac{1}{T} \cdot \sum_{t=1}^{T} \left(p^t\right)_j = \frac{1}{T} \cdot \frac{1}{\lambda} \cdot \sum_{t=1}^{T} \left(C^{\top} y^t\right)_j = \frac{1}{\lambda} \cdot \left(C^{\top} y\right)_j. \tag{2.4}$$

From (2.2), (2.3) and (2.4), we get:

$$\frac{(1 - \epsilon)}{\lambda} \cdot \left(C^{\top} y\right)_j \le \frac{1}{\lambda} \cdot (1 + \epsilon), \text{ and hence } \left(C^{\top} y\right)_j \le (1 + 4\epsilon) \text{ for all } j \in [n].$$

The last inequality holds since $\epsilon < 1/2$. This concludes the proof of Lemma 2.2.

2.1.3 A Near-Linear Time Implementation

---

01. $\hat{x}^1 \leftarrow \mathbb{1}$, $t \leftarrow 1$, and $T \leftarrow \lambda \ln(n)/\epsilon^2$.
02. LOOP
03. $W \leftarrow \left\|\hat{x}^t\right\|_1$.
04.   FOR ALL $i \in [m]$:
05.   IF $\left(C \cdot \frac{\hat{x}^t}{W}\right)_i < 1 - \epsilon/2$ THEN
06. $\delta \leftarrow$ ENFORCE$(i, t, \hat{x}^t, W)$.   // See Figure 4.
07. $t \leftarrow t + \delta$.
08.   IF $t = T$, THEN
09.   Terminate the LOOP and return (NULL, y), where $y := (1/T) \cdot \sum_{t'=1}^{T} y^{t'}$.
10.   IF $\left\|\hat{x}^t\right\|_1 > (1 - \epsilon/2)^{-1} \cdot W$, THEN
11.   GO TO step (03).   // Initiate a new phase.
12.   Terminate the LOOP and return (x^t, NULL), where $x^t := \frac{\hat{x}^t}{\left\|\hat{x}^t\right\|_1}$.

Figure 3: An Implementation of the Whack-a-mole MWU Algorithm.

We now show how to implement the whack-a-mole MWU algorithm from Section 2.1.1 in near-linear time. This implementation is outlined in Figure 3, and is based on two key ideas. (1) We split up the working of the Whack-a-mole MWU algorithm into *phases*, and ensure that within any given phase the total weight $\left\|\hat{x}^t\right\|_1$ of all the experts (see the discussion in the beginning of Section 2.1.2) changes by at most a $(1 - \epsilon/2)^{-1}$ multiplicative factor. (2) The weight of an expert $j \in [n]$ can only increase during the course of the algorithm. Because of this, within a given phase we only need to consider each constraint at most once, provided upon considering the constraint we keep repeatedly whacking it until it gets satisfied.

At the start of a new phase, we set $W \leftarrow \left\|\hat{x}^t\right\|_1$. Throughout the phase, the value of W will not change and it will serve as an estimate of the total weight of the experts. Since W remains within a multiplicative $(1 - \epsilon/2)^{-1}$ factor of $\left\|\hat{x}^t\right\|_1$, the vector $\frac{\hat{x}^t}{W}$ remains a good estimate of the actual solution $x^t := \frac{\hat{x}^t}{\left\|\hat{x}^t\right\|_1}$.

1. $\delta \leftarrow$ STEP-SIZE$(i, t, \hat{x}^t, W)$.   // See Figure 5.
2. FOR $t' = t$ to $(t + \delta - 1)$:
3. $\hat{x}^{t'+1} \leftarrow$ WHACK$(i, \hat{x}^{t'})$.   // See Figure 2.
4. $i_{t'} \leftarrow i$.
5.   Let $y^{t'} \in \Delta^m$ be the vector where $\left(y^{t'}\right)_i = 1$ and $\left(y^{t'}\right)_{i'} = 0$ for all $i' \in [m] \setminus \{i\}$.
6. RETURN δ.

Figure 4: ENFORCE$(i, t, \hat{x}^t, W)$.

1. For every integer $\kappa \geq 1$, let $z^\kappa \in \mathbb{R}^n_{\geq 0}$ be such that $(z^\kappa)_j = \left(1 + \epsilon \cdot \frac{C_{ij}}{\lambda}\right)^\kappa \cdot \left(\hat{x}^t\right)_j$ for all $j \in [n]$.
2. If $\left(C \cdot \frac{z^{T-t}}{W}\right)_i < 1$, THEN
3. $\delta \leftarrow T - t$.
4. ELSE
5.   Using binary search, compute the the smallest integer $\delta \in [T - t]$ such that $\left(C \cdot \frac{z^\delta}{W}\right)_i \geq 1$.
6. Return δ

Figure 5: STEP-SIZE$(i, t, \hat{x}^t, W)$

During a given phase, we scan through all the covering constraints in any arbitrary order. We now explain how to implement a typical iteration of this scan, where (say) we are considering the constraint $i \in [m]$. We first check if $\left(C \cdot \frac{\hat{x}^t}{W}\right)_i \geq 1 - \epsilon/2$, that is, whether the constraint is approximately satisfied. If the answer is yes, then we do nothing with this constraint and proceed to the next iteration of the scan. In contrast, if the answer is no, then we *enforce* this constraint by calling the subroutine ENFORCE$(i, t, \hat{x}^t, W)$. This subroutine works as follows. It finds an integer $\delta \geq 1$ which indicates the minimum number of times the constraint $i \in [m]$ needs to be whacked before it gets satisfied, assuming that we continue to be in the same phase even at the end of all these whacks. Constraint i then repeatedly gets whacked δ times, and after that we set $t \leftarrow t + \delta$. At this point, if $t = T$, then we terminate the algorithm and return a solution for the dual packing LP, as in Section 2.1.1. Else if $\left\|\hat{x}^t\right\|_1 > (1 - \epsilon/2)^{-1} \cdot W$, because the total weight of the experts increased a lot due to the previous δ whacks, then we initiate a new phase.

If at the end of the scan, we observe that W is still an accurate estimate of the total weight $\left\|\hat{x}^t\right\|_1$ *and* $t < T$, then we terminate the algorithm and return $x^t := \frac{\hat{x}^t}{\|\hat{x}^t\|_1}$ as a solution to the primal covering LP.

Lemma 2.5. *The procedure described in Figure 3 implements the algorithm from Section 2.1.1.*

Proof. We refer to each call to WHACK$(i, \hat{x}^t)$ as a *round* (see Section 2.1.2). Both Figure 1 and Figure 3 initialize $\hat{x}^1 \leftarrow \mathbb{1}$ and run for at most $T = \lambda \ln(n)/\epsilon^2$ rounds. We will show that the way Figure 3 implements each round and the way it eventually returns a solution are both consistent with Figure 1.

Towards this end, first consider the scenario where the procedure in Figure 3 enforces a constraint $i \in [m]$ at the start of a round $t \in [T]$ by calling the subroutine ENFORCE$(i, t, \hat{x}^t, W)$, which repeatedly whacks the constraint δ times. From Figure 4 and Figure 5, we infer that

$$\left(C \cdot \frac{\hat{x}^{t'}}{W}\right)_i < 1 \text{ for all } t \leq t' \leq t + \delta - 1. \tag{2.5}$$

Since W is the total weight of all the experts at the start of the concerned phrase, and since the weight of

any expert can only increase with time, we have: $\left\|x^{t'}\right\|_1 \geq W$ for all $t' \geq t$. Hence, from (2.5) we get:

$$\left(C \cdot x^{t'}\right)_i < 1 \text{ for all } t \leq t' \leq t + \delta - 1, \text{ where } x^{t'} := \frac{\hat{x}^{t'}}{\|x^{t'}\|_1}. \tag{2.6}$$

This implies that the decisions to whack constraint $i \in [m]$ in successive rounds $t' \in [t, t + \delta - 1]$ are consistent with the rule governing the whacking of constraints in Figure 1.

Next, note that after enforcing the constraint $i \in [m]$, the procedure in Figure 3 sets $t \leftarrow t + \delta$. At this point, if $t = T$, then it decides to return the vector $y := (1/T) \cdot \sum_{t'=1}^T y^{t'}$ as a solution to the dual packing LP. Clearly, this decision is also consistent with steps $(11) - (12)$ in Figure 1.

Finally, consider the scenario where the procedure in Figure 3 returns a vector $x^{t''}$ in round t'' (see step (12) in Figure 3). Focus on the very last phase, which spans from (say) round t' to round t'', where $t' < t''$. Let W be the value of $\|\hat{x}^t\|_1$ at the start of this phase. Fix any constraint $i \in [m]$, which was considered (say) at the start of round $t_i \in [t', t'']$ by the FOR loop in Figure 3. Now, there are two possible cases.

Case I: The constraint i did not get enforced in this phase. This happens if $\left(C \cdot \frac{\hat{x}^{t_i}}{W}\right)_i \geq 1 - \epsilon/2$. Here, we derive that $\left(C \cdot \frac{\hat{x}^{t''}}{W}\right)_i \geq \left(C \cdot \frac{\hat{x}^{t_i}}{W}\right)_i \geq 1 - \epsilon/2$, since each co-ordinate of $\hat{x}$ can only increase over time.

Case II: The constraint i got enforced in this phase, by getting repeatedly whacked δ times starting from round t_i. Thus, we have $t_i + \delta < T$ (otherwise, the algorithm would return a dual packing solution y) and $\left(C \cdot \frac{\hat{x}^{t_i+\delta}}{W}\right)_i \geq 1$. Analogous to Case I, here we derive that $\left(C \cdot \frac{\hat{x}^{t''}}{W}\right)_i \geq \left(C \cdot \frac{\hat{x}^{t_i+\delta}}{W}\right)_i \geq 1 \geq 1 - \epsilon/2$.

To summarize, we have the following guarantee for every constraint $i \in [m]$ at the start of round t''.

$$\left(C \cdot \frac{\hat{x}^{t''}}{W}\right)_i \geq 1 - \epsilon/2. \tag{2.7}$$

Since no new phase got initiated just before round t'' (see steps $(10) - (11)$ in Figure 3), we infer that $\left\|\hat{x}^{t''}\right\|_1 \leq (1 - \epsilon/2)^{-1} \cdot W$. Thus, from (2.7), we get the following guarantee for every constraint $i \in [m]$.

$$\left(C \cdot x^{t''}\right)_i \geq \left(C \cdot \frac{\hat{x}^{t''}}{W}\right)_i \cdot (1 - \epsilon/2) \geq (1 - \epsilon/2)^2 \geq 1 - \epsilon, \text{ where } x^{t''} := \frac{\hat{x}^{t''}}{W}. \tag{2.8}$$

In other words, the vector $x^{t''}$ satisfies the inequality $C \cdot x^{t''} \geq (1 - \epsilon) \cdot \mathbb{1}$, and hence the decision to return $x^{t''}$ as a solution to the covering LP is also consistent with the template described in Figure 1. $\square$

Theorem 2.6. *The procedure in Figure 3 either returns an $x^t \in \mathbb{R}_{\geq 0}^n$ with $\mathbb{1}^\top x^t = 1$ and $Cx^t \geq (1 - \epsilon) \cdot \mathbb{1}$, or it returns a $y \in \mathbb{R}_{\geq 0}^m$ with $\mathbb{1}^\top y = 1$ and $C^\top y \leq (1 + 4\epsilon) \cdot \mathbb{1}$.*

Proof. Follows from Theorem 2.3 and Lemma 2.5. $\square$

It now remains to bound the running time of this algorithm. Towards this end, the lemma below analyzes the time taken by a call to the subroutine STEP-SIZE$(i, \hat{x}^t, W)$.

Lemma 2.7. *A call to STEP-SIZE$(i, t, \hat{x}^t, W)$ can be implemented in $O\left(N_i \cdot \log^2\left(\frac{\lambda \log(n)}{\epsilon}\right)\right)$ time, where N_i is the number of non-zero entries in row $i \in [m]$ of the matrix $C \in [0, \lambda]^{m \times n}$.*

Proof. It takes $O(N_i \cdot \log \kappa)$ time to compute $\left(C \cdot \frac{z^\kappa}{W}\right)_i$ for any $\kappa \geq 1$ (see step (1) of Figure 5). Furthermore, the value of $\left(C \cdot \frac{z^\kappa}{W}\right)_i$ can only increase as we increase κ. Thus, we can find δ, as in steps $(2) - (5)$ of Figure 5, by doing a binary search in $O(N_i \cdot \log T \cdot \log T)$ time. The lemma follows since $T = \lambda \ln(n)/\epsilon^2$. $\square$

The next lemma bounds the time taken to enforce a constraint.

Lemma 2.8. *A call to* ENFORCE$(i, t, \hat{x}^t, W)$ *can be implemented in* $O\left(N_i \cdot \log^2\left(\frac{\lambda \log(n)}{\epsilon}\right)\right)$ *time, where* N_i *is the number of non-zero entries in row* $i \in [m]$ *of the matrix* $C \in [0, \lambda]^{m \times n}$.

Proof. The key idea is that we do *not* need to *explicitly* store all the vectors $\hat{x}^{t+1}, \ldots, \hat{x}^{t+\delta}$ and $y^t, \ldots, y^{t+\delta-1}$. Instead, using standard data structures, the overall algorithm (outlined in Figure 3) keeps track of only the following information corresponding to a specific call to ENFORCE$(i, t, \hat{x}^t, W)$.

(1) The value of t at the start of the call, and the step-size δ computed by the call.

(2) The index $i \in [m]$ corresponding to the call. This index implicitly defines all the vectors $y^t, \ldots, y^{t+\delta-1}$.

(3) A vector $\hat{x} \in \mathbb{R}^n_{\geq 0}$, which was equal to $\hat{x}^t$ just before the call, and needs to be set to $\hat{x}^{t+\delta}$ at the end of the call. This will be required to implement the next iteration of the FOR loop in Figure 3.

The overall algorithm (in Figure 3) can easily recover the answer it needs to return, provided it keeps track of the above mentioned pieces of information corresponding to each call to the ENFORCE subroutine.

Thus, the runtime of a call to ENFORCE$(i, t, \hat{x}^t, W)$ is dominated by the time spent on the following two tasks: (1) computing the value of δ, and (2) ensuring that the vector $\hat{x}$, which was equal to $\hat{x}^t$ just before the call, equals $\hat{x}^{t+\delta}$ at the end of the call. The time needed for the former task is bounded by Lemma 2.7, whereas the time needed for the latter task is $O(N_i \cdot \log \delta)$. The lemma follows since $\delta \leq T = \frac{\lambda \ln(n)}{\epsilon^2}$. $\square$

We next bound the total number of phases in the algorithm.

Lemma 2.9. *Throughout the duration of the algorithm outlined in Figure 3, we have:* $\left\|\hat{x}^t\right\|_1 \leq n^{(1/\epsilon)}$.

Proof. Consider any round $t \in [T]$, which corresponds to an iteration of the FOR loop in Figure 1 (see Lemma 2.5). Observe that:

$$\left\|\hat{x}^{t+1}\right\|_1 - \left\|\hat{x}^t\right\|_1 = \sum_{j \in [n]} \left(\left(\hat{x}^{t+1}\right)_j - \left(\hat{x}^t\right)_j\right) = \sum_{j \in [n]} \left(\hat{x}^t\right)_j \cdot \left(\epsilon \cdot \frac{C_{i_t,j}}{\lambda}\right) = \frac{\epsilon}{\lambda} \cdot \left(C\hat{x}^t\right)_{i_t} < \frac{\epsilon}{\lambda} \cdot \left\|\hat{x}^t\right\|_1.$$

The last inequality follows since $\left(Cx^t\right)_{i_t} < 1$ and $x^t := \hat{x}^t / \left\|\hat{x}^t\right\|_1$. Rearranging the terms, we get:

$$\left\|\hat{x}^{t+1}\right\|_1 \leq \left(1 + \frac{\epsilon}{\lambda}\right) \cdot \left\|\hat{x}^t\right\|_1. \tag{2.9}$$

As $T = \frac{\lambda \ln(n)}{\epsilon^2}$ and $\left\|\hat{x}^0\right\|_1 = n$, from (2.9) we get: $\left\|\hat{x}^t\right\|_1 \leq \left(1 + \frac{\epsilon}{\lambda}\right)^T \cdot \left\|\hat{x}^0\right\|_1 \leq n^{(1/\epsilon)}$ for all $t \in [T]$. $\square$

Corollary 2.10. *The algorithm outlined in Figure 3 has at most* $O\left(\frac{\log(n)}{\epsilon^2}\right)$ *many phases.*

Proof. We initiate a new phase whenever $\left\|\hat{x}^t\right\|$ increases by a multiplicative factor of $(1 - \epsilon/2)^{-1}$. Hence, by Lemma 2.9, the number of phases is at most $O\left(\log_{(1-\epsilon/2)^{-1}} n^{(1/\epsilon)}\right) = O\left(\frac{\log n}{\epsilon^2}\right)$. $\square$

We are now ready to bound the total runtime of our algorithm.

Theorem 2.11. *The algorithm outlined in Figure 3 can be implemented in* $O\left(N \cdot \frac{\log(n)}{\epsilon^2} \cdot \log^2\left(\frac{\lambda \log(n)}{\epsilon}\right)\right)$ *time, where* N *is the total number of non-zero entries in the matrix* $C \in [0, \lambda]^{m \times n}$.

Proof. Consider a given phase of the algorithm. From Lemma 2.8, it follows that the time taken to enforce any specific constraint $i \in [m]$ in this phase is at most $O\left(N_i \cdot \log^2\left(\frac{\lambda \log(n)}{\epsilon}\right)\right)$. Since each constraint gets enforced at most once during this phase, the total time spent in this phase is at most:

$$O\left(\sum_{i \in [m]} N_i \cdot \log^2\left(\frac{\lambda \log(n)}{\epsilon}\right)\right) = O\left(N \cdot \log^2\left(\frac{\lambda \log(n)}{\epsilon}\right)\right).$$

The theorem now follows from Corollary 2.10. $\square$

2.2 Dynamic Whack-a-Mole MWU Algorithm for Covering LPs

In this section, we focus on designing a dynamic algorithm for Problem 2.1 in the following setting. At preprocessing, we receive a constraint matrix $C \in [0, \lambda]^{m \times n}$. Subsequently, the matrix C undergoes a sequence of *restricting* updates, where each update decreases the value of some entry C_{ij} of the matrix C.[9] Throughout this sequence of updates, we need to maintain either a vector $\tilde{x} \in \mathbb{R}_{\geq 0}^n$ with $\mathbb{1}^\top \tilde{x} \leq 1 + \Theta(\epsilon)$ and $C\tilde{x} \geq (1 - \Theta(\epsilon)) \cdot \mathbb{1}$, or a vector $y \in \mathbb{R}_{\geq 0}^m$ with $\mathbb{1}^\top y = 1$ and $C^\top y \leq (1 + \Theta(\epsilon)) \cdot \mathbb{1}$.

We will show that our implementation of the whack-a-mole MWU algorithm from Section 2.1.3 seamlessly extends to this dynamic setting. In more details, our dynamic algorithm works as follows.

Preprocessing: At preprocessing, we run the static algorithm from Section 2.1.3 on the input matrix C. Depending on its outcome, we consider one of the following two cases.

Case I: The static algorithm returns a vector $y \in \mathbb{R}_{\geq 0}^m$, as in step (09) of Figure 3. It follows that $\mathbb{1}^\top y = 1$ and $C^\top y \leq (1 + 4\epsilon) \cdot \mathbb{1}$ (see Theorem 2.6). In this case, the vector y will continue to remain an approximately feasible solution to the dual packing LP as the matrix C undergoes restricting entry updates in future. Hence, our dynamic algorithm terminates without having to process any update.

Case II: The static algorithm returns a vector $x^t := \hat{x}^t / \|\hat{x}^t\|_1$, as in step (12) of Figure 3. It follows that $\mathbb{1}^\top x^t = 1$ and $Cx^t \geq (1 - \epsilon) \cdot \mathbb{1}$ (see Theorem 2.6). Our dynamic algorithm, however, will explicitly maintain only the vector $\tilde{x}^t := \hat{x}^t / W$, which is a very good approximation to x^t since $W \leq \|\hat{x}^t\|_1 \leq (1 - \epsilon/2)^{-1} \cdot W \leq (1 + \epsilon) \cdot W$. Thus, in this case, after preprocessing our dynamic algorithm returns a vector $\tilde{x}^t$ which satisfies $\mathbb{1}^\top \tilde{x}^t \leq 1 + \epsilon$ and $C\tilde{x}^t \geq (1 - \epsilon) \cdot \mathbb{1}$.

Handling a restricting entry update to C: Consider an update which decreases the value of some entry C_{ij} of the matrix C. To handle this update, we simply run the steps (05) - (11) of Figure 3. In words, we observe that if any constraint in the covering LP gets violated due to this update, then it must be the constraint $i \in [m]$. Thus, we check whether the current solution $\tilde{x}^t := \hat{x}^t / W$ approximately satisfies constraint i. If not, then we enforce that constraint by repeatedly whacking it. At the end of this process, we end up in one of three possible cases. (Case 1): W is no longer an accurate estimate of $\|\hat{x}^t\|_1$. In this case, we initiate a new phase. (Case 2): $t = T$. Here, we return an approximately feasible solution y to the dual packing LP, and we terminate the dynamic algorithm since y remains a valid dual solution after any future update. (Case 3): If we are neither in Case 1 nor in Case 2, then we infer that $\tilde{x}^t$ is now an approximately feasible solution to the covering LP, with $\mathbb{1}^\top \tilde{x}^t \leq 1 + \epsilon$ and $C\tilde{x}^t \geq (1 - \epsilon) \cdot \mathbb{1}$. This holds because repeatedly whacking constraint i does not lead to any other constraint being violated, provided we remain in the same phase.

Rounds/phases: Before proceeding any further, we recall that a new *round* begins whenever we whack a constraint, and the variable t denotes the total number of rounds we have seen so far. In contrast, a new *phase* begins whenever $\|\hat{x}^t\|_1$ increases by a multiplicative factor of $(1 - \epsilon/2)^{-1}$.

It now remains to analyze our dynamic algorithm. We start by noting that Theorem 2.6 and Lemma 2.9 seamlessly extend to the dynamic setting. We respectively summarize the analogues of these statements in Theorem 2.12 and Lemma 2.13 below. Their proofs are deferred to Appendix A.

Theorem 2.12. *If our dynamic algorithm returns a vector $y \in \mathbb{R}_{\geq 0}^m$ after handling an update, then $\mathbb{1}^\top y = 1$ and $C^\top y \leq (1 + 4\epsilon) \cdot \mathbb{1}$. In contrast, if our dynamic algorithm returns a vector $\tilde{x}^t := \hat{x}^t / W \in \mathbb{R}_{\geq 0}^n$ after handling an update, then $\mathbb{1}^\top \tilde{x}^t \leq 1 + \epsilon$ and $C\tilde{x}^t \geq (1 - \epsilon) \cdot \mathbb{1}$.*

Lemma 2.13. *Throughout the duration of our dynamic algorithm, we have $\|\hat{x}\|_1^t \leq n^{(1/\epsilon)}$.*

---

[9]As we will see later in Section 2.5.2, we can assume that all updates are applied only to entries of the matrix C (and not on the objective nor the RHS of the constraints).

We now focus on bounding the total update time of our dynamic algorithm. The key challenge here is to derive an upper bound on the maximum number of times a given constraint $i \in [m]$ can get enforced, over the entire duration of the algorithm.[10] This is done in Lemma 2.14, whose proof appears in Section 2.2.1.

Lemma 2.14. *Throughout the entire duration of our dynamic algorithm, a given constraint $i \in [m]$ can get enforced at most $O\left(\frac{\log n}{\epsilon^2} \cdot \log\left(\frac{\lambda \log n}{\epsilon}\right)\right)$ times.*

We are now ready to bound the total update time of our dynamic algorithm.

Theorem 2.15. *To handle any sequence of τ restricting entry updates to C, our dynamic algorithm takes $O\left(\tau + N \cdot \frac{\log(n)}{\epsilon^2} \cdot \log^3\left(\frac{\lambda \log(n)}{\epsilon}\right)\right)$ time, where N is the number of non-zero entries in C at preprocessing.*

Proof. (Sketch) The total time update time is dominated by the time spent on the following two tasks.

Task I. After an update decreases the value of some entry C_{ij} of the input matrix C, decide whether the constraint $i \in [m]$ is approximately satisfied, i.e., whether $\left(C \cdot \frac{\hat{x}^t}{W}\right) > 1 - \epsilon/2$.

Task II. Enforce a given constraint $i \in [m]$.

We first focus on bounding the total time spent on Task II. Towards this end, we first adapt the argument in the proof of Lemma 2.8. This leads us to conclude that whenever we enforce a constraint $i \in [m]$, it takes $O\left(N_i \cdot \log^2\left(\frac{\lambda \log(n)}{\epsilon}\right)\right)$ time, where N_i is the number of non-zero entries in row $i \in [m]$ of the input matrix C at preprocessing. Hence, Lemma 2.14 implies that the total time spent on enforcing a given constraint $i \in [m]$ is at most $O\left(N_i \cdot \frac{\log(n)}{\epsilon^2} \cdot \log^3\left(\frac{\lambda \log(n)}{\epsilon}\right)\right)$. Summing this up over all the constraints $i \in [m]$, we infer that the total time spent on Task II is at most:

$$O\left(\sum_{i \in [m]} N_i \cdot \frac{\log(n)}{\epsilon^2} \cdot \log^3\left(\frac{\lambda \log(n)}{\epsilon}\right)\right) = O\left(N \cdot \frac{\log(n)}{\epsilon^2} \cdot \log^3\left(\frac{\lambda \log(n)}{\epsilon}\right)\right). \qquad (2.10)$$

It now remains to bound the total time spent on Task I. Towards this end, we maintain a variable $\hat{z}_j$ for each co-ordinate $j \in [n]$. We always ensure that $\hat{z}_j = (1 + \epsilon)^\kappa$ for some nonnegative integer $\kappa \geq 0$. Furthermore, we ensure that $\hat{z}_j$ always lies within a multiplicative $(1 + \epsilon)$ factor of $\left(\hat{x}^t\right)_j$. In other words, the value of $\hat{z}_j$ always forms an accurate estimate of $\left(\hat{x}^t\right)_j$. Finally, for each constraint $i \in [m]$, we explicitly maintain the value of $\left(C \cdot \frac{\hat{z}}{W}\right)_i$. This way, we can keep track of the value of $\left(C \cdot \frac{\hat{x}^t}{W}\right)_i$ for all $i \in [m]$, within a multiplicative factor of $(1 + \epsilon)$. This is sufficient for us to detect whether a given constraint $i \in [m]$ is approximately satisfied, in $O(1)$ time after an update.

Fix any co-ordinate $j \in [n]$. Note that $\left(\hat{x}^1\right)_j = 1$, and by Lemma 2.13 we have $\left(\hat{x}^t\right)_j \leq \sum_{j' \in [n]} \left(\hat{x}^t\right)_{j'} \leq n^{(1/\epsilon)}$ for all $t \in [T]$. In words, the value of $\left(\hat{x}^t\right)_j$ always lies in the interval $\left[1, n^{(1/\epsilon)}\right]$. We need to update the estimate $\hat{z}_j$ each time the value of $\left(\hat{x}^t\right)_j$ increases by a multiplicative factor of $(1 + \epsilon)$. Thus, throughout the duration of our algorithm, the value of $\hat{z}_j$ gets updated at most $\log_{(1+\epsilon)} n^{(1/\epsilon)} = O(\log n/\epsilon^2)$ times.

Finally, whenever the value of $\hat{z}_j$ changes, we need to spend an additional $O(N^j)$ time to reflect this change in the values of $\left(C \cdot \frac{\hat{z}}{W}\right)_i$ for all $i \in [m]$, where N^j denotes the number of non-zero entries in column $j \in [n]$ of the matrix C at preprocessing. Hence, the total time spent in this manner, in order to maintain the estimates $\hat{z}_j$ and $\left(C \cdot \frac{\hat{z}}{W}\right)_i$, is at most $O\left(\sum_{j \in [n]} N^j \cdot (\log n/\epsilon^2)\right) = O(N \log n/\epsilon^2)$. So the total time spent by our algorithm on Task I is at most $O(\tau + N \log n/\epsilon^2)$. The theorem now follows from (2.10). $\qquad \square$

---

[10]We say that constraint $i \in [m]$ gets enforced whenever we call ENFORE$(i, t, \hat{x}^t, W)$.

2.2.1 Proof of Lemma 2.14

Throughout the proof, fix a constraint $i \in [m]$. We associate a *step-size* with each enforcement of this constraint. Specifically, suppose that the constraint gets enforced in round t with step-size δ. Then the constraint gets whacked δ times during this enforcement (see Figure 4). Furthermore, note that the step-size δ lies in the range $[1, T]$, according to Figure 5. We discretize this range $[1, T]$ into $O(\log T)$ many intervals in powers of 2. Armed with this discretization, we now assign an integral *rank* to each enforcement of this constraint. Specifically, a given enforcement has rank $1 \leq \kappa \leq O(\log T)$ iff its step-size $\delta \in \left[2^{\kappa-1}, 2^{\kappa}\right)$. We will now bound the maximum number of enforcements of a given rank encountered by constraint i.

Claim 2.1. *Fix any integer $1 \leq \kappa \leq O(\log T)$. Throughout the duration of our dynamic algorithm, the constraint $i \in [m]$ encounters at most $O\left(\frac{\log n}{\epsilon^2}\right)$ many enforcements of rank κ.*

Proof. Suppose that the constraint $i \in [m]$ encounters $\gamma + 1$ enforcements of rank κ. Furthermore, suppose that these enforcements occurr in rounds $t_1 < t_2 < \cdots < t_\gamma < t_{\gamma+1}$. We will show that $\gamma = O(\log n/\epsilon^2)$.

Focus on the second-last of these enforcements, that occur in round t_γ. Just before this enforcement, we have $\left(C \cdot \frac{\hat{x}^t}{W}\right)_i < 1 - \epsilon/2$. In contrast, just after this enforcement, we have $\left(C \cdot \frac{\hat{x}^t}{W}\right)_i \geq 1$.[11] So the value of $\left(C \cdot \frac{\hat{x}^t}{W}\right)_i$ increases by at least a multiplicative factor of $1 + \Theta(\epsilon)$ during this enforcement. Hence, there must exist some co-ordinate $j^* \in [n]$ such that $\left(\hat{x}^t\right)_{j^*}$ also increases by at least a multiplicative factor of $1 + \Theta(\epsilon)$ during the same enforcement in round t_γ. On the other hand, since the concerned enforcement has rank κ, it has a step-size of at most 2^κ. Thus, due to this enforcement $\left(\hat{x}^t\right)_{j^*}$ increases by at most a factor of $\left(1 + \epsilon \cdot \frac{C_{ij^*}^{t_\gamma}}{\lambda}\right)^{2^\kappa}$, where $C_{ij^*}^{t_\gamma}$ denotes the value of C_{ij^*} just before round t_γ. Accordingly, we infer that:

$$\left(1 + \epsilon \cdot \frac{C_{ij^*}^{t_\gamma}}{\lambda}\right)^{2^\kappa} \geq 1 + \Theta(\epsilon). \tag{2.11}$$

Next, consider any previous enforcement with rank κ that occurs in some round $t_{\gamma'}$ (where $1 \leq \gamma' \leq \gamma$), and focus on the same co-ordinate j^*. Since the concerned enforcement has step-size at least $2^{\kappa-1}$, we infer that due to this enforcement $\left(\hat{x}^t\right)_{j^*}$ increases by at least a multiplicative factor of:

$$\left(1 + \epsilon \cdot \frac{C_{ij^*}^{t_{\gamma'}}}{\lambda}\right)^{2^{\kappa-1}} = \sqrt{\left(1 + \epsilon \cdot \frac{C_{ij^*}^{t_\gamma}}{\lambda}\right)^{2^\kappa}} \geq \sqrt{1 + \Theta(\epsilon)} = 1 + \Theta(\epsilon).$$

The inequality follows from (2.11). To summarize, we conclude that during each of the enforcements in rounds $t_1, \ldots, t_\gamma$, the value of $\left(\hat{x}^t\right)_{j^*}$ increases by at least a factor of $1 + \Theta(\epsilon)$.

Finally, observe that $\left(\hat{x}^t\right)_{j^*} = 1$ at the start of the algorithm, and that the value of $\left(\hat{x}^t\right)_{j^*}$ increases monotonically with time. Thus, at the end of the enforcement in round t_γ, we have:

$$(1 + \Theta(\epsilon))^\gamma \leq \left(\hat{x}^t\right)_{j^*} \leq \sum_{j \in [n]} \left(\hat{x}^t\right)_j \leq n^{(1/\epsilon)}. \tag{2.12}$$

The last inequality holds due to Lemma 2.13. From (2.12), we infer that $\gamma = O(\log n/\epsilon^2)$. $\square$

Claim 2.1 implies that a given constraint $i \in [m]$ is enforced at most $O\left(\frac{\log n}{\epsilon^2} \cdot \log T\right)$ times by our dynamic algorithm. Lemma 2.14 now follows from the fact that $T = \lambda \ln(n)/\epsilon^2$.

---

[11]Here, we rely on the fact that this is not the last enforcement of constraint i, for otherwise we might execute step (3) in Figure 5.

15

2.3 Streaming Whack-a-Mole MWU Algorithm

In this section, we focus on designing a streaming algorithm for Problem 2.1 in the following setting. We get a matrix $C \in R_{\geq 0}^{m \times n}$ as input. The rows of C are stored one after the other in a read-only repository. We wish to design an algorithm which makes one or more passes through this repository, and then either outputs a vector $x \in \mathbb{R}_{\geq 0}^n$ with $\mathbb{1}^\top x = 1$ and $Cx \geq (1 - \epsilon) \cdot \mathbb{1}$, or outputs a vector $y \in \mathbb{R}_{\geq 0}^m$ with $\mathbb{1}^\top y = 1$ and $C^\top y \leq (1 + 4\epsilon) \cdot \mathbb{1}$. The goal is to minimize the number of passes and the space complexity (excluding the space taken up by the repository to store the input) of the algorithm.

We observe that the static whack-a-mole MWU algorithm, as described in Section 2.1.3, immediately extends to this streaming setting if we allow the algorithm to have a space complexity of $O(m+n)$. Specifically, we can implement the main FOR loop in Figure 3 in a single pass as follows. The algorithm explicitly maintains a vector $\hat{x} = \hat{x}^t \in \mathbb{R}_{\geq 0}^n$, the values $W, t \in \mathbb{R}_{\geq 0}$, and a vector $y^* = (1/T) \cdot \sum_{t'=1}^t y^{t'} \in \mathbb{R}_{\geq 0}^m$ in its memory. Note that when $t = T$, we have $y^* = y$. While making a pass through the read-only repository, suppose that the algorithm encounters row $i \in [m]$ of the matrix C. The algorithm checks whether $\left(C \cdot \frac{\hat{x}}{W}\right)_i < 1 - \epsilon/2$, and if yes, then it enforces the constraint $i \in [m]$. Using the vectors $\hat{x}$ and y^*, this enforcement step can be performed without incurring any extra overhead in the space complexity.

Since we can implement the main FOR loop in Figure 3 in one pass, the total number of passes is equal to the number of phases of the whack-a-mole algorithm. The next theorem now follows from Corollary 2.10.

Theorem 2.16. *Consider a streaming setting where the rows of the matrix $C \in [0, \lambda]^{m \times n}$ arrive one after the other. Then there is a deterministic streaming algorithm with space complexity $O(m + n)$ that makes $O\left(\frac{\log n}{\epsilon^2}\right)$ many passes through this stream, and either returns a vector $x \in \mathbb{R}_{\geq 0}^n$ with $\mathbb{1}^\top x = 1$ and $Cx \geq (1 - \epsilon) \cdot \mathbb{1}$, or returns a vector $y \in \mathbb{R}_{\geq 0}^m$ with $\mathbb{1}^\top y = 1$ and $C^\top y \leq (1 + 4\epsilon) \cdot \mathbb{1}$.*

Next, note that if we are only required to return either the vector $x \in \mathbb{R}_{\geq 0}^n$ or a special symbol NULL (indicating that the dual packing LP has an approximately feasible solution with objective $= 1$), then we can further reduce the space complexity of our streaming algorithm. This holds because in such a scenario we only need to maintain the vector $\hat{x} \in \mathbb{R}_{\geq 0}^n$ and the values $W, t \in \mathbb{R}_{\geq 0}$. In particular, we no longer need to maintain the vector $y^* \in \mathbb{R}_{\geq 0}^m$ while making a pass through the read-only repository. Instead, when we observe that $t = T$, we simply return NULL. This leads to the following corollary.

Corollary 2.17. *Consider a streaming setting where the rows of the matrix $C \in [0, \lambda]^{m \times n}$ arrive one after the other. Then there is a deterministic streaming algorithm with space complexity $O(n)$ that makes $O\left(\frac{\log n}{\epsilon^2}\right)$ many passes through this stream, and either returns a vector $x \in \mathbb{R}_{\geq 0}^n$ with $\mathbb{1}^\top x = 1$ and $Cx \geq (1 - \epsilon) \cdot \mathbb{1}$, or returns NULL. In the latter case, it is guaranteed that there exists a vector $y \in \mathbb{R}_{\geq 0}^m$ with $\mathbb{1}^\top y = 1$ and $C^\top y \leq (1 + 4\epsilon) \cdot \mathbb{1}$ (although the algorithm does not return y).*

2.4 Online Whack-a-Mole MWU Algorithm

In this section, we focus on designing an online algorithm for Problem 2.1 in the following setting. An adversary chooses a matrix $C \in [0, \lambda]^{m \times n}$ as input. In the beginning, the values of n and λ are public knowledge, whereas only the adversary knows the value of m and the entries of the matrix C. Subsequently, the adversary reveals to us the rows of this matrix one after another. We need to maintain a vector $x \in \mathbb{R}_{\geq 0}^n$ such that $\mathbb{1}^\top x \leq 1 + \Theta(\epsilon)$ and $Cx \geq (1 - \Theta(\epsilon)) \cdot \mathbb{1}$ until a certain point in time, and after that we need to terminate our algorithm by returning a $y \in \mathbb{R}_{\geq 0}^m$ such that $\mathbb{1}^\top y \geq 1 - \Theta(\epsilon)$ and $C^\top y \leq (1 + \Theta(\epsilon)) \cdot \mathbb{1}$. We incur a recourse of one each time we decrease the value of some variable $(x)_j, j \in [n]$. Our goal is to design an algorithm in this online setting with as small total recourse as possible.

We observe that the dynamic whack-a-mole MWU algorithm, as described in Section 2.2, seamlessly extends to this online setting. Thus, we maintain the vector $\tilde{x}^t := \hat{x}^t / W$. Whenever a new row $i \in [m]$ of

16

the matrix C arrives, we check whether $\left(C \cdot \frac{\hat{x}^t}{W}\right) < 1 - \epsilon/2$, and if the answer is yes, then we enforce the corresponding constraint $i \in [m]$. The correctness of this algorithm follows from Theorem 2.12.

We now derive an upper bound on the number of phases. Corollary 2.13 implies that $\left\|\hat{x}^t\right\|_1 \leq n^{(1/\epsilon)}$ throughout the duration of the algorithm. Since $\left\|\hat{x}^1\right\|_1 = n$ at the start of the algorithm, and since we initiate a new phase whenever $\left\|\hat{x}^t\right\|_1$ increases by a multiplicative factor of $1 + \Theta(\epsilon)$, it follows that the total number of phases is at most $\log_{(1+\Theta(\epsilon))} n^{(1/\epsilon)} = O\left(\frac{\log n}{\epsilon^2}\right)$.

Next, note that within a given phase our algorithm incurs *zero* recourse. This holds because the values $\left(\hat{x}^t/W\right)_j$ can only increase when we enforce a constraint within the phase (as W does not change). On the other hand, we incur a recourse of n each time we initiate a new phase, since we have n variables and each of these variables decreases its value as we increase W at the start of the phase. Hence, the total recourse of the algorithm is n times the number of phases. This leads to the following theorem.

Theorem 2.18. *Consider an online setting where the rows of the matrix $C \in [0, \lambda]^{m \times n}$ arrive one after another. Let N denote the number of non-zero entries in C. There is a deterministic online algorithm with total recourse $O\left(\frac{n \log n}{\epsilon^2}\right)$ which has the following property. It maintains a vector $x \in \mathbb{R}^n_{\geq 0}$ with $\mathbb{1}^\top x \leq 1 + \Theta(\epsilon)$ and $Cx \geq (1 - \Theta(\epsilon)) \cdot \mathbb{1}$ until a certain point in time, and after that it terminates and returns a vector $y \in \mathbb{R}^m_{\geq 0}$ with $\mathbb{1}^\top y \geq 1 - \Theta(\epsilon)$ and $C^\top y \leq (1 + \Theta(\epsilon)) \cdot \mathbb{1}$.*

2.5 Reductions to General Packing-Covering LPs

Consider a matrix $C \in \mathbb{R}^{m \times n}_{\geq 0}$, vectors $a \in \mathbb{R}^n_{>0}$, $b \in \mathbb{R}^m_{>0}$, and a generic covering LP defined below.

$$\text{Minimise } a^\top x \qquad \text{s.t. } Cx \geq b \text{ and } x \in \mathbb{R}^n_{\geq 0}. \tag{2.13}$$

The dual of the above LP is given by:

$$\text{Maximise } b^\top y \qquad \text{s.t. } C^\top y \leq a \text{ and } y \in \mathbb{R}^m_{\geq 0}. \tag{2.14}$$

Let OPT be the optimal objective value of this pair of primal and dual LPs. Say that $x \in \mathbb{R}^n_{\geq 0}$ is an ϵ-*approximate optimal solution* to LP (2.13) iff $a^\top x \leq (1 + \Theta(\epsilon)) \cdot$ OPT and $Cx \geq (1 - \Theta(\epsilon)) \cdot b$. Similarly, say that $y \in \mathbb{R}^m_{\geq 0}$ is an ϵ-*approximate optimal solution* to LP (2.14) iff $b^\top y \geq (1 - \Theta(\epsilon)) \cdot$ OPT and $C^\top y \leq (1 + \Theta(\epsilon)) \cdot a$.[12] We will now explain how to use the whack-a-mole MWU algorithm to obtain ϵ-approximate optimal solutions to this pair of LPs in static, dynamic, streaming and online settings.

Specifically, we describe how an algorithm for Problem 2.1 can be used in a black-box manner to solve general packing-covering LPs. Such a reduction works by first guessing the value of OPT, and then showing an equivalence between: (a) finding a solution with objective = OPT and (b) solving Problem 2.1. The equivalence follows from standard scaling techniques.

2.5.1 Static Setting

In this section, we show how to prove the following theorem.

Theorem 2.19. *There is a deterministic ϵ-approximation algorithm for solving LP (2.13) that runs in $O\left(N \cdot \frac{\log(n)}{\epsilon^3} \cdot \log^2\left(\frac{nU \log(n)}{\epsilon L}\right) \cdot \log(nU/L)\right)$ time. Here, N denotes the number of non-zero entries in C, whereas L (resp. U) respectively denotes a lower (resp. upper) bound on the minimum (resp. maximum) value of any non-zero entry in C, a, b. The same guarantee also holds for solving LP (2.14).*

---

[12]This notion of an ϵ-approximation is equivalent to the one defined at the start of Section 1, upto a multiplicative factor of $1 + \Theta(\epsilon)$. To see why this is true, consider an x such that $1^\top x \leq (1 + \Theta(\epsilon)) \cdot$ OPT and $Cx \geq (1 - \Theta(\epsilon)) \cdot b$. Then it follows that $1^\top \tilde{x} \leq (1 + \Theta(\epsilon)) \cdot$ OPT and $C\tilde{x} \geq b$, where $\tilde{x} := \frac{x}{1 - \Theta(\epsilon)}$. A similar argument holds for an ϵ-approximation to LP (2.14).

Define a matrix $C' \in \mathbb{R}^{m \times n}_{\geq 0}$ such that $C'_{ij} := C_{ij}/(a_j b_i)$ for all $i \in [m], j \in [n]$. It is easy to check that LP (2.13) and LP (2.14) can be equivalently written as:

$$\text{Minimise } \mathbb{1}^\top x \qquad \text{s.t. } C'x \geq \mathbb{1} \text{ and } x \in \mathbb{R}^n_{\geq 0}. \tag{2.15}$$

$$\text{Maximise } \mathbb{1}^\top y \qquad \text{s.t. } (C')^\top y \leq \mathbb{1} \text{ and } y \in \mathbb{R}^m_{\geq 0}. \tag{2.16}$$

We can compute the matrix C' in $O(N)$ time. Furthermore, the maximum (resp. minimum) value of any non-zero entry of C' is upper (resp. lower) bounded by U/L^2 (resp. L/U^2). Let OPT denote the optimal objective value of LP (2.15) and LP (2.16). Note that $L^2/U \leq \text{OPT} \leq nU^2/L$. We discretize this range $[L^2/U, nU^2/L]$ in powers of $(1 + \epsilon)$, to obtain $\log_{(1+\epsilon)}(nU^3/L^3) = \Theta(\log(nU/L)/\epsilon)$ many *guesses* for OPT. For each such guess μ, we consider the following problem.

Problem 2.20. *Either return an $x \in \mathbb{R}^n_{\geq 0}$ such that $\mathbb{1}^\top x \leq (1 + \Theta(\epsilon)) \cdot \mu$ and $C'x \geq (1 - \Theta(\epsilon)) \cdot \mathbb{1}$, or return a $y \in \mathbb{R}^m_{\geq 0}$ such that $\mathbb{1}^\top y \geq (1 - \Theta(\epsilon)) \cdot \mu$ and $(C')^\top y \leq (1 + \Theta(\epsilon)) \cdot \mathbb{1}$.*

Next, define a matrix $C'' \in \mathbb{R}^{m \times n}_{\geq 0}$ where $C''_{ij} := \mu \cdot C_{ij}$ for all $i \in [m], j \in [n]$, and observe that Problem 2.20 is equivalent to the following problem.

Problem 2.21. *Either return an $x \in \mathbb{R}^n_{\geq 0}$ such that $\mathbb{1}^\top x \leq 1 + \Theta(\epsilon)$ and $C''x \geq (1 - \Theta(\epsilon)) \cdot \mathbb{1}$, or return a $y \in \mathbb{R}^m_{\geq 0}$ such that $\mathbb{1}^\top y \geq 1 - \Theta(\epsilon)$ and $(C'')^\top y \leq (1 + \Theta(\epsilon)) \cdot \mathbb{1}$.*

Note that $C'' \in [0, \lambda]^{m \times n}$ where $\lambda = \mu \cdot (U/L^2) \leq nU^3/L^3$. Accordingly, we can solve Problem 2.21 by using our whack-a-mole MWU algorithm from Section 2.1. According to Theorem 2.11, this takes $O\left(N \cdot \frac{\log(n)}{\epsilon^2} \cdot \log^2\left(\frac{nU^3 \log(n)}{\epsilon L^3}\right)\right) = O\left(N \cdot \frac{\log(n)}{\epsilon^2} \cdot \log^2\left(\frac{nU \log(n)}{\epsilon L}\right)\right)$ time.

It is easy to check that we can recover ϵ-approximate optimal solutions to LP (2.13) and LP (2.14) if we solve Problem 2.21 for each of the $\Theta(\log(nU/L)/\epsilon)$ many guesses μ. This leads us to Theorem 2.19.

2.5.2 Dynamic Setting

In this section, we show how to prove the following theorem.

Theorem 2.22. *Consider any sequence of τ restricting updates to LP (2.13), where each update either decreases an entry of C, or increases an entry of a, b. Let N denote the total number of non-zero entries in C at preprocessing. Let L (resp. U) denote a lower (resp. upper) bound on the minimum (resp. maximum) value of a non-zero entry of C, a, b. We can deterministically maintain an ϵ-approximate optimal solution to LP (2.13) in $O\left(\tau \cdot \frac{\log(nU/L)}{\epsilon} + N \cdot \frac{\log(n)}{\epsilon^3} \cdot \log^3\left(\frac{nU \log(n)}{\epsilon L}\right) \cdot \log(nU/L)\right)$ total time. The same guarantee also holds for maintaining an ϵ-approximate solution to LP (2.14), which undergoes relaxing updates.*

As in Section 2.5.1, we make $\Theta(\log(nU/L)/\epsilon)$ many *guesses* for OPT. For each such guess μ, we maintain a solution to Problem 2.20, which, in turn, is equivalent to Problem 2.21. This allows us to maintain ϵ-approximate optimal solutions to LP (2.13) and LP (2.14) in the dynamic setting.

Fix any guess μ, and focus on the corresponding instance of Problem 2.21. Since we are satisfied with an ϵ-approximate optimal solution, we can safely assume that any update with a *meaningful* impact on the solution changes the corresponding entry by at least a multiplicative factor of $(1 + \epsilon)$. Henceforth, we only focus on handling these meaningful updates. Note that each entry of C, a, b participates in at most $\log_{(1+\epsilon)}(U/L) = O\left(\frac{\log(U/L)}{\epsilon}\right)$ many meaningful updates. Furthermore, each time there is a meaningful restricting update to some b_i, it leads to at most $O(N_i)$ many restricting entry updates to the matrix C'', where N_i is the number of non-zero entries in the ith row of C at preprocessing. Similarly, whenever there is a meaningful restricting update to some a_j, it leads to at most $O(N^j)$ many restricting entry updates to the matrix C'', where N^j is the number of non-zero entries in the jth column of C at preprocessing.

Consider any sequence of τ restricting updates to LP (2.13). By the above discussion, this leads to at most $\tau + \left(\sum_{j \in [n]} N^j + \sum_{i \in [m]} N_i \right) \cdot O\left(\frac{\log(U/L)}{\epsilon} \right) = \tau + O\left(N \cdot \frac{\log(U/L)}{\epsilon} \right)$ many restricting entry updates in Problem 2.21, which we solve using the algorithm from Section 2.2. As $\lambda \leq nU^3/L^3$, Theorem 2.15 implies a total update time of:

$$O\left(\tau + N \cdot \frac{\log(U/L)}{\epsilon} + N \cdot \frac{\log(n)}{\epsilon^2} \cdot \log^3\left(\frac{\lambda \log(n)}{\epsilon} \right) \right) = O\left(\tau + N \cdot \frac{\log(n)}{\epsilon^2} \cdot \log^3\left(\frac{nU \log(n)}{\epsilon L} \right) \right).$$

Theorem 2.22 follows since we run $\Theta\left(\frac{\log(nU/L)}{\epsilon} \right)$ copies of the dynamic algorithm, one for each μ.

2.5.3 Streaming Setting

In this section, we consider the streaming setting where the rows of the constraint matrix of a covering LP (or equivalently, the columns of the constraint matrix of the dual packing LP) are arriving one after another. We show how to prove the following two theorems.

Theorem 2.23. *Consider a streaming setting where the columns of the constraint matrix of LP (2.14) arrive one after another. There is a deterministic algorithm with space complexity $O(m + n)$ that makes $O\left(\frac{\log n \cdot \log(nU/L)}{\epsilon^3} \right)$ passes through the stream, and returns an ϵ-approximate optimal solution to LP (2.14). Here, L (resp. U) denotes a lower (resp. upper) bound on the value of any non-zero entry of C, a, b.*

Theorem 2.24. *Consider a streaming setting where the rows of the constraint matrix of LP (2.13) arrive one after another. There is a deterministic algorithm with space complexity $O(n)$ that makes $O\left(\frac{\log n \cdot \log(nU/L)}{\epsilon^3} \right)$ passes through the stream, and returns an ϵ-approximate optimal solution to LP (2.13). Here, L (resp. U) denotes a lower (resp. upper) bound on the value of any non-zero entry of C, a, b.*

As in Section 2.5.1, we make $\Theta(\log(nU/L)/\epsilon)$ many *guesses* for OPT. For each such guess μ, we solve Problem 2.20 in the streaming setting, which, in turn, is equivalent to Problem 2.21. This allows us to return ϵ-approximate optimal solutions to LP (2.13) and LP (2.14).

Fix any guess μ, and focus on the corresponding instance of Problem 2.21. Suppose that we are allowed to have a space complexity of $O(m + n)$. By Theorem 2.16, we can solve Problem 2.21 by making $O(\frac{\log n}{\epsilon^2})$ many passes through the stream. Theorem 2.23 now follows since we need to solve Problem 2.21 for each guess μ, which increases the number of passes by a multiplicative factor of $\Theta(\log(nU/L)/\epsilon)$.

Finally, Theorem 2.24 follows if we apply the same strategy described above and use Corollary 2.17.

2.5.4 Online Setting

In this section, we explain how to prove the following theorem.

Theorem 2.25. *Consider an online setting where the constraints of LP (2.13) arrive one after another. There is a deterministic algorithm which maintains an ϵ-approximate optimal solution to this LP with total recourse $O\left(n \cdot \frac{\log n}{\epsilon^3} \cdot \log\left(\frac{nU}{L} \right) \right)$. Here, L (resp. U) is a lower (resp. upper) bound on the value of any non-zero entry of C, a, b.*

As in Section 2.5.1, we make $\Theta(\log(nU/L)/\epsilon)$ many *guesses* for OPT. For each such guess μ, we solve Problem 2.20 in the online setting, which, in turn, is equivalent to Problem 2.21. This allows us to maintain an ϵ-approximate optimal solutions to LP (2.13). Theorem 2.25 now follows from Theorem 2.18 as the total recourse increases by a multiplicative factor of $\Theta(\log(nU/L)/\epsilon)$.

2.6 Challenges for Dynamic Whack-a-Mole MWU Algorithm for Packing LPs

We have so far focussed on the whack-a-mole MWU algorithm for covering LPs. For packing LPs, there exists an analogous whack-a-mole MWU algorithm. To see this, consider the following problem.

Problem 2.26. *Given a matrix $P \in [0, \lambda]^{m \times n}$ where $\lambda > 0$, either return an $x \in \mathbb{R}^n_{\geq 0}$ with $\mathbb{1}^\top x \geq 1 - \Theta(\epsilon)$ and $Px \leq (1 + \Theta(\epsilon)) \cdot \mathbb{1}$, or return a $y \in \mathbb{R}^m_{\geq 0}$ with $\mathbb{1}^\top y \leq 1 + \Theta(\epsilon)$ and $P^\top y \geq (1 - \Theta(\epsilon)) \cdot \mathbb{1}$.*

The basic template for a whack-a-mole algorithm for Problem 2.26 is described in Figure 6 and Figure 7. Note that this is completely analogous to the template in Section 2.1.1.

---

01.   Define $T \leftarrow \frac{\lambda \ln(n)}{\epsilon^2}$, and two vectors $\hat{x}^1, x^1 \in \mathbb{R}^n_{\geq 0}$ where $\hat{x}^1 \leftarrow \mathbb{1}$ and $x^1 \leftarrow \frac{\hat{x}^1}{\|\hat{x}^1\|_1}$.
02.   FOR $t = 1$ TO T:
03.   EITHER
04.   Conclude that $\left(P \cdot x^t\right)_i \leq 1 + \epsilon$ for all $i \in [m]$.
05.   Terminate the FOR loop, and RETURN (x^t, NULL).
06.   OR
07.   Find a packing constraint $i_t \in [m]$ such that $\left(P \cdot x^t\right)_{i_t} > 1$.
08. $\hat{x}^{t+1} \leftarrow \text{WHACK}(i_t, \hat{x}^t)$.   // See Figure 7.
09. $x^{t+1} \leftarrow \frac{\hat{x}^{t+1}}{\|\hat{x}^{t+1}\|_1}$.
10.   Let $y^t \in \Delta^m$ be the vector where $\left(y^t\right)_{i_t} = 1$ and $\left(y^t\right)_i = 0$ for all $i \in [m] \setminus \{i_t\}$.
11. $y \leftarrow (1/T) \cdot \sum_{t=1}^T y^t$.
12.   RETURN (NULL, y).

Figure 6: The Whack-a-Mole MWU Algorithm for a Packing LP.

---

1. FOR ALL $j \in [n]$:
2. $\hat{z}_j \leftarrow \left(1 - \epsilon \cdot \frac{P_{ij}}{\lambda}\right) \cdot \hat{x}_j$.
3. RETURN $\hat{z}$

Figure 7: WHACK$(i, \hat{x})$.

The key point to note is that unlike in Section 2.1.1, here the weight $\left(\hat{x}^t\right)_j$ of an expert $j \in [n]$ (see the discussion in the beginning of Section 2.1.2) can only *decrease* whenever we whack a constraint. Nevertheless, we can easily extend the analysis from Section 2.1.1 to obtain the following theorem.

Theorem 2.27. *The algorithm in Figure 6 either returns a vector $x^t \in \mathbb{R}^n_{\geq 0}$ with $\mathbb{1}^\top x^t = 1$ and $Px^t \leq (1 + \Theta(\epsilon)) \cdot \mathbb{1}$, or it returns a vector $y \in \mathbb{R}^m_{\geq 0}$ with $\mathbb{1}^\top y = 1$ and $P^\top y \geq (1 - \Theta(\epsilon)) \cdot \mathbb{1}$.*

From this basic template, it is straightforward to obtain a near-linear time static algorithm for Problem 2.26. The framework also seamlessly extends to give us whack-a-mole MWU algorithms for packing LPs in the streaming and online settings. Thus, every theorem derived in Sections 2.5.1, 2.5.3 and 2.5.4 holds even if we switch the occurrences of LP (2.13) with that of LP (2.14) in the concerned theorem statement. The only exception is the theorem derived in Section 2.5.2. Below, we highlight the main challenge in obtaining a dynamic whack-a-mole MWU algorithm for packing LPs under restricting updates.

Recall that a key part of the analysis in Section 2.2 was deriving an upper bound on the number of times a given constraint can get enforced throughout the duration of the algorithm (see Lemma 2.14). The proof

of Lemma 2.14, on the other hand, relied on the property that the weight $(\hat{x}^t)_j$ of every expert $j \in [n]$ is monotonically non-decreasing with time and always lies in the range $\left[1, n^{(1/\epsilon)}\right]$, and hence is polynomially bounded (see the discussion leading to (2.12)). This property followed from three observations: (1) at the start of the algorithm we have $(\hat{x}^1)_j = 1$, (2) the total weight $\sum_{j' \in [n]} (\hat{x}^t)_j$ of all the experts is at most $n^{(1/\epsilon)}$, and (3) the weight $(\hat{x}^t)_j$ of a given expert $j \in [n]$ is at most the total weight $\sum_{j' \in [n]} (\hat{x}^t)_j$.

Coming back to our current setting, recall that the weight $(\hat{x}^t)_j$ of an expert $j \in [n]$ can only decrease over time when we run the whack-a-mole MWU algorithm on a packing LP. Thus, in order to bound the maximum number of times a constraint can get enforced, we need to show that the weight $(\hat{x}^t)_j$ of every expert $j \in [n]$ lies in the range $\left[\frac{1}{n^{\text{poly}(1/\epsilon)}}, 1\right]$ throughout the duration of the algorithm. A natural way to replicate the argument from the previous paragraph to our current setting would be to make the following sequence of claims: (a) at the start of the algorithm we have $(\hat{x}^1)_j = 1$, (b) the total weight $\sum_{j' \in [n]} (\hat{x}^t)_j$ of all the experts is at least $n^{(1/\epsilon)}$, and (c) the weight $(\hat{x}^t)_j$ of a given expert $j \in [n]$ is *at least* the total weight $\sum_{j' \in [n]} (\hat{x}^t)_j$. Now, the crucial observation is that although we can extend the analysis of the whack-a-mole MWU algorithm for covering LP to prove claims (a) and (b), claim (c) clearly does *not* hold for obvious reasons. Indeed, if we consider a natural extension of the static whack-a-mole MWU algorithm for packing LPs to a setting with restricting updates, then we can *not* ensure that the weight $(\hat{x}^t)_j$ of an expert $j \in [n]$ remains polynomially bounded. This, in turn, implies that we cannot upper bound the maximum number of times a constraint gets enforced, and hence we cannot bound the total update time of the algorithm.

We address this issue in Section 3, where we present an efficient dynamic algorithm for maintaining an ϵ-approximate solution to a *positive LP* under relaxing updates. Section 3.3 shows that as a simple corollary of this result, we obtain an efficient dynamic algorithm for maintaining an ϵ-approximate optimal solution to a packing (resp. covering) LP under restricting (resp. relaxing) updates (see Theorem 3.15).

3 The Greedy MWU Algorithm

In this section we focus on *positive* LPs, which are defined as follows:

Find $x \in \mathbb{R}_{\geq 0}^n$ such that $Px \leq a$ and $Cx \geq b$, where $P \in \mathbb{R}_{\geq 0}^{m_p \times n}, a \in \mathbb{R}_{\geq 0}^{m_p}, C \in \mathbb{R}_{\geq 0}^{m_c \times n}, b \in \mathbb{R}_{\geq 0}^{m_c}$. (3.1)

An ϵ-approximate solution for the positive LP is either an $x \in \mathbb{R}_{\geq 0}^n$ satisfying $Px \leq (1 + \epsilon)a$ and $Cx \geq (1 - \epsilon)b$, or a symbol $\perp$ indicating that the LP is infeasible. In the dynamic setting, an update to the LP can change an entry of P, a, C or b. Observe that the update is relaxing if it increases an entry of C or a, or decreases an entry of P or b. In this section, we wish to design a dynamic algorithm for maintaining an ϵ-approximation solution to a positive LP undergoing relaxing updates. Note that such an algorithm will work as follows. Initially, it will return $\perp$ for a sequence of relaxing updates. After that, at a certain point in time it will return an $x \in \mathbb{R}_{\geq 0}^n$ satisfying $Px \leq (1 + \epsilon)a$ and $Cx \geq (1 - \epsilon)b$. From this point onward, the same x will continue to remain an ϵ-approximate solution to the input positive LP after every future update.

Notation and preliminaries. We now introduce a few key notations and concepts that will be used throughout the rest of Section 3.[13] We classify a relaxing update as an *entry update* if it changes an entry of P or C, whereas we classify a relaxing update as a *translation update* if it changes an entry of a or b. Throughout the sequence of updates, let L (resp. U) respectively denote an upper (resp, lower) bound on the value of any nonzero entry in P, C, a, b; and let N denote an upper bound on the total number of nonzero entries in P, C. We assume that $U/L = O(\text{poly}(m_p + m_c + n))$, where m_p, m_c and n respectively denote the number of rows in P, the number of rows in C, and the number of variables. We refer to $Px \leq a$ as the

---

[13]We emphasize that some of these notations are different from the ones used in Section 2.

packing constraints, and we refer to $Cx \geq b$ as the *covering* constraints. For all $i \in [m_p]$ and $k \in [n]$, we let $P(i, k) \in \mathbb{R}_{\geq 0}$ denote the entry in the ith row and k^{th} column of the matrix P. Furthermore, we let $P_i \in \mathbb{R}_{\geq 0}^{1 \times n}$ denote the ith row of the matrix P. We analogously define the notations $C(j, k) \in \mathbb{R}_{\geq 0}$ and $C_j \in \mathbb{R}_{\geq 0}^{1 \times n}$, for all $j \in [m_c]$ and $k \in [n]$. Finally, we use the symbol v_i to denote the ith co-ordinate of a vector $v = (v_1, \ldots, v_m) \in \mathbb{R}^m$.

Our main result is summarised below. In this section, we present a high level overview of the main ideas behind the proof of Theorem 3.1. The full details are deferred to Appendix B and Appendix C.

Theorem 3.1. *We can maintain an ϵ-approximate solution to a positive LP undergoing a sequence of $t > 0$ relaxing updates in $O\left(N \cdot \frac{\log^2(m_c + m_p + U/L)}{\epsilon^2} + t\right)$ total update time.*

A simplifying assumption: To convey the main ideas behind our algorithm, in this section we focus on the setting where all the relaxing updates made to the input LP are entry updates, for the following reason.

Consider an input positive LP (P, C, a, b), which asks us to find an x such that $Px \leq a$ and $Cx \geq b$. Since we are happy with a $\Theta(\epsilon)$-approximate solution to the input LP, we can keep ignoring the relaxing translation updates to a packing constraint $(Px)_i \leq a_i$ (resp. covering constraint $(Cx)_j \geq b_j$) as long as a_i (resp. b_j) does *not* increase (resp. decrease) by more than a multiplicative factor of $(1 + \epsilon)$. Thus, during the course of our algorithm, a given packing or covering constraint will essentially go through at most $\kappa = O(\log_{(1+\epsilon)}(U/L))$ many relaxing translation updates.

Next, note that we can simulate a relaxing translation update to a packing constraint $(Px)_i \leq a_i$ (resp. covering constraint $(Cx)_j \geq b_j$) by making $nzp_{(i)}$ (resp. $nzc_{(j)}$) many relaxing entry updates, where $nzp_{(i)}$ (resp. $nzc_{(j)}$) denotes the current number of nonzero entries in row i (resp. row j) of P (resp. C). This is because of the following reason: If the translation update asks us to scale up (resp. scale down) the value of a_i (resp. b_j) by a multiplicative factor of $\alpha \geq 1$, then we can implement this update by scaling down (resp. scaling up) every nonzero entry in row i (resp. row j) of P (resp. C) by the same factor α.

It follows that all the relaxing translation updates encountered by an algorithm for this problem can be simulated by at most κN many relaxing entry updates. Hence, if we prove Theorem 3.1 for relaxing entry updates, then we can immediately convert it into an algorithm that handles *both* translation and entry updates with total update time $O\left(N \cdot \frac{\log^2(m_c + m_p + U/L)}{\epsilon^2} + t + \kappa N\right) = O\left(N \cdot \frac{\log^2(m_c + m_p + U/L)}{\epsilon^2} + t\right)$.

Accordingly, for the rest of Section 3, the phrase *relaxing update* will refer to a *relaxing entry update*.

Scaling the constraints: Since we have to deal with only the entry updates in this section, for notational convenience we will scale the right hand sides of all the packing and covering constraints to 1. Accordingly, throughout the rest of Section 3 the input positive LP will be given by an ordered pair (P, C), where $P \in \mathbb{R}_{\geq 0}^{m_p \times n}$ and $C \in \mathbb{R}_{\geq 0}^{m_c \times n}$ and it asks us to find an $x \in \mathbb{R}_{\geq 0}^n$ such that $Px \leq 1$ and $Cx \geq 1$.

Organisation: In Section 3.1, we recap a known *static* greedy MWU algorithm for positive LPs [Qua20, You14]. We present and analyse our dynamic algorithm in Section 3.2. Finally, we show that the analysis in Section 3.2 implies a dynamic algorithm for solving a packing (resp. covering) LP under restricting (resp. relaxing) updates. This observation is summarized in Section 3.3.

3.1 Static Greedy MWU Algorithm [Qua20, You14]

Define two functions $f_p(x)$ and $f_c(x)$, which respectively correspond to the *soft-max* of the packing constraints and the *soft-min* of the covering constraints, as stated below.

$$f_p(x) = \frac{1}{\eta} \cdot \log \left(\sum_{i=1}^{m_p} \exp(\eta P_i x) \right) \text{ and } f_c(x) = -\frac{1}{\eta} \cdot \log \left(\sum_{j=1}^{m_c} \exp(-\eta C_j x) \right),$$

$$\text{where } \eta = \frac{\log(m_p + m_c + U/L)}{\epsilon}. \tag{3.2}$$

It can be shown that the function $f_p(x)$ (resp. $f_c(x)$) closely approximates the maximum (resp. minimum) value among the left hand sides of the packing (resp. covering) constraints. Specifically, we have:

$$\max_{i \in [m_p]} P_i x \leq f_p(x) \leq \max_{i \in [m_p]} P_i x + \epsilon \text{ and } \min_{j \in [m_c]} C_j x - \epsilon \leq f_c(x) \leq \min_{j \in [m_c]} C_j x, \text{ for all } x \in \mathbb{R}^n_{\geq 0}. \tag{3.3}$$

The algorithm attempts to find an $x \in \mathbb{R}^n_{\geq 0}$ where $f_p(x) \leq 1$ and $f_c(x) \geq 1$. By (3.3), such an x is indeed an ϵ-approximate solution to the input positive LP. Note that both the functions $f_p(x)$ and $f_c(x)$ are continuous and differentiable, and hence their gradients exist at all points.

3.1.1 The main idea

Say that a direction $z \in \mathbb{R}^n_{\geq 0}$ is *cheap* with respect to a point $x \in \mathbb{R}^n_{\geq 0}$ iff $\langle \nabla f_p(x), z \rangle \leq \langle \nabla f_c(x), z \rangle$. In other words, starting from the point x, if we take an infinitesimally small step towards a cheap direction w.r.t. x, then the increase in $f_p(x)$ is at most the increase in $f_c(x)$. The high level idea behind the algorithm can be summarised as follows: It starts at an $x = \vec{0} \in \mathbb{R}^n_{\geq 0}$, where we have $f_p(x) = f_c(x) = 0$. It then continuously keeps *moving* this point x in a cheap direction. This leads to the following invariant: $f_p(x) \leq f_c(x)$. The algorithm stops once it reaches a point x where $f_c(x) = 1$. Such an x satisfies $f_p(x) \leq 1$ and $f_c(x) \geq 1$, and hence is an ϵ-approximate solution to the input positive LP, according to (3.3). On the other hand, if at some point in time during this continuous process, we reach a point x that does *not* admit any cheap direction, then it can be shown that the input LP is infeasible.

In order to actually implement the scheme described above, we first need to *discretise* the process which keeps moving the point x. This discretised process will consist of a sequence of steps. As before, initially we have $x = \vec{0}$. At the start of each step, we identify an (approximately) cheap direction $z \in \mathbb{R}^n_{\geq 0}$ w.r.t. the current x, which satisfies $\langle \nabla f_p(x), z \rangle \leq (1 + \Theta(\epsilon)) \cdot \langle \nabla f_c(x), z \rangle$, and then we take a small discrete *jump* along that direction by setting $x \leftarrow x + \delta z$ for some sufficiently small $\delta > 0$. It can be shown that due to each of these steps, the increase in the value of $f_p(x)$ is at most $(1 + \Theta(\epsilon))$ times the increase in the value of $f_c(x)$, which leads to the invariant: $f_p(x) \leq (1 + \Theta(\epsilon)) \cdot f_c(x)$. Furthermore, during each of these steps, the value of $f_c(x)$ increases by at most $\Theta(\epsilon)$, for sufficiently small δ. The process terminates *immediately* after we reach a stage where $f_c(x) \geq 1$. Thus, at termination we have $1 \leq f_c(x) \leq (1 + \Theta(\epsilon))$ and $f_p(x) \leq (1 + \Theta(\epsilon)) f_c(x) \leq (1 + \Theta(\epsilon))$, and hence x is a $\Theta(\epsilon)$-approximate solution to the input positive LP. In contrast, if at any step during this process, we end up with an x which does not admit any (approximately) cheap direction, then we can certify that the input LP is infeasible.

We will work with a concrete version of the framework described above. Say that a *coordinate* $k \in [n]$ is (approximately) *cheap* with respect to a point $x \in \mathbb{R}^n_{\geq 0}$ iff $\langle \nabla f_p(x), \vec{e}_k \rangle \leq (1 + \Theta(\epsilon)) \cdot \langle \nabla f_c(x), \vec{e}_k \rangle$, where $e_k \in \{0, 1\}^n$ is the unit vector along coordinate k. We will use the following crucial observation: Any given point x admits a cheap direction iff it admits a cheap coordinate. Thus, all we need to do is start at $x = \vec{0}$, and then move the point x in a sequence of steps, where at each step we move x by a small amount along a cheap coordinate. We now present this algorithmic template more formally in Section 3.1.2.

23

3.1.2 The basic algorithmic template

We start by introducing some crucial notations. Given any $x \in \mathbb{R}^n_{\geq 0}$, we associate a *weight* with each LP constraint. To be specific, for all $i \in [m_p]$ and $j \in [m_c]$, we have $w_p(x, i) := \exp(\eta \cdot P_i x)$ and $w_c(x, j) := \exp(-\eta \cdot C_j x)$. Let $w_p(x) := \sum_{i=1}^{m_p} w_p(x, i)$ and $w_c(x) := \sum_{j=1}^{m_c} w_c(x, j)$.

Definition 3.2. *Consider any $k \in [n]$. The* cost *of coordinate k at a given $x \in \mathbb{R}^n_{\geq 0}$ is defined as:*

$$\lambda(x, k) := \frac{\sum_{i=1}^{m_p} w_p(x, i) \cdot P(i, k)}{\sum_{j=1}^{m_c} w_c(x, j) \cdot C(j, k)}.$$

The coordinate $k \in [n]$ is said to be cheap *w.r.t. a given $x \in \mathbb{R}^n_{\geq 0}$ iff $\lambda(x, k) \leq (1 + \Theta(\epsilon)) \cdot w_p(x)/w_c(x)$.*

It turns out that the notion of a cheap coordinate, as per Definition 3.2, is identical to the one used in the last paragraph of Section 3.1.1 (see Property 3.3 and Property 3.4). The algorithm is described in Figure 8.

---

1. INITIALISE $x \leftarrow \vec{0} \in \mathbb{R}^n_{\geq 0}$.
2. WHILE $\min_{j \in [m_c]} C_j x < 1$:
3.   IF there is no cheap coordinate at x, THEN
4.   RETURN that the LP is infeasible.
5.   ELSE
6.   Let $k \in [n]$ be a cheap coordinate.
7. $x \leftarrow \text{BOOST}(x, k)$.   // See Figure 9.
8. RETURN x.

Figure 8: A static algorithm for solving a positive LP (P, C).

---

1. Find the maximum δ such that $\max_{i \in [m_p]} P(i, k) \cdot \delta \leq \epsilon/\eta$ and $\max_{j \in [m_c]: C_j x < 2} C(j, k) \cdot \delta \leq \epsilon/\eta$.
2. RETURN $x + \delta \vec{e}_k$.

Figure 9: $\text{BOOST}(x, k)$.

It is easy to check that this algorithm follows the framework outlined in Section 3.1.1. To summarise, the algorithm starts at $x = \vec{0}$. Subsequently, each iteration of the WHILE loop in Figure 8 either declares that the input LP is infeasible, or identifies a cheap coordinate w.r.t. the current x and calls the subroutine in Figure 9 to *boost* (i.e., change) x by a small amount along that direction. It stops when $\min_{j \in [m_c]} C_j x \geq 1$.

We now state two key properties of the algorithm, whose proofs are deferred to Appendix B. Property 3.3 justifies the algorithm's decision to declare that the input LP is infeasible if it cannot find any cheap coordinate w.r.t. the current $x \in \mathbb{R}^n_{\geq 0}$. Property 3.4 ensures that whenever the algorithm decides to *boost* the current x along a cheap coordinate $k \in [n]$, the increase in $f_p(x)$ is at most $(1 + \Theta(\epsilon))$ times the increase in $f_c(x)$, which, in turn, is itself at most $\Theta(\epsilon)$.

Property 3.3. *Consider any $x \in \mathbb{R}^n_{\geq 0}$. If there is no cheap coordinate at x, then the LP (P, C) is infeasible.*

Property 3.4. *Assume $\epsilon < 1/10$ and let $k \in [n]$ be a cheap coordinate at $y \in \mathbb{R}^n_{\geq 0}$, and let $z \leftarrow \text{BOOST}(y, k)$. Then we have:*

$$f_p(z) - f_p(y) \leq (1 + \Theta(\epsilon)) \cdot (f_c(z) - f_c(y)) \leq \Theta(\epsilon).$$

Recall that initially $x = \vec{0}$, and so $f_p(x) = f_c(x) = 0$. Each subsequent iteration of the WHILE loop in Figure 8 boosts the current x along a cheap coordinate. Hence, Property 3.4 implies the following invariant:

Invariant 3.5. *During the course of this algorithm, we always have $f_p(x) \leq (1+\Theta(\epsilon)) \cdot f_c(x)$. Furthermore, each call to the subroutine* BOOST(x, k) *increases the value of $f_c(x)$ by at most an additive $\Theta(\epsilon)$.*

Corollary 3.6. *If the algorithm in Figure 8 returns an $x \in \mathbb{R}^n_{\geq 0}$ in step 8, then x is indeed a $\Theta(\epsilon)$-approximate solution to the input LP. Otherwise, it correctly declares that the input LP is infeasible.*

Proof. Case (i): The algorithm terminates in step 8 of Figure 8. All the claims made in the next paragraph hold because of (3.3) and Invariant 3.5.

Just before the last iteration of the WHILE loop in Figure 8, we had $f_p(x) \leq (1 + \Theta(\epsilon)) \cdot f_c(x)$ and $f_c(x) \leq \min_{j \in [m_c]} C_j x < 1$. Accordingly, at the end of the last iteration of the concerned WHILE loop, we have $\max_{i \in [m_p]} P_i x \leq f_p(x) \leq (1 + \Theta(\epsilon)) \cdot f_c(x) \leq (1 + \Theta(\epsilon)) \cdot (1 + \Theta(\epsilon)) \leq 1 + \Theta(\epsilon)$, and $\min_{j \in [m_c]} C_j x \geq 1$. Hence, the x that is returned in step 8 of Figure 8 is a $\Theta(\epsilon)$-approximate solution to the input LP.

Case (ii): The algorithm terminates because during some iteration of the WHILE *loop in Figure 8 it cannot find any cheap coordinate w.r.t. the current x.* In this case, by Property 3.3, the input LP is infeasible. $\square$

3.2 An Overview of Our Dynamic Greedy MWU Algorithm

We describe the basic template that will be followed by our dynamic algorithm in Section 3.2.1. For ease of exposition, in Section 3.2.2 we describe and analyse our dynamic algorithm in a setting where all the relaxing entry updates occur to the covering constraints, whereas the packing constraints remain unchanged. Finally, in Section 3.2.3 we present an overview of the complete algorithm which can handle relaxing entry updates to *both* the packing and covering constraints.

3.2.1 The basic template for our dynamic algorithm

Initialisation: At preprocessing, we start by implementing the procedure described in Figure 8. At this stage, if we end up with a $\Theta(\epsilon)$-approximate solution x to the positive LP, then we are done. This is because such an x will continue to remain a $\Theta(\epsilon)$-approximate solution to the input LP even after it undergoes any sequence of relaxing updates in future. Accordingly, henceforth we assume that the procedure in Figure 8 ends up with an x at which there is no cheap coordinate.

Handling a relaxing update: After the update, we keep on boosting x along cheap coordinates until either (a) the set of cheap coordinates becomes empty again, or (b) we have $\min_{j \in [m_c]} C_j \geq 1$. In the former case, we declare that the input LP remains infeasible even after the relaxing update. In the latter case, we declare the current x as a $\Theta(\epsilon)$-approximate solution to the input LP from this point onward.

Clearly, as per the discussion in Section 3.1, any dynamic algorithm following the above template solves the problem of maintaining a $\Theta(\epsilon)$-approximate solution to a positive LP under relaxing updates.

3.2.2 Handling a sequence of relaxing entry updates to the covering constraints

We now present an overview of our dynamic algorithm, under the assumption that every relaxing update increases the value of some entry in the matrix C (whereas the matrix P remains unchanged). We follow the template outlined in Section 3.2.1. The main challenge is to define the appropriate data structures, and to describe how to *choose* a cheap coordinate for boosting x at any given point in time.

Before proceeding any further, note that there are two types of *events* which influence the outcome of any dynamic algorithm following the template from Section 3.2.1: (1) a relaxing update which increases the

value of some entry in the matrix C, and (2) a call to the subroutine $\text{BOOST}(x, k)$ which *boosts* the current x along the coordinate k. We start with the key invariant that drives the analysis of our dynamic algorithm.

Invariant 3.7. *During the course of our dynamic algorithm, we always have $f_p(x) \leq (1 + \Theta(\epsilon)) \cdot f_c(x)$. Furthermore, every call to the subroutine $\text{BOOST}(x, k)$ increases $f_c(x)$ by at most an additive $\Theta(\epsilon)$.*

Proof. The algorithm starts at $x = \vec{0}$, where $f_p(x) = f_c(x) = 0$ and so the invariant holds. Subsequently, the values of $f_p(x)$ and $f_c(x)$ can change because of one of the two following types of events.

(1) A relaxing update increases the value of an entry $C(j, k)$ of the matrix C, for $j \in [m_c]$, $k \in [n]$. This increases the value of $C_j x$; whereas for all $j' \in [m_c] \setminus \{j\}$ and $i \in [m_p]$, the values of $C_{j'} x$ and $P_i x$ remain unchanged. We accordingly infer that $f_c(x)$ increases, whereas $f_p(x)$ remains unchanged. Thus, if the invariant was true just before this event, then it continues to remain true just after the event.

(2) The algorithm make a call to $\text{BOOST}(x, k)$, for some $k \in [n]$. By Property 3.4, due to this event the increase in $f_p(x)$ is at most $(1 + \Theta(\epsilon))$ times the increase in $f_c(x)$, and this latter quantity is at most $\Theta(\epsilon)$. Thus, if the invariant was true just before this event, then it continues to remain true just after the event. $\square$

Corollary 3.8. *During the course of our dynamic algorithm, we always have $\max_{i \in [m_p]} P_i x \leq 1 + \Theta(\epsilon)$.*

Proof. Follows from (3.3), Invariant 3.7, and the fact that we return x the moment $\min_{j \in [m_c]} C_j x$ is ≥ 1. $\square$

Next, we derive a few important observations which show that the weights $w_p(x, i), w_c(x, j)$ of the constraints and the costs $\lambda(x, k)$ of the coordinates change (almost) monotonically over time.

Observation 3.9. *Consider an event which consists of either a relaxing update to the matrix C or a call to the subroutine $\text{BOOST}(x, k)$. Such an event can only increase the weight $w_p(x, i)$ of a packing constraint $i \in [m_p]$, and it can only decrease the weight $w_c(x, j)$ of a covering constraint $j \in [m_c]$.*

Proof. Consider a relaxing update which increases the value of some entry $C(j, k)$ of the matrix C. This increases the value of $C_j x$; whereas for all $j' \in [m_c] \setminus \{j\}$ and $i \in [m_p]$, the values of $C_{j'} x$ and $P_i x$ remain unchanged. Accordingly, this event decreases the weight $w_c(x, j)$; whereas for all $j' \in [m_c] \setminus \{j\}$ and $i \in [m_p]$, the weights $w_c(x, j')$ and $w_p(x, i)$ remain unchanged due to this event.

Next, consider a call to $\text{BOOST}(x, k)$. For all $i \in [m_p]$ with $P(i, k) > 0$ this increases the value of $P_i x$, whereas for all $i \in [m_p]$ with $P(i, k) = 0$ the value of $P_i x$ remains unchanged. Similarly, for all $j \in [m_c]$ with $C(j, k) > 0$ this increases the value of $C_j x$, whereas for all $j \in [m_c]$ with $C(j, k) = 0$ the value of $C_j x$ remains unchanged. In effect, this means that for all $i \in [m_p]$ the weight $w_p(x, i)$ can only increase due to this event, whereas for all $j \in [m_p]$ the weight $w_c(x, j)$ can only decrease due to this event. $\square$

Observation 3.10. *Consider an event which consists of a call to the subroutine $\text{BOOST}(x, k)$ for some $k \in [n]$. Because of this event, the cost $\lambda(x, k')$ of any coordinate $k' \in [n]$ can only increase.*

Proof. Follows from Observation 3.9 and Definition 3.2. $\square$

Observation 3.11. *Consider a relaxing update to the entry $C(j, k)$ of the matrix C, where $j \in [m_c]$, $k \in [n]$. Because of this event, the cost $\lambda(x, k')$ of a coordinate $k' \in [n] \setminus \{k\}$ can only increase, whereas the cost of the coordinate k can change in either direction (i.e., it can increase or decrease).*

Proof. Because of this relaxing update, the value of $C(j, k)$ increases, whereas the values of all other entries in the matrices C and P remain unchanged. The proof follows from Observation 3.9 and Definition 3.2. $\square$

Key data structures: We will explicitly maintain the following quantities: (1) The point $x \in \mathbb{R}^n_{\geq 0}$. (2) The weights $w_p(x, i), w_c(x, j)$ for all $i \in [m_p], j \in [m_c]$. (3) The total weights $w_p(x)$ and $w_c(x)$. (4) The values of $P_i x$ and $C_j x$, for all $i \in [m_p]$ and $j \in [m_c]$.

We are now ready to describe the exact implementation of our dynamic algorithm in more details.

Phases: Observation 3.9 implies that the ratio $\gamma(x) := w_p(x)/w_c(x)$ increases monotonically over time during the course of our algorithm. Armed with this observation, we split the working of our algorithm into *phases*. In the beginning, we have $x = \vec{0}$ and hence $\gamma(x) = w_p(x)/w_c(x) = 1/1 = 1$. We initiate a new phase whenever the value of $\gamma(x)$ increases by a multiplicative factor of $(1 + \Theta(\epsilon))$. Thus, a given phase can span a sequence of calls to the BOOST(x, k) subroutine at preprocessing, or it can also span a sequence of calls to the BOOST(x, k) subroutine interspersed with a sequence of relaxing updates to the matrix C.

Implementing a given phase: Let $\gamma^0(x)$ be the value of $\gamma(x) := w_p(x)/w_c(x)$ at the start of the phase. During the phase, $\gamma(x)$ does not change by more than an $(1 + \Theta(\epsilon))$ factor. Accordingly, at any point in time within the phase, we classify a coordinate $k \in [n]$ as being cheap iff $\lambda(x, k) \leq (1 + \Theta(\epsilon)) \cdot \gamma^0(x)$.

We maintain a set $E \subseteq [n]$ that always contains all the coordinates that are currently cheap. At the start of the phase, we initialise $E \leftarrow [n]$, and then call the subroutine BOOST-ALL(x, E) which is described in Figure 10. Note that at the end of any iteration of the inner WHILE loop (steps $4 - 9$ in Figure 10), the concerned coordinate k is no longer cheap. Furthermore, the monotonicity of the costs (as captured by Observation 3.10) imply that a coordinate k, once removed from the set E, does *not* become cheap again within the same phase due to some future iteration of the outer WHILE loop in Figure 10.

Handling a relaxing update within the phase: Consider a relaxing update to an entry $C(j, k)$ of the matrix C. Just before this update, we had $E = \emptyset$. If the value of $\gamma(x)$ becomes more than $(1 + \Theta(\epsilon))\gamma^0(x)$ because of this update, then we terminate the current phase and initiate a new one. Otherwise, by Observation 3.11, the only coordinate that can become cheap because of this update is k. Accordingly, we set $E \leftarrow \{k\}$, and call the subroutine BOOST-ALL(x, E). Observation 3.10 again implies that as we keep boosting x along the coordinate k, it does not lead to any other coordinate $k' \in [n] \setminus \{k\}$ becoming cheap.

```
1.  WHILE E ≠ ∅:
2.      Consider any k ∈ E.
3.      E ← E \ {k}.
4.      WHILE λ(x, k) ≤ (1 + Θ(ε)) · γ⁰(x)
5.          x ← BOOST(x, k).              // See Figure 9.
6.          IF min_{j∈[m_c]} C_j x ≥ 1, THEN
7.              RETURN x.
8.          IF γ(x) > (1 + Θ(ε)) · γ⁰(x), THEN
9.              Terminate the current phase.
```

Figure 10: BOOST-ALL(x, E).

The next two lemmas capture a couple of crucial properties of this dynamic algorithm.

Lemma 3.12. *Fix any coordinate $k \in [n]$. The dynamic algorithm described above calls the subroutine* BOOST(x, k) *at most* $O\left(\frac{\log^2(m_c + m_p + U/L)}{\epsilon^2}\right)$ *many times.*

Proof. Fix a coordinate $k \in [n]$ for the rest of the proof.

Consider any call to the BOOST(x, k) subroutine during the course of our dynamic algorithm, which increases coordinate k of the vector $x \in \mathbb{R}^n_{\geq 0}$ by some amount $\delta > 0$. From Figure 9, observe that just before

this specific call to the subroutine, either there exists a packing constraint $i \in [m_p]$ with $P(i, k) \cdot \delta = \epsilon/\eta$, or there exists a *not-too-large* covering constraint $j \in [m_c]$[14] with $C(j, k) \cdot \delta = \epsilon/\eta$. In the former (resp. latter) case, we refer to the index i (resp. j) as the *pivot* and the current value of $P(i, k)$ (resp. $C(j, k)$) as the *pivot-value* corresponding to this specific call to the subroutine $\text{BOOST}(x, k)$.[15]

Without loss of generality, assume that $U = 2^\tau \cdot L$ where $\tau = \log(U/L)$ is an integer, and that the value of any nonzero entry in the constraint matrix lies in the interval $(L, U]$ at all times. Partition the interval $(L, U]$ into τ segments $\mathcal{I}_0, \ldots, \mathcal{I}_{\tau-1}$, where $\mathcal{I}_\ell = (L \cdot 2^\ell, L \cdot 2^{\ell+1}]$ for all $\ell \in \{0, \ldots, \tau-1\}$.

Claim 3.1. *Consider any $\ell \in \{0, \ldots, \tau-1\}$. Throughout the duration of our dynamic algorithm, at most $4\eta/\epsilon$ calls with pivot-values in $\mathcal{I}_\ell$ and pivots in $[m_c]$ are made to the $\text{BOOST}(x, k)$ subroutine (see (3.2)).*

Proof. Let $x_k \in \mathbb{R}_{\geq 0}$ denote the kth coordinate of the vector $x \in \mathbb{R}_{\geq 0}^n$. For ease of exposition, we say that a call to $\text{BOOST}(x, k)$ is *covering-critical* iff its pivot is in $[m_c]$ and its pivot-value is in $\mathcal{I}_\ell$. We wish to upper bound the total number of covering-critical calls made during the course of our dynamic algorithm.

Consider any specific covering-critical call to $\text{BOOST}(x, k)$ with pivot $j \in [m_c]$ and pivot-value $\alpha \in \mathcal{I}_\ell = (L \cdot 2^{\ell-1}, L \cdot 2^\ell]$. From Figure 9, it follows that this call increases x_k by $\delta = \epsilon/(\eta\alpha) \geq \epsilon/(\eta L 2^\ell)$.

Suppose that T covering-critical calls have been made to the $\text{BOOST}(x, k)$ subroutine, where $T > 4\eta/\epsilon$. Let us denote these calls in increasing order of time by: $\Gamma_1, \Gamma_2, \ldots, \Gamma_T$. Thus, for all $t \in [T]$, we let Γ_t denote the tth covering-critical call made during the course of our dynamic algorithm. From the discussion in the preceding paragraph, each of these critical calls Γ_t increases x_k by at least $\epsilon/(\eta L 2^\ell)$.

Initially $x_k = 0$, and it increases monotonically over time. Hence, just before the last critical call Γ_T we have $x_k \geq (T-1) \cdot \epsilon/(\eta L 2^\ell) \geq (4\eta/\epsilon) \cdot \epsilon/(\eta L 2^\ell) = 1/(L 2^{\ell-2})$. Let $j_T \in [m_c]$ be the pivot of the call Γ_T. Since the pivot-value of Γ_T lies in $\mathcal{I}_\ell$, it follows that $C(j_T, k) > L \cdot 2^{\ell-1}$ just before the call Γ_T. Thus, just before the call Γ_T, we have $C_{j_T} \cdot x \geq C(j_T, k) \cdot x_k > L 2^{\ell-1} \cdot 1/(L 2^{\ell-2}) = 2$. But if $C_{j_T} \cdot x > 2$ just before the call Γ_T, then j_T cannot be the pivot of Γ_T (see Figure 9). This leads to a contradiction. Hence, we must have $T \leq 4\eta/\epsilon$, and this concludes the proof of the claim. $\square$

Claim 3.2. *Consider any $\ell \in \{0, \ldots, \tau-1\}$. Throughout the duration of our dynamic algorithm, at most $4\eta/\epsilon$ calls with pivot-values in $\mathcal{I}_\ell$ and pivots $i \in [m_p]$ are made to the $\text{BOOST}(x, k)$ subroutine (see (3.2)).*

Proof. (Sketch) As in the proof of Claim 3.1, let $x_k \in \mathbb{R}_{\geq 0}$ denote the kth coordinate of the vector $x \in \mathbb{R}_{\geq 0}^n$. Say that a call to $\text{BOOST}(x, k)$ is *packing-critical* iff its pivot is in $[m_p]$ and its pivot-value is in $\mathcal{I}_\ell$.

Suppose that T covering-critical calls have been made to the $\text{BOOST}(x, k)$ subroutine, where $T > 4\eta/\epsilon$. Let us denote these calls in increasing order of time by: $\Gamma_1, \Gamma_2, \ldots, \Gamma_T$. Thus, for all $t \in [T]$, we let Γ_t denote the tth packing-critical call made during the course of our dynamic algorithm. Let $i_T \in [m_p]$ be the pivot of the last packing-critical call Γ_T. Following the same argument as in the proof of Claim 3.1, we conclude that $P_{i_T} \cdot x > 2$ just before the call Γ_T. But if $P_{i_T} \cdot x > 2$, then $f_p(x) \geq \max_{i \in [m_p]} P_i x > 2$ according to (3.3). Now, applying Invariant 3.7, we conclude that just before the call Γ_T we have: $\min_{j \in [m_c]} C_j x \geq f_c(x) \geq (1+\epsilon)^{-1} \cdot f_p(x) > (1+\epsilon)^{-1} \cdot 2 > 1$. This leads to a contradiction, since if $\min_{j \in [m_c]} C_j x > 1$, then our dynamic algorithm would immediately declare that it has found a $\Theta(\epsilon)$-approximate solution to the input LP from this point onward, without making any further calls to the $\text{BOOST}(x, k)$ subroutine (in particular, without making the call Γ_T). Thus, it must be the case that $T \leq 4\eta/\epsilon$, and this concludes the proof of the claim. $\square$

---

[14]We say that a covering constraint $j \in [m_c]$ is *not-too-large* iff $C_j x < 2$.

[15]Note that the values of the entries in the constraint matrix change over time due to the sequence of updates. The *pivot-value* of a call to $\text{BOOST}(x, k)$ refers to the value of the concerned entry in the constraint matrix *just before* the specific call to $\text{BOOST}(x, k)$.

Any call to the $\text{BOOST}(x, k)$ subroutine has its pivot in $[m_p] \cup [m_c]$ and pivot-value in $\mathcal{I}_\ell$, for some $\ell \in \{0, \ldots, \tau - 1\}$.[16] Thus, Claim 3.1 and Claim 3.2 imply that the total number of calls made to the $\text{BOOST}(x, k)$ subroutine is at most $2 \cdot \tau \cdot (4\eta/\epsilon) = O\left(\frac{\log^2(m_c + m_p + U/L)}{\epsilon^2}\right)$. $\qquad\square$

Lemma 3.13. *The dynamic algorithm described above has at most* $O\left(\frac{\log(m_c + m_p + U/L)}{\epsilon^2}\right)$ *many phases.*

Proof. By Observation 3.9, the ratio $\gamma(x) := w_p(x)/w_c(x)$ increases monotonically over time. Initially, we have $x = \vec{0}$, and hence $\gamma(x) = w_p(x)/w_c(x) = 1$. Let κ be the time-instant at which the last ever call to $\text{BOOST}(x, k)$ is made by our dynamic algorithm. We will upper bound $\gamma(x)$ just before the time-instant κ.

Just before the time-instant κ, by Corollary 3.8 we have $\max_{i \in [m_p]} P_i x \le 1 + \Theta(\epsilon)$ and hence $w_p(x, i) \le \exp(\eta(1 + \Theta(\epsilon)))$ for all $i \in [m_p]$, which implies that $w_p(x) = \sum_{i=1}^{m_p} w_p(x, i) \le m_p \cdot \exp(\eta(1 + \Theta(\epsilon)))$.

Just before the time-instant κ, we also have $\min_{j \in [m_c]} C_j x < 1$. Hence, at that moment there exists some $j' \in [m_c]$ with $C_{j'} x < 1$ and $w_c(x, j') \ge 1/\exp(\eta)$. This means that $w_c(x) \ge w_c(x, j') \ge 1/\exp(\eta)$.

To summarise, we infer that $\gamma(x) = w_p(x)/w_c(x) \le m_p \cdot \exp(\eta(1 + \Theta(\epsilon))) \cdot \exp(\eta) = O(m_p \cdot \exp(3 \cdot \eta))$ just before the time-instant κ. Thus, the value of $\gamma(x)$ increases from 1 at the start of the algorithm to at most $\Lambda = O(\exp(3 \cdot \eta) \cdot m_p)$ just before the last call to the $\text{BOOST}(x, k)$ subroutine.

Note that $\gamma(x)$ increases monotonically over time in the range $[1, \Lambda]$, and we create a new phase whenever $\gamma(x)$ increases by a $(1 + \Theta(\epsilon))$ factor. Thus, from (3.2), we conclude that the total number of phases is at most $\log_{(1+\Theta(\epsilon))} \Lambda = O\left(\frac{\log(m_p + m_c + U/L)}{\epsilon^2}\right)$. $\qquad\square$

Analysing the total update time: We first focus on bounding the total time spent on all the calls to the $\text{BOOST}(x, k)$ subroutine during the course of our algorithm. By Lemma 3.12, the point x gets boosted along a given coordinate $k \in [n]$ at most $\Gamma = O\left(\frac{\log^2(m_c + m_p + U/L)}{\epsilon^2}\right)$ times. Whenever we boost x along some coordinate k, we need to spend $O(nz(k))$ time to update all the relevant weights $\{w_p(x, i)\}$, $\{w_c(x, j)\}$, $w_p(x)$, $w_c(x)$ and the values $\{C_j x\}, \{P_i x\}$; where $nz(k)$ is the total number of nonzero entries in the kth column of the matrices P, C at the current moment. Hence, the total time spent in all the calls to the $\text{BOOST}(x, k)$ subroutine is at most $\sum_{k=1}^{n} \Gamma \cdot O(nz(k)) = \Gamma \cdot O(N) = O\left(N \cdot \frac{\log^2(m_c + m_p + U/L)}{\epsilon^2}\right)$.

Next, note that whenever we create a new phase, we set $E \leftarrow [n]$ and call the subroutine $\text{BOOST-ALL}(x, E)$ as in Figure 10. Also, whenever there is a relaxing update to an entry $C(j, k)$ in the covering matrix, we set $E \leftarrow \{k\}$ and then make a call to $\text{BOOST-ALL}(x, E)$. Thus, it might very well be the case that a coordinate $k \in [n]$ moves in and out of the set E on multiple occasions, without being boosted at all. The time spent on these *apparently futile* operations, which modify the set E without leading to any call to $\text{BOOST}(x, k)$, is captured by steps 1–3 of Figure 10. Lemma 3.13 implies that the total time spent in this manner, throughout a sequence of t relaxing updates and across all the coordinates, is at most:

$$
\begin{aligned}
(\text{number of phases}) \cdot O(\text{number of coordinates}) + O(t) &= O\left(n \cdot \frac{\log(m_c + m_p + U/L)}{\epsilon^2} + t\right) \\
&= O\left(N \cdot \frac{\log(m_c + m_p + U/L)}{\epsilon^2} + t\right).
\end{aligned}
$$

Finally, note that the preprocessing time is at most $O\left(N \cdot \frac{\log(m_c + m_p)}{\epsilon^2}\right)$, since the static algorithm from Section 3.1 can be implemented in $O\left(N \cdot \frac{\log(m_c + m_p)}{\epsilon^2}\right)$ time [You14]. This leads us to Theorem 3.1 for relaxing entry updates to covering constraints.

---

[16]This holds because the pivot-values lie in the range $(L, U]$, and this range has been partitioned into subintervals: $\mathcal{I}_0, \ldots, \mathcal{I}_{\tau-1}$.

3.2.3 The full algorithm: Handling relaxing entry updates to both packing and covering constraints

We start by identifying the main difficulty in extending the dynamic algorithm from Section 3.2.2 to the setting where relaxing updates occur to *both* packing and covering constraints of the input positive LP. Consider a relaxing update to the entry $P(i, k)$, for some $i \in [m_p], k \in [n]$. Due to this relaxing update, the value of $P(i, k)$ gets reduced, and accordingly the weight $w_p(x, i)$ also decreases. This contradicts the monotonicity of the weights as captured in Observation 3.9. Thus, $w_p(x)$ no longer increases monotonically with time, which in turn invalidates the proof of Lemma 3.13. We circumvent this difficulty by working with an *extended* positive LP (P^*, C^*), which is derived from the input LP (P, C) in the following manner.

The extended LP: Throughout the sequence of updates to the input LP, we ensure that we always have:

- $P^* \in \mathbb{R}_{\geq 0}^{m_p \times (n+1)}$, where $P^*(i, k) = P(i, k)$ and $P^*(i, n+1) \geq 0$ for all $i \in [m_p], k \in [n]$.

- $C^* \in \mathbb{R}_{\geq 0}^{m_c \times (n+1)}$, where $C^*(j, k) = C(j, k)$ and $C^*(j, n+1) = 0$ for all $j \in [m_c], k \in [n]$.

It is easy to check that the input LP (P, C) is feasible iff the extended LP (P^*, C^*) is feasible. Furthermore, if $x^* = (x_1^*, \ldots, x_{n+1}^*) \in \mathbb{R}_{\geq 0}^{n+1}$ is a $\Theta(\epsilon)$-approximate solution to the extended LP (P^*, C^*), then $x = (x_1^*, \ldots, x_n^*) \in \mathbb{R}_{\geq 0}^n$ is a $\Theta(\epsilon)$-approximate solution to the input LP (P, C). Accordingly, our new dynamic algorithm will attempt to approximately solve the extended LP (P^*, C^*). Note that we are free to set the values $P^*(i, n+1)$ of the matrix P^* in any way we chose (provided they remain nonnegative). We will use this to our advantage while dealing with relaxing updates to the packing constraints in the input LP. It is important to emphasise that except the entries $P^*(i, n+1)$, at any point in time the values of all other entries in the matrices P^* and C^* are determined by the current status of the input LP (P, C).

Initialisation: At preprocessing, we set $P^*(i, n+1) \leftarrow 0$ for all $i \in [m_p]$. We start with an $x^* = (x_1^*, \ldots, x_{n+1}^*) \in \mathbb{R}_{\geq 0}^{n+1}$ where $x_1^* = \cdots = x_n^* = 0$ and $x_{n+1}^* = 1$. At this point in time, we have $f_p(x^*) = f_c(x^*) = 0$. From this point onward, we will keep modifying x^* until it becomes a $\Theta(\epsilon)$-approximate solution to the extended LP (P^*, C^*), by following the same template as outlined in Section 3.2.1 and Section 3.2.2. In addition, now after every relaxing update to an entry in the packing matrix (say) $P(i, k)$, we will change the value of $P^*(i, n+1)$ in such a way that works to our advantage. This is explained in more details below. For clarity of exposition, henceforth we assume that after an update to an entry $P(i, k)$ (resp. $C(j, k)$) of the matrix P (resp. C), the corresponding entry $P^*(i, k)$ (resp. $C^*(j, k)$) implicitly gets updated so as to ensure the equality $P^*(i, k) = P(i, k)$ (resp. $C^*(j, k) = C(j, k)$).

Pseudo-updates to the extended LP: Consider any relaxing update to an entry $P(i, k)$ of the matrix P, where $i \in [m_p], k \in [n]$. Immediately after this relaxing update, we *increase* the value of $P^*(i, n+1)$ to such an extent that the value of $P_i^* x^*$ (and hence the weight $w_p(x^*)$) remains unchanged. We refer to this step as a *pseudo-update* to the extended LP. The only purpose behind the pseudo-update is to ensure that the monotonicity of the weights as captured by Observation 3.9 continues to hold.

In summary, the overall algorithm follows the same template as in Section 3.2.1 and Section 3.2.2, with the following caveat: Immediately after every relaxing update to a packing constraint in the input LP, we perform the corresponding pseudo-update to the extended LP so as to ensure the validity of Observation 3.9. It is easy to check that all the observations, lemmas and inferences derived in Section 3.2.2 continue to hold here. This leads us to the dynamic algorithm promised in Theorem 3.1, for relaxing entry updates.

3.3 Handling Restricting (resp. Relaxing) Updates to a Packing (resp. Covering) LP

Consider the setting where we wish to maintain a solution to Problem 2.1 when the matrix C is undergoing relaxing entry updates (or equivalently, the matric $C^\top$ is undergoing restricting entry updates). Thus, each

update increases the value of some entry of C. The idea is simple. We run the dynamic algorithm from Section 3.2 on the positive LP defined by (3.4) and (3.5), which is undergoing relaxing entry updates.

$$\mathbb{1}^\top x \leq 1 \tag{3.4}$$
$$Cx \geq \mathbb{1} \tag{3.5}$$

Suppose that this algorithm has just finished processing an update. Consider two possible cases.

Case I: The algorithm returns a solution $x \in \mathbb{R}^n_{\geq 0}$ which satisfies $\mathbb{1}^\top x \leq 1 + \Theta(\epsilon)$ and $Cx \geq (1 - \Theta(\epsilon)) \cdot \mathbb{1}$. In this case, we simply return the vector x and terminate our algorithm. The vector x will continue to remain an ϵ-approximate solution to the input LP after any future update.

Case II: The algorithm declares that the positive LP is infeasible as there is no cheap co-ordinate. From Section 3.1.2, recall that $w_c(x, j)$ denotes the weight of a covering constraint $j \in [m]$, and that $w_c(x) = \sum_{j \in [m]} w_c(x, j)$ denotes the total weight of all the covering constraints. Define the vector $\hat{y} \in \mathbb{R}^m_{\geq 0}$, where $\hat{y}_j = w_c(x, j)/w_c(x)$ for all $j \in [m]$. We now claim that $\hat{y}$ is a feasible solution to the dual packing LP.

Claim 3.3. *We have $\mathbb{1}^\top \hat{y} = 1$ and $C^\top \hat{y} \leq \mathbb{1}$.*

Proof. Since $\{\hat{y}_j\}$ denotes the normalized weights of the covering constraints, we have $\mathbb{1}^\top \hat{y} = 1$. Next, fix any $k \in [n]$. Since we are in Case II, the coordinate k is *not* cheap. So, from Definition 3.2, we infer that:

$$\frac{1}{\sum_{j \in [m]} \hat{y}_j \cdot C(j,k)} \geq 1 + \Theta(\epsilon), \text{ or equivalently, } \left(C^\top \hat{y}\right)_k = \sum_{j \in [m]} \hat{y}_j \cdot C(j,k) \leq 1 - \Theta(\epsilon) \leq 1.$$

Accordingly, we get $C^\top \hat{y} \leq \mathbb{1}$. $\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\square$

Our algorithm maintains an estimate $w_c^*(x) \in [w_c(x), (1 + \epsilon)w_c(x)]$. If after processing an update, we end up in Case II, then we return a $y \in \mathbb{R}^m_{\geq 0}$, where $y_j := w_c(x, j)/w_c^*(x)$ for all $j \in [m]$. As $\hat{y}_j \leq y_j \leq (1 + \Theta(\epsilon)) \cdot \hat{y}_j$ for all $j \in [m]$, Claim 3.3 implies that $\mathbb{1}^\top y \geq 1$ and $C^\top y \leq (1 + \Theta(\epsilon)) \cdot \mathbb{1}$.

It is easy to maintain the vector y without any significant overhead in the total update time, because of three reasons: (1) the algorithm from Section 3.2 explicitly maintains the weight $w_c(x, j)$ for each $j \in [m]$, (2) we do not need to change the estimate $w_c^*(x)$ as long as we remain in the same phase, and (3) by Lemma 3.13 we have at most $O\left(\frac{\log(m+U/L)}{\epsilon^2}\right)$ many phases. Note that whenever we start a new phase we have to spend $O(m)$ time to update the vector y. Theorem 3.14 now follows from Theorem 3.1.

Theorem 3.14. *We can deterministically maintain a solution to Problem 2.1 when the matrix C undergoes t relaxing entry updates in $O\left(t + N \cdot \frac{\log^2(m+U/L)}{\epsilon^2} + m \cdot \frac{\log(m+U/L)}{\epsilon^2}\right) = O\left(t + N \cdot \frac{\log^2(m+U/L)}{\epsilon^2}\right)$ total time, where N denotes the maximum number of non-zero entries in C throughout these updates.*

Finally, we consider the problem of maintaining an ϵ-approximate optimal solution to a generic covering (resp. packing) LP under relaxing (resp. restricting) updates. It is easy to verify that if we start with Theorem 3.14 and apply the same reduction outlined in Section 2.5.2, then we obtain the following result.

Theorem 3.15. *Consider any sequence of t relaxing updates to LP (2.13), where each update either increases an entry of C, or decreases an entry of a, b. Throughout these updates, let N denote the maximum number of non-zero entries in C, and let L (resp. U) be a lower (resp. upper) bound on the minimum (resp. maximum) value of a non-zero entry of C, a, b. We can deterministically maintain an ϵ-approximate optimal solution to LP (2.13) in $O\left(t \cdot \frac{\log(nU/L)}{\epsilon} + N \cdot \frac{\log^2(m+U/L)}{\epsilon^3} \cdot \log(nU/L)\right)$ total time. The same guarantee also holds for maintaining an ϵ-approximate solution to LP (2.14) under restricting updates.*

4 Conditional Lower Bounds

In previous sections, we presented partially dynamic algorithms for packing-covering LP-s with polylog-arithmic amortized update times. This leads to a couple of natural questions: (1) Can we solve the same problem in polylogarithmic *worst case* update time? (2) Can we design a *fully dynamic* algorithm for packing-covering LPs in polylogarithmic *amortized* update time? We will now provide conditional lower bounds that rule out the possibility of having these stronger guarantees.

Our lower bounds hold under Strong Exponential Time Hypothesis (SETH), and the starting points of our reductions are Problem 4.1 and Theorem 4.2 described below [ARW17].

Problem 4.1. *As input, we get four parameters N, k, m, β^* such that $\Theta\left(N^{o(1)}\right) = \beta^* \leq k \leq m = \Theta\left(N^{o(1)}\right)$, and two collections of N vectors $A = \left\{a^{(1)}, \ldots, a^{(N)}\right\}$ and $B = \left\{b^{(1)}, \ldots, b^{(N)}\right\}$ such that $a, b \in \{0, 1\}^m$ for all $a \in A, b \in B$. Furthermore, each vector $b \in B$ has exactly k non-zero entries. We want to design an algorithm that can distinguish between the following two cases.*

- *(1) There exists some $a \in A$ and $b \in B$ such that $a \geq b$.*

- *(2) For all $a \in A$ and $b \in B$, we have $a \cdot b \leq k^* := \frac{k}{\beta^*}$.*

Theorem 4.2. *[ARW17] Under SETH, any algorithm for Problem 4.1 runs in $\Omega(N^{2-\epsilon})$ time for all $\epsilon > 0$.*

Now, we are ready to prove Theorem 1.2, and we state a more detailed version of it below.

Theorem 4.3. *Assume SETH holds. Let N and OPT respectively denote the number of non-zero entries in the input LP and its optimal objective value. Then there exists some $\beta = \Theta\left(N^{o(1)}\right)$ such that:*

- *(i) Any algorithm which maintains an estimate $\frac{\text{OPT}}{\beta} \leq \nu \leq \text{OPT}$ for a packing or covering LP under fully dynamic entry updates must have an amortized update time of $\Omega(N^{1-o(1)})$.*

- *(ii) Any algorithm which maintains an estimate $\frac{\text{OPT}}{\beta} \leq \nu \leq \text{OPT}$ for a packing or covering LP under partially dynamic entry updates must have a worst-case update time of $\Omega(N^{1-o(1)})$.*

Furthermore, the same lower bounds hold even under translation updates.

4.1 Proof of Theorem 4.3 - part (i)

We prove the theorem for packing LPs. The proof for covering LPs immediately follows via LP duality.

Recall the parameters N, k, m, β^* from the statement of Problem 4.1. Set $\beta := \beta^*/3$. Suppose that there is a dynamic algorithm $\mathcal{A}$ that maintains an estimate $\frac{\text{OPT}}{\beta} \leq \nu \leq \text{OPT}$ for a packing LP undergoing fully dynamic entry updates in $\Theta(T)$ amortized update time. Using $\mathcal{A}$ as a subroutine, we design a static algorithm $\mathcal{A}^*$ for Problem 4.1, as described below.

Upon receiving its input, the algorithm $\mathcal{A}^*$ defines the following packing LP, where initially we have $\lambda_j = 1$ for all $j \in [m]$. The algorithm $\mathcal{A}^*$ now feeds this packing LP as an input to $\mathcal{A}$ at pre-processing.

$$\text{Maximize} \sum_{j=1}^{m} x_j \tag{4.1}$$

$$\text{s.t.} \quad a \cdot x \ \leq \ k^* \qquad \text{for all } a \in A. \tag{4.2}$$

$$\lambda_j \cdot x_j \ \leq \ 1 \qquad \text{for all } j \in [m]. \tag{4.3}$$

$$x_j \ \geq \ 0 \qquad \text{for all } j \in [m]. \tag{4.4}$$

Subsequently, the algorithm $\mathcal{A}^*$ performs the steps outlined in Figure 11.

---

1. FOR ALL $b \in B$:
2.   Set $\lambda_j = L := m/k^*$ for all $j \in [m]$ with $b_j = 0$, and feed these updates to $\mathcal{A}$.
3.   IF $\mathcal{A}$ returns an estimate $\nu \leq 2k^*$, THEN
4.   REPORT that we are in case (1) of Problem 4.1, and TERMINATE.
5.   Return the LP to its initial state, by setting $\lambda_j = 1$ for all $j \in [m]$ with $b_j = 0$, and feeding these updates to the dynamic algorithm $\mathcal{A}$.
6. REPORT that we are in case (2) of Problem 4.1 and TERMINATE.

Figure 11: Solving Problem 4.1 using the dynamic algorithm $\mathcal{A}$.

Claim 4.1. *Suppose that $a \cdot b \leq k^*$ for all $a \in A, b \in B$. Then the algorithm $\mathcal{A}^*$ correctly reports that we are in case (2) of Problem 4.1.*

Proof. Consider any vector $b \in B$, and focus on the corresponding iteration of the FOR loop in Figure 11. After step (2) of this iteration, we have $\lambda_j = L$ for all $j \in [m]$ with $b_j = 0$, and $\lambda_j = 1$ for all $j \in [m]$ with $b_j = 1$. Now, define an x where $x_j = 1$ for all $j \in [m]$ with $b_j = 1$, and $x_j = 0$ for all $j \in [m]$ with $b_j = 0$. Such an x is a feasible solution to the LP and has an objective of $\sum_{j=1}^{m} x_j = |\{j \in [m] : b_j = 1\}| = k$. This implies that OPT $\geq k$. Accordingly, in step (3), the dynamic algorithm $\mathcal{A}$ returns a value $\nu \geq \frac{k}{\beta} = \frac{3k}{\beta^*} = 3k^*$. Hence, the algorithm $\mathcal{A}^*$ does *not* terminate during this iteration.

The preceding discussion implies that $\mathcal{A}^*$ does not terminate during any iteration of the FOR loop in Figure 11. Instead, at the end of the FOR loop, it correctly reports that we are in case (2) of Problem 4.1. □

Claim 4.2. *Suppose that there exists vectors $a \in A, b \in B$ such that $a \geq b$. Then the algorithm $\mathcal{A}^*$ correctly reports that we are in case (1) of Problem 4.1.*

Proof. Focus on the iteration of the FOR loop in Figure 11 which deals with the vector $b \in B$. After step (2) of this iteration, we have $\lambda_j = L$ for all $j \in [m]$ with $b_j = 0$, and $\lambda_j = 1$ for all $j \in [m]$ with $b_j = 1$. Partition the set $[m]$ into two subsets: $P_0 = \{j \in [m] : b_j = 0\}$ and $P_1 = \{j \in [m] : b_j = 1\}$. Since $a \geq b$, we have $a_j = 1$ for all $j \in P_1$. Let x be any feasible solution to LP (4.1). Observe that:

$$\sum_{j \in P_1} x_j = \sum_{j \in P_1} a_j x_j \leq a \cdot x \leq k^*. \tag{4.5}$$

Next, recall that $\lambda_j = L$ for all $j \in P_0$. Thus, we have $x_j \leq 1/L$ for all $j \in P_0$, and hence:

$$\sum_{j \in P_0} x_j \leq \frac{m}{L} = \kappa^*. \tag{4.6}$$

Summing (4.5) and (4.6), we now get: $\sum_{j=1}^{m} x_j \leq 2k^*$. Since this inequality holds for *every* feasible solution x to the LP, we get OPT $\leq 2k^*$. Accordingly, during step (3) of the concerned iteration of the FOR loop in Figure 11, the dynamic algorithm $\mathcal{A}$ returns a value $\nu \leq 2k^*$. Thus, the algorithm $\mathcal{A}^*$ terminates at this point, before correctly reporting that we are in case (1) of Problem 4.1. □

The correctness of algorithm $\mathcal{A}^*$ follows from Claim 4.1 and Claim 4.2. The claim below bounds its run time in terms of the update time of the dynamic algorithm $\mathcal{A}$.

Claim 4.3. *The algorithm $\mathcal{A}^*$ runs in $\hat{O}(N \cdot T)$ time, where T is the update time of $\mathcal{A}$.*

33

Proof. The algorithm $\mathcal{A}^*$ starts by feeding LP (4.1), which is of $\hat{O}(N)$ size, to $\mathcal{A}$ as input at pre-processing. Subsequently, during each iteration $b \in B$ of the FOR loop, it feeds $\Theta(m) = \hat{O}(1)$ updates to $\mathcal{A}$. Since the FOR loop runs for at most N iterations, overall the dynamic algorithm $\mathcal{A}$ needs to handle at most $\hat{O}(N)$ updates. Hence, the total time spent by the dynamic algorithm $\mathcal{A}$ is at most $\hat{O}(N \cdot T)$. The claims follows since the runtime of $\mathcal{A}^*$ is dominated by the total update time of $\mathcal{A}$. $\square$

Theorem 4.2 implies that assuming SETH, the run time of algorithm $\mathcal{A}^*$ must be $\Omega(N^{2-\epsilon})$ for every constant $\epsilon > 0$. Hence, from Claim 4.3 it follows that the amortized update time of the dynamic algorithm $\mathcal{A}$ is at least $T = \Omega(N^{1-o(1)})$. This concludes the proof of part-(i) of Theorem 4.3.

If we wish to prove the same lower bound for translation updates, then the reduction remains almost the same. The only difference now is that in step (2) of Figure 11, instead of changing the value of the coefficient λ_j we now change the constraints $x_j \leq 1$ to $x_j \leq 0$ for the concerned co-ordinates j, and un-do these changes in step (5) as before.

4.2   Proof of Theorem 4.3 - part (ii)

We prove the theorem for packing LPs. The proof for covering LPs immediately follows via LP duality.

As in Section 4.2, set $\beta := \beta^*/3$. Suppose that there is a dynamic algorithm $\mathcal{A}$ that maintains an estimate $\frac{\text{OPT}}{\beta} \leq \nu \leq \text{OPT}$ for a packing LP undergoing restricting entry updates in $\Theta(T)$ worst-case update time. Using $\mathcal{A}$ as a subroutine, we design a static algorithm $\mathcal{A}^*$ for Problem 4.1. To achieve this goal, we follow almost exactly the same strategy as in Section 4.2. The only difference being that in step (5) of Figure 11, instead of feeding a new set of updates to $\mathcal{A}$, we *roll back* the state of $\mathcal{A}$ to where it was at pre-processing. This allows us to deal with the fact that $\mathcal{A}$ can only handle restricting (*not* fully dynamic) updates. The rest of the arguments in the proof remain the same as in Section 4.2.

The preceding discussion implies the desired lower-bound when the input LP undergoes restricting updates. To derive the analogous lower-bound for relaxing updates, we need to make only the following minor tweak in our design of algorithm $\mathcal{A}^*$: Initially, we set $\lambda_j = L$ for all $j \in [m]$. Subsequently, in step (2) of Figure 11, we set $\lambda_j = 1$ for all $j \in [m]$ with $b_j = 1$, and feed these updates to $\mathcal{A}$. Next, in step (5) of Figure 11, we *roll back* the state of $\mathcal{A}$ to where it was at pre-processing.

Finally, we can easily extend our lower bounds to hold in settings where we only have translation updates, using the idea outlined in the last paragraph of Section 4.2.

A   Proofs of Theorem 2.12 and Lemma 2.13

Proof of Lemma 2.13: To conclude Lemma 2.13 notice that Lemma 2.9 seamlessly extends to the dynamic setting. The only property of the dynamic input matrix C the proof of Lemma 2.9 exploits is that at the time of the t-th WHACK sub-routine it holds that $(Cx^t)_{i_t} < 1$ which is similarly guaranteed in the dynamic setting. To argue it formally let C^t refer to the state of constraint matrix C at the time of the t-th WHACK call. Fix some $t \in [T]$. The following line of inequalities in analogous with the proof of Lemma 2.13.

$$\left\|\hat{x}^{t+1}\right\|_1 - \left\|\hat{x}^t\right\|_1 = \sum_{j \in [n]} \left(\left(\hat{x}^{t+1}\right)_j - \left(\hat{x}^t\right)_j \right) = \sum_{j \in [n]} \left(\hat{x}^t\right)_j \cdot \left(\epsilon \cdot \frac{C^t_{i_t,j}}{\lambda} \right) = \frac{\epsilon}{\lambda} \cdot \left(C^t \hat{x}^t\right)_{i_t} < \frac{\epsilon}{\lambda} \cdot \left\|\hat{x}^t\right\|_1$$

The last inequality follows since $\left(C^t x^t\right)_{i_t} < 1$ and $x^t := \hat{x}^t / \left\|\hat{x}^t\right\|_1$. Rearranging the terms, we get:

$$\left\|\hat{x}^{t+1}\right\|_1 \leq \left(1 + \frac{\epsilon}{\lambda}\right) \cdot \left\|\hat{x}^t\right\|_1$$

As $T = \frac{\lambda \ln(n)}{\epsilon^2}$ and $\left\| \hat{x}^0 \right\|_1 = n$ we get the following: $\left\| \hat{x}^t \right\|_1 \leq \left(1 + \frac{\epsilon}{\lambda}\right)^T \cdot \left\| \hat{x}^0 \right\|_1 \leq n^{(1/\epsilon)}$ for all $t \in [T]$.

Proof of Theorem 2.12: The extension of Theorem 2.6 to the dynamic setting is also relatively straightforward. Observe, that at the initialization of our dynamic algorithm we simply run the static implementation. Therefore before any update occurs Theorem 2.12 can be concluded from Theorem 2.6. We will first argue that if the dynamic algorithm returns vector $\tilde{x}^t$ then it must satisfy that $C \cdot \tilde{x}^t \geq \mathbb{1} \cdot (1 - \epsilon)$ and $|\tilde{x}^t|_1 \leq 1 + \epsilon$ even after some restricting updates have occurred.

Consider the starting point of a given phase at time τ. At this point $\mathbb{1}^\top \tilde{x}^\tau = \mathbb{1}^\top \hat{x}^\tau / \left\| \hat{x}^\tau \right\|_1 = 1$ and we fix $W = \left\| \hat{x}^\tau \right\|_1$. During the phase $\tilde{x}^t$ may only monotonously increase. However, if at any point $\left\| \hat{x}^t \right\|_1 > (1 - \epsilon/2)^{-1} \cdot W$ a new phase begins. As $\tilde{x}^t = \hat{x}^t / W$ during the phase $\left\| \tilde{x}^t \right\|_1 \leq (1 - \epsilon/2)^{-1} \leq 1 + \epsilon$.

Fix some $i \in [m]$. At the start of each phase the algorithm ensures that $(C \cdot \tilde{x}^t)_i \geq 1$ through enforcing all currently violating constraints. As during a phase $\tilde{x}^t$ may only monotonously increase until further restricting updates occur to constraint i it will remain satisfied. At all times the algorithm keeps an ϵ-approximate estimate of $(C \cdot \tilde{x}^t)$. Whenever a restricting update occurs to some constraint element $C_{i,j}$ and we observe that out estimate of the constraint value falls bellow 1 we enforce constraint i until $(C \cdot \tilde{x}^t)_i \geq 1$. Due to us keeping an ϵ-approximate estimate of $(C \cdot \tilde{x}^t)_i$ this will never allow it to fall bellow $1 - \epsilon$. Note that the value of $(C \cdot \tilde{x}^t)_i$ only depends on the i-th row of C and $\tilde{x}^t$ therefore if a restricting update occurs to constraint i and we decide not to enforce the constraint itself then the values of all other constraints remain unaffected.

It remains to argue that whenever the dynamic algorithm returns vector y following a series of updates it holds that $|y|_1 = 1$ and $C^\top y \leq \mathbb{1} \cdot (1 + 4 \cdot \epsilon)$. Once the algorithm has returned such a vector y it will no longer change its output. As C-s elements may only get reduced over time y will maintain these properties. Here we refer to the proof of Lemma 2.2 with slight deviations. Recall the *experts* setting from [AHK12] as presented in the proof Lemma 2.2. Further recall the statement of Lemma 2.4.

We map the whack-a-mole algorithm to the experts setting in the same manner however now accounting for the evolving constraint matrix. Let C^t stand for the state of C at the time of the t-th call to WHACK. Each covering constraint $j \in [n]$ corresponds to an expert, each iteration of the FOR loop in Figure 1 corresponds to a round, the vector $\hat{x}^t$ corresponds to the weight vector w^t at the start of round $t \in [T]$, and finally, the payoff for an expert $j \in [n]$ in round $t \in [T]$ is given by $\left(p^t\right)_j := (1/\lambda) \cdot C_{i_t,j}^t$.

Since $C_{i_t,j}^t \in [0, \lambda]$, we have $\left| \left(p^t\right)_j \right| = \left(p^t\right)_j$ for all $j \in [n], t \in [T]$. Thus, Lemma 2.4 implies that:

$$\sum_{t=1}^T \left(p^t\right)^\top \cdot x^t \geq \sum_{t=1}^T (1 - \epsilon) \cdot \left(p^t\right)_j - \frac{\ln(n)}{\epsilon}, \text{ for all experts } j \in [n]. \tag{A.1}$$

Diving both sides of the above inequality by T, and then rearranging the terms, we get:

$$\frac{(1 - \epsilon)}{T} \cdot \sum_{t=1}^T \left(p^t\right)_j \leq \frac{1}{T} \cdot \sum_{t=1}^T \left(p^t\right)^\top \cdot x^t + \frac{\ln(n)}{\epsilon \cdot T}, \text{ for all experts } j \in [n]. \tag{A.2}$$

We next upper bound the right hand side (RHS) of (A.2). Since the algorithm picks a violated covering constraint to whack in each round, we have: $\left(p^t\right)^\top \cdot x^t = (1/\lambda) \cdot \left(C^t x^t\right)_{i_t} \leq (1/\lambda)$. Taking the average of this inequality across all the T rounds, we get: $(1/T) \cdot \sum_{t=1}^T \left(p^t\right)^\top \cdot x^t \leq (1/\lambda)$. Since $T = \lambda \ln(n)/\epsilon^2$, we derive the following upper bound on the RHS of (A.2).

$$\frac{1}{T} \cdot \sum_{t=1}^T \left(p^t\right)^\top \cdot x^t + \frac{\ln(n)}{\epsilon \cdot T} \leq \frac{1}{\lambda} \cdot (1 + \epsilon). \tag{A.3}$$

We now focus our attention on the left hand side (LHS) of (A.2). Fix any expert $j \in [n]$. We first express the payoff obtained by this expert at round $t \in [T]$ in terms of the vector y^t, and get: $\left(p^t\right)_j = \frac{1}{\lambda} \cdot C_{i_t,j} = $

$\frac{1}{\lambda} \cdot \left((C^t)^\top y^t\right)_j$. Since $y = \frac{1}{T} \cdot \sum_{t=1}^{T} y^t$, the average payoff for the expert j across all the T rounds is given by:

$$\frac{1}{T} \cdot \sum_{t=1}^{T} \left(p^t\right)_j = \frac{1}{T} \cdot \frac{1}{\lambda} \cdot \sum_{t=1}^{T} \left((C^t)^\top y^t\right)_j \geq \frac{1}{T} \cdot \frac{1}{\lambda} \cdot \sum_{t=1}^{T} \left((C^T)^\top y^t\right)_j = \frac{1}{\lambda} \cdot \left((C^T)^\top y\right)_j. \tag{A.4}$$

In A.4 the inequality holds as the elements of C may only reduce over time. The inequality is the most significant deviation from the proof in the static setting. From (A.2), (A.3) and (A.4), we get:

$$\frac{(1 - \epsilon)}{\lambda} \cdot \left((C^T)^\top y\right)_j \leq \frac{1}{\lambda} \cdot (1 + \epsilon), \text{ and hence } \left((C^T)^\top y\right)_j \leq (1 + 4\epsilon) \text{ for all } j \in [n].$$

The last inequality holds since $\epsilon < 1/2$. This concludes the proof of Theorem 2.12.

B   Missing Proofs from Section 3.1

Claim B.1. *A coordinate k is cheap if and only if $\langle \nabla f_p(x), e_k \rangle \leq (1 + \epsilon)\langle \nabla f_c(x), e_k \rangle$.*

Proof. Follows from definitions: $\langle \nabla f_p(x), e_k \rangle = \frac{\sum_{i \in [m_p]} e^{\eta \cdot P_i x} \cdot P(i,k)}{\sum_{i \in [m_p]} e^{\eta \cdot P_i x}}$, $\langle \nabla f_c(x), e_k \rangle = \frac{\sum_{i \in [m_c]} e^{\eta \cdot C_i x} \cdot C(i,k)}{\sum_{i \in [m_c]} e^{\eta \cdot C_i x}}$,

hence if $\frac{\langle \nabla f_p(x), e_k \rangle}{\langle \nabla f_c(x), e_k \rangle} \leq 1 + \epsilon$ we have that $\frac{\sum_{i \in [m_p]} e^{\eta \cdot P_i x} \cdot P(i,k)}{\sum_{i \in [m_c]} e^{\eta \cdot C_i x} \cdot C(i,k)} \cdot \frac{\sum_{i \in [m_c]} e^{\eta \cdot C_i x}}{\sum_{i \in [m_p]} e^{\eta \cdot P_i x}} = \lambda(x,k) \cdot \frac{w_c(x)}{w_p(x)} \leq 1 + \epsilon$. $\qquad \square$

B.1   Proof of Property 3.3

Proof. The proof is analogous to the argument of Young [You14]. Let y be a feasible solution to the linear program. Then we have that $\frac{\sum_{i=1}^{m_p} (Cy)_i \cdot w_c(x,i)}{w_c(x)} \geq 1 \geq \frac{\sum_{i=1}^{m_p} (Py)_i \cdot w_p(x,i)}{w_p(x)}$ as $w_c(x,i)/w_c(x)$ and $w_p(x,i)/w_p(x)$ just are describing distributions over the the constraints. This could be phrased as:

$$y \cdot \left(C^T \cdot \frac{w_c(x)}{|w_c(x)|} - P^T \cdot \frac{w_p(x)}{|w_p(x)|}\right)$$

Which as y is non-negative implies that there is an i such that $C_i^T \cdot \frac{w_c(x)}{|w_c(x)|} - P_i^T \cdot \frac{w_p(x)}{|w_p(x)|} \geq 0$ which is equivalent to saying $\lambda(x,i) \leq \frac{w_p(x)}{w_c(x)}$ meaning there is a cheap coordinate. $\qquad \square$

B.2   Proof of Property 3.4

For the convenience of the reader we will restate the statement precisely: given ϵ is sufficiently small such that $\epsilon \leq 1/10$ we have that:

$$f_p(x + \delta \cdot e_k) - f_p(x) \leq (1 + 10 \cdot \epsilon) \cdot (f_c(x + \delta \cdot e_k) - f_c(x)) \leq (1 + 10 \cdot \epsilon) \cdot \epsilon \leq 2 \cdot \epsilon$$

Proof. Note, that if $\epsilon \leq 1/10$ then $exp(-\log(m_c + U/L)/\epsilon) \leq exp(-2 \cdot \ln(m_c + U/L) - 1/\epsilon + 2) \leq \frac{\epsilon \cdot L}{U \cdot m_c}$.

Let k be a cheap coordinate which the algorithms boosts by δ and let x stand for the value of the output vector before boosting. If coordinate k is boosted we have that $\langle \nabla f_p(x), e_k \rangle \leq (1 + \epsilon) \cdot \langle \nabla f_c(x), e_k \rangle$. We will show that $\langle \nabla f_p(x + \delta \cdot e_k), e_k \rangle \leq \langle \nabla f_p(x), e_k \rangle \cdot (1 + O(\epsilon))$ and $\langle \nabla f_c(x + \delta \cdot e_k), e_k \rangle \geq \langle \nabla f_p(x), e_k \rangle \cdot (1 - O(\epsilon))$. As $f_c(x)$ and $f_p(x)$ monotonously reduces and increases respectively with x this implies that

during the boost it must hold that $\langle \nabla f_p(x), e_k \rangle \leq (1 + O(\epsilon)) \cdot \langle \nabla f_c(x), e_k \rangle$ which in turn implies that $f_p(x + \delta \cdot e_k) - f_p(x) \leq (f_c(x + \delta \cdot e_k) - f_c(x)) \cdot (1 - O(\epsilon))$ implying one half of the claim.

$$
\begin{aligned}
\langle \nabla f_p(x + \delta \cdot e_k), e_k \rangle &= \frac{\sum_{i \in [m_p]} e^{\eta \cdot P_i(x + \delta \cdot e_k)} \cdot P(i, k)}{\sum_{i \in [m_p]} e^{\eta \cdot P_i(x + \delta \cdot e_k)}} \\
&\leq \frac{\sum_{i \in [m_p]} e^{\epsilon + \eta \cdot P_i x} \cdot P(i, k)}{\sum_{i \in [m_p]} e^{\eta \cdot P_i x}} \qquad (\text{B.1}) \\
&= e^{\epsilon} \cdot \langle \nabla f_p(x), e_k \rangle \\
&\leq (1 + 2 \cdot \epsilon) \cdot \langle \nabla f_p(x), e_k \rangle \qquad (\text{B.2})
\end{aligned}
$$

Inequality B.1 follows from the definition of δ and Inequality B.2 applies for any $\epsilon \leq 1$.

Define $I = \{i \in [m_c] : C_i x > 2$ and $A = \{i \in [m_c] : C_i x \leq 2\}$ representing the inactive and active constraints of C.

$$
\begin{aligned}
\langle \nabla f_c(x), e_k \rangle &= \frac{\sum_{i \in [m_c]} e^{-\eta \cdot C_i x} \cdot C(i, k)}{\sum_{i \in [m_c]} e^{-\eta \cdot C_i x}} \\
&= \frac{\sum_{i \in A} e^{-\eta \cdot C_i x} \cdot C(i, k) + \sum_{i \in I} e^{-\eta \cdot C_i x} \cdot C(i, k)}{\sum_{i \in A} e^{-\eta \cdot C_i x} + \sum_{i \in I} e^{-\eta \cdot C_i x}} \\
&\leq \frac{\sum_{i \in A} e^{-\eta \cdot C_i x} \cdot C(i, k) + \sum_{i \in I} e^{-\eta \cdot C_i x} \cdot C(i, k)}{\sum_{i \in A} e^{-\eta \cdot C_i x}} \\
&\leq \frac{\sum_{i \in A} e^{-\eta \cdot C_i x} \cdot C(i, k) + \sum_{i \in I} e^{-2 \cdot \eta} \cdot C(i, k)}{\sum_{i \in A} e^{-\eta \cdot C_i x}} \\
&\leq \frac{\sum_{i \in A} e^{-\eta \cdot C_i x} \cdot C(i, k) + e^{-\eta} \cdot m_c \cdot e^{-\frac{\log(m_c + U/L)}{\epsilon}} \cdot \max_{i \in [m_c]}\{C(i, k)\}}{\sum_{i \in A} e^{-\eta \cdot C_i x}} \\
&\leq \frac{\sum_{i \in A} e^{-\eta \cdot C_i x} \cdot C(i, k) + e^{-\eta} \cdot L \cdot \epsilon}{\sum_{i \in A} e^{-\eta \cdot C_i x}} \qquad (\text{B.3}) \\
&\leq \frac{\sum_{i \in A} e^{-\eta \cdot C_i x} \cdot C(i, k)}{\sum_{i \in A} e^{-\eta \cdot C_i x}} \cdot (1 + \epsilon) \qquad (\text{B.4})
\end{aligned}
$$

Equation B.3 follows if ϵ is selected small enough such that $e^{-\frac{\log(m_c + U/L)}{\epsilon}} \leq \frac{\epsilon \cdot L}{U \cdot m_c}$. Equation B.4 follows from the fact that in order for a boost to occur there must be at least one unsatisfied covering constraint, hence $\sum_{i \in A} e^{-\eta \cdot C_i x} \cdot C(i, k) \geq e^{-\eta} \cdot L$.

$$\langle \nabla f_c(x + \delta \cdot e_i), e_k \rangle \;=\; \frac{\sum_{i \in [m_c]} e^{-\eta \cdot C_i(x + \delta \cdot e_i)} \cdot C(i,k)}{\sum_{i \in [m_c]} e^{-\eta \cdot C_i(x + \delta \cdot e_i)}}$$

$$=\; \frac{\sum_{i \in A} e^{-\eta \cdot C_i(x + \delta \cdot e_i)} \cdot C(i,k) + \sum_{i \in I} e^{-\eta \cdot C_i(x + \delta \cdot e_i)} \cdot C(i,k)}{\sum_{i \in A} e^{-\eta \cdot C_i(x + \delta \cdot e_i)} + \sum_{i \in I} e^{-\eta \cdot C_i(x + \delta \cdot e_i)}}$$

$$\geq\; \frac{\sum_{i \in A} e^{-\eta \cdot C_i(x + \delta \cdot e_i)} \cdot C(i,k)}{\sum_{i \in A} e^{-\eta \cdot C_i(x + \delta \cdot e_i)} + \sum_{i \in P} e^{-\eta \cdot C_i(x + \delta \cdot e_i)}}$$

$$\geq\; \frac{\sum_{i \in A} e^{-\eta \cdot C_i(x + \delta \cdot e_i)} \cdot C(i,k)}{\sum_{i \in A} e^{-\eta \cdot C_i(x + \delta \cdot e_i)} + m_c \cdot e^{-2 \cdot \eta}}$$

$$\geq\; \frac{\sum_{i \in A} e^{-\eta \cdot C_i(x + \delta \cdot e_i)} \cdot C(i,k)}{\sum_{i \in A} e^{-\eta \cdot C_i(x + \delta \cdot e_i)}} \cdot \frac{1}{1 + \epsilon} \qquad\qquad\qquad \text{(B.5)}$$

$$\geq\; \frac{\sum_{i \in A} e^{-\eta \cdot C_i x} \cdot C(i,k)}{\sum_{i \in A} e^{-\eta \cdot C_i x}} \cdot \frac{e^{-\epsilon}}{1 + \epsilon} \qquad\qquad\qquad \text{(B.6)}$$

Here Equation B.5 similarly follows from the fact that $\sum_{i \in A} e^{-\eta \cdot C_i(x + \delta \cdot e_i)} \geq e^{-\eta - \epsilon}$ (by the choice of δ) and by the definition of η. Equation B.6 follows from the definition of δ. Therefore, we get that $\langle \nabla f_c(x + \delta \cdot e_i), e_k \rangle \cdot (1 + 4 \cdot \epsilon) \geq \langle \nabla f_c(x + \delta \cdot e_i), e_k \rangle \cdot \frac{(1 + \epsilon)^2}{e^{-\epsilon}} \geq \langle \nabla f_c(x), e_k \rangle$ (given $\epsilon < 1/5$).

As $\langle \nabla f_p(x + \delta \cdot e_i), e_k \rangle \leq (1 + 2 \cdot \epsilon) \langle \nabla f_p(x), e_k \rangle$ and $\langle \nabla f_p(x), e_k \rangle \leq \langle \nabla f_x(x), e_k \rangle \cdot (1 + \epsilon)$ we have that $\langle \nabla f_p(x + \delta \cdot e_i), e_k \rangle \leq \langle \nabla f_c(x + \delta \cdot e_i), e_k \rangle \cdot (1 + \epsilon) \cdot (1 + 2 \cdot \epsilon) \cdot (1 + 4\epsilon) \leq \langle \nabla f_c(x + \delta \cdot e_i), e_k \rangle \cdot (1 + 10 \cdot \epsilon)$ (given $\epsilon \leq 1/10$).

As both $f_c(x)$ and $f_p(x)$ are monotone in x we get that $f_p(x + \delta \cdot e_k) - f_p(x) \leq (f_c(x + \delta \cdot e_k) - f_c(x)) \cdot (1 + 10 \cdot \epsilon)$. Note that $f_c(x) \geq \min_{i \in [m_c]} C_i x$ and $f_c(x + \delta \cdot e_k) + \epsilon \leq \min_{i \in [m_c]} C_i(x + e_k \cdot \epsilon) \leq \min_{i \in [m_c]} C_i x + \epsilon$ therefore $f_c(x + \delta \cdot e_k) - f_c(x) \leq \epsilon$. Therefore, we can conclude the original inequality. $\qquad\square$

C   Complete Algorithm from Section 3.2

In Section 3, we presented an overview of our dynamic algorithm for positive LPs under relaxing updates. But we glossed over some implementation details to highlight the key ideas. For instance, we did not specify the dynamic data structure that will be needed to find the appropriate δ during a call to the $\textsc{Boost}(x, k)$ subroutine (see step 1 in Figure 9). Furthermore, in Section 3.2.2 we focused only on relaxing updates to the covering constraints, and we sketched in Section 3.2.3 how to deal with the relaxing updates to the packing constraints. Accordingly, to address these gaps, in this section we present a full, comprehensive version of our dynamic algorithm for positive LPs under relaxing updates.

Preliminaries: Let $P \in \Re_{\geq 0}^{m_p \times n}$ and $C \in \Re_{\geq 0}^{m_c \times n}$ be input matrices to the linear program. Let N stand for the number of entries in P and C which are non-zero at any point during the run of the algorithm. Let U and L stand for upper and lower bounds on the elements of the constraint matrices P, C at all times, respectively. Let t stands for the total number of relaxing updates progressed by the dynamic algorithm.

Let $R_i^P : i \in [m_p]$, $R_j^C : j \in [m_c]$ refer to the number of elements in the i-th row of P and the j-th row of C respectively which take a non-zero value at any time. Similarly, define $N_k^P : k \in [n]$ and $N_k^C : k \in [n]$ to represent the number of elements in the k-th column of P and C respectively taking a non-zero value at any point.

Theorem C.1. *There is an algorithm for maintaining a certificate of infeasability or returning an ϵ-approximate solution for a relaxing dynamic positive linear program in $O(\frac{N \cdot \log^2(m_p + m_c + U/L)}{\epsilon^2} + t)$ deterministic time (for $\epsilon < 1/200$).*

We will dedicate this entire section to proving Theorem C.1. For sake of simplicity we will first show how we can handle a dynamic linear program undergoing solely entry wise updates. In Subsection C.6 we will show to extend the algorithm for translation updates. We will make the following assumption on the distribution of inputs:

Assumption C.2. *Non-zero elements of input matrices P and C lie in the range of $[1/poly(N), poly(N)]$ at all times.*

C.1   Variables

Define P^*, C^* and X^* as described in Section 3.2.3:

- $P^* \in \mathbb{R}_{\geq 0}^{m_p \times (n+1)}$, where $P^*(i, k) = P(i, k)$ and $P^*(i, n+1) \geq 0$ for all $i \in [m_p], k \in [n]$.

- $C^* \in \mathbb{R}_{\geq 0}^{m_c \times (n+1)}$, where $C^*(j, k) = C(j, k)$ and $C^*(j, n+1) = 0$ for all $j \in [m_c], k \in [n]$.

- $x^* \in \mathbb{R}_{\geq 0}^{n+1}$ and x refers to the first n coordinates of x^*

From now on we will refer to the linear program defined by $Px \leq \mathbb{1}$, $CX \geq \mathbb{1}$ as the original linear program and to the linear program defined by $P^*x^* \leq \mathbb{1}$, $C^*x^* \geq \mathbb{1}$ as the extended linear program. Use variable i and j to refer to row indices of P and C respectively, and use k for referring to column indices of P or C.

Observation C.3. *If x is a feasible solution to the original linear program then $\{x, 1\}$ (x extended with a 1) is a valid solution to the extended linear program. If x^* is a feasible solution to the extended linear program then the first n coordinates of x^* form a feasible solution to the original linear program. Hence, the original linear program is feasible if and only if the extended linear program is feasible.*

Due to Observation C.3 we will focus only on the extended linear program in this section.

Data-Structure: We will initially define the following variables:

$$\eta = \frac{\log(m_c + m_p + U/L)}{\epsilon} \tag{C.1}$$

- $w_c(x^*, i) = 1$ for $i \in [m_c]$ representing covering constraint weights

- $w_p(x^*, j) = 1$ for $j \in [m_p]$ representing packing constraint weights

- $w_c(x^*) = \sum_{i \in [m_c]} w_c(x^*, i)$ and $w_p(x^*) = \sum_{j \in [m_p]} w_p(x^*, j)$

- $\lambda(x^*)_0 = \frac{w_p(x^*)}{w_c(x^*)}$

For the purposes of the analysis we will define $\lambda(x^*, k) = \forall k \in [n] : \lambda(x^*, k) = \frac{\sum_{j \in [m_p]} w_p(x^*, j) \cdot P^*(j, k)}{\sum_{i \in [m_c]} w_c(x^*, i) \cdot C^*(i, k)}$ representing the price of each coordinate. Note that after initialization the algorithm will not keep track of exact $\lambda(x^*, k)$ values. Similarly, for the purposes of the algorithm description and analysis define but do not maintain

$$\forall k \in [n] : \delta_k = \max_\delta\{\max\{\max_{i\in[m_p]} P^*(i,k)\cdot\delta, \max_{j\in[m_c]|C_j^*x^*<2} C^*(j,k)\cdot\delta\} = \epsilon/\eta\}$$

Approximate Variables: The algorithm will initialize the following approximate variables:

- $\forall i \in [m_p] : \hat{w}_p(x^*,i) = w_p(x^*,i)$

- $\forall j \in [m_c] : \hat{w}_c(x^*,j) = w_c(x^*,j)$

- $\hat{w}_p(x^*) = w_p(x^*)$

- $\hat{w}_c(x^*) = w_c(x^*)$

- $\forall k \in [n] : \hat{\lambda}(x^*,k) = \lambda(x^*,k)$

- $\hat{\lambda}_0(x^*) = \lambda_0(x^*)$

Invariant C.4. *The following inequalities are simultaneously satisfied:*

- $\forall i \in [m_c] : w_c(x^*,i) \leq \hat{w}_c(x^*,i) \leq w_c(x^*,i)\cdot(1+\epsilon)$

- $\forall i \in [m_p] : w_p(x^*,i) \geq \hat{w}_p(x^*,i) \geq w_p(x^*,i)\cdot(1-\epsilon)$

- $\forall k \in [n] : \hat{\lambda}(x^*,k) = \frac{\sum_{j\in[m_p]} \hat{w}_p(x^*,j)\cdot P^*(j,k)}{\sum_{i\in[m_c]} \hat{w}_c(x^*,i)\cdot C^*(i,k)}$ *and* $\hat{\lambda}_0(x^*) \geq \lambda_0(x^*)\cdot(1-\epsilon)$

C.2 Algorithm

***Iterate* subroutine:** Initially the algorithm calls the *Iterate()* sub-routine which initializes $\hat{\lambda}_0(x^*) = \lambda_0(x^*)$ and iterates over all coordinates $k \in [n]$. *Iterate()* boosts any coordinate k satisfying $\hat{\lambda}(x^*,k) \leq \lambda_0(x^*)\cdot(1+5\cdot\epsilon)$ until $\hat{\lambda}(x^*,k) > \lambda_0(x^*)\cdot(1+5\cdot\epsilon)$. If due to the boosts $\lambda_0(x^*)$ has increased enough such that $\lambda_0(x^*)\cdot(1-\epsilon) > \hat{\lambda}_0(x^*)$ *Iterate()* is called recursively.

***Boost* subroutine:** A call to *Boost(k)* increases the k-th coordinate of x^*. The algorithm selects an increment of δ such that all packing constraints and all not 'well' satisfied covering constraints are increased by at most ϵ/η, that is $\delta = \max_\delta\{\max\{\max_i P*(i,k)\cdot\delta, \max_{j|C_j^*<2} C^*(j,k)\cdot\delta\} = \epsilon/\eta\}$. The algorithm increments x^* by $e_k\cdot\delta$ and afterwards proceeds to restore the affected variables $w_c(x^*,i), w_p(x^*,j), w_c(x^*), w_p(x^*)$ and $\lambda_0(x^*)$ precisely. Afterwards *UpdatePWeights, UpdateCWeights* subroutines are called to ensure the correctness of Invariant C.4.

***UpdatePweights, UpdateCWeights* subroutines:** These subroutines are used to update $\hat{w}_p, \hat{w}_c$ and $\hat{\lambda}$ values respectively. Without loss of generality assume that *UpdateCWeights(k)* was called. The algorithm checks for all $i \in [m_P]$ such that the i-th element of the k-th column of P^* is not empty ($P*(i,k) > 0$) weather $\hat{w}_p(x^*,i) < w_p(x^*,i)\cdot(1-\epsilon)$ as these are the $w_p(x^*,i)$ values which have increased due to the boosting coordinate k. For rows where the slack between the approximate and actual values became too significant the algorithm readjusts the approximate parameter ($\hat{w}_p(x^*,i) \leftarrow w_p(x^*,i)$) and adjusts all values of $\hat{\lambda}(x^*,k')$ which might have been affected by this change (that is the ones corresponding to columns of P^* where the i-th row is non empty, that is $P^*(i,k') > 0$).

***UpdateP* subroutine:** A call *UpdateP(i,k,Δ)* handles a reduction of Δ to $P^*(i,k)$. After updating $P*(i,k)$ the algorithm increments $P*(i,n+1)$ with $\Delta \cdot x_i^*$ to guarantee that $w_p(x^*,i)$ remains the same after increasing $P^*(i,k)$. This way the only $\hat{\lambda}$ value which is affected is $\hat{\lambda}(x^*,k)$ which the subroutine updates

to ensure the correctness of Invariant C.4. Due to the update of $\hat{\lambda}(x^*, k)$ coordinate k might become cheap (that is $\hat{\lambda}(x^*, k) \leq \lambda_0(x^*) \cdot (1 + 5 \cdot \epsilon)$) so the subroutine boosts k until it is no longer cheap. If some boosts occur then $\lambda_0(x^*)$ might reduce enough such that $\hat{\lambda}_0(x^*) < \lambda_0(x^*) \cdot (1 - \epsilon)$ in which case a new call to *Iterate* is made to ensure the correctness of Invariant C.4.

Algorithm 1: Main subroutine of the algorithm

Input: $C^* \in \Re_{\geq 0}^{m_c \times n}, P^* \in \Re_{\geq 0}^{m_p \times n}$
Output: $x^* \in \Re_{\geq 0}^{n+1} | C^* x^* \geq \mathbb{1}, P^* x^* \leq \mathbb{1} \cdot (1 + 200 \cdot \epsilon)$

1 Function Iterate():
2 $\hat{\lambda}_0(x^*) \leftarrow \lambda_0(x^*)$
3   **for** $k \in [n]$ **do**
4   **while** $\hat{\lambda}(x^*, k) \leq \lambda_0(x^*) \cdot (1 + 5 \cdot \epsilon)$ **do**
5   Boost(k)   `// See Algorithm 2`
6   **if** $\hat{\lambda}_0(x^*) < \lambda_0(x^*) \cdot (1 - \epsilon)$ **then**
7   Iterate()

8 Function Main():
9   Iterate()

Algorithm 2: Boosting of coordinate k

---

1 Function Boost(k):
2 $\delta \leftarrow$ sample in $[\delta_k/4, \delta_k]$ `// See Section C.5`
3 $x^* \leftarrow x^* + e_k \cdot \delta$ $\not{e_k}$ is the unit k-th coordinate vector
4   **if** $C^* x^* \geq \mathbb{1}$ **then**
5   Return x^*
6   **for** $\forall i \in [m_p] | P^*(i, k) > 0$ **do**
7 $w_p(x^*, i) \leftarrow \exp(\eta \cdot P_i^* x^*)$
8 $w_p(x^*) \leftarrow \sum_{i \in [m_p]} w_p(x^*, i)$ `// Updated in` $O(N_k^P)$ `time`
9   **for** $\forall j \in [m_c] | C^*(j, k) > 0$ **do**
10 $w_c(x^*, j) \leftarrow \exp(-\eta \cdot C_j^* x^*)$
11 $w_c(x^*) \leftarrow \sum_{j \in [m_c]} w_c(x^*, j)$ `// Updated in` $O(N_k^C)$ `time`
12 $\lambda_0(x^*) = \frac{w_p(x^*)}{w_c(x^*)}$
13   UpdatePWeights(k)   `// See Algorithm 3`
14   UpdateCWeights(k)   `// See Algorithm 4`

C.3 Correctness

Claim C.1. *Invariant C.4 is always satisfied by Algorithm 1.*

Proof. At initialization the invariant clearly holds by the definition of the approximate variables as their exact counterparts. To prove the correctness of the invariant we will consider the three instances where variables are updated and argue inductively that they preserve Invariant C.4.

Algorithm 3: The subroutine updating packing weights

1 Function `UpdatePWeights(k):`

2   **for** $j \in [m_p] | P^*_{j,k} > 0$ **do**

3   **if** $\hat{w}_p(x^*, j) < w_p(x^*, j) \cdot (1 - \epsilon)$ **then**

4 $\hat{w}_p(x^*, j) = w_p(x^*, j)$

5   **for** $k' \in [n] | P^*_{j,k'} > 0$ **do**

6 $\hat{\lambda}(x^*, k') = \dfrac{\sum_{j' \in [m_p]} \hat{w}_p(x^*, j') \cdot P^*(j', k')}{\sum_{i \in [m_c]} \hat{w}_c(x^*, i) \cdot C^*(i, k')}$

Algorithm 4: The subroutine updating covering weights

1 Function `UpdateCWeights(i):`

2   **for** $i \in [m_c] | C^*_{i,k} > 0$ **do**

3   **if** $\hat{w}_c(x^*, i) > w_c(x^*, i) \cdot (1 + \epsilon)$ **then**

4 $\hat{w}_c(x^*, i) = w_c(x^*, i)$

5   **for** $k' \in [n] | C^*_{i,k'} > 0$ **do**

6 $\hat{\lambda}(x^*, k') = \dfrac{\sum_{j \in [m_p]} \hat{w}_p(x^*, j) \cdot P^*(j, k')}{\sum_{i' \in [m_c]} \hat{w}_c(x^*, i') \cdot C^*(i', k')}$

Algorithm 5: Subroutine handling updates to the packing constraint matrix P^*

    Input: (j, k, Δ): $P^*(j, k)$ decreased by Δ

1 Function `UpdateP(j, k, Δ):`

2 $P^*(i, k) \leftarrow P^*(i, k) - \Delta$

3 $P^*(i, n + 1) = P^*(i, n + 1) + \Delta \cdot x^*_i$

4 $\hat{\lambda}(x^*, k) = \dfrac{\sum_{i' \in [m_p]} \hat{w}_p(x^*, i') \cdot P^*(i', k)}{\sum_{j \in [m_c]} \hat{w}_c(x^*, j) \cdot C^*(j, k)}$ // Updated in $O(1)$ time

5   **while** $\hat{\lambda}(x^*, k) \leq \lambda_0(x^*) \cdot (1 + 5 \cdot \epsilon)$ **do**

6   Boost(k)   // See Algorithm 2

7   **if** $\hat{\lambda}_0(x^*) < \lambda_0(x^*) \cdot (1 - \epsilon)$ **then**

8   Iterate()   // See Algorithm 1

Algorithm 6: Subroutine handling updates to the covering constraint matrix C^*

Input: (j, k, Δ): $C^*(j, k)$ increased by Δ

1 Function UpdateC(j, k, Δ):

2 $C^*(j, k) \leftarrow C^*(j, k) + \Delta$

3   **if** $C^* x^* \geq \mathbb{1}$ **then**

4   Return x^*

5 $w_c(x^*, j) \leftarrow \exp(-\eta \cdot C_j^* x^*)$

6 $w_c(x^*) \leftarrow \sum_{j \in [m_c]} w_c(x^*, j)$ // Updated in $O(1)$ time

7 $\lambda_0(x^*) = \frac{w_p(x^*)}{w_c(x^*)}$

8   **if** $w_c(x^*, j) < \hat{w}_c(x^*, j) \cdot (1 - \epsilon)$ **then**

9 $\hat{w}_c(x^*, j) = w_c(x^*, j)$

10   **for** $k' \in [n] | C_{j,k'}^* > 0$ **do**

11 $\hat{\lambda}(x^*, k') = \frac{\sum_{i \in [m_p]} \hat{w}_p(x^*, i) \cdot P^*(i, k')}{\sum_{j' \in [m_c]} \hat{w}_c(x^*, j') \cdot C^*(j', k')}$

12   **for** $k' \in [n] | C_{j,k'}^* > 0$ **do**

13   **while** $\hat{\lambda}(x^*, k') \leq \lambda_0(x^*) \cdot (1 + 5 \cdot \epsilon)$ **do**

14   Boost(k')   // See Algorithm 2

15 $\hat{\lambda}(x^*, k) = \frac{\sum_{i \in [m_p]} \hat{w}_p(x^*, i) \cdot P^*(i, k)}{\sum_{j' \in [m_c]} \hat{w}_c(x^*, j') \cdot C^*(j', k)}$

16   **while** $\hat{\lambda}(x^*, k) \leq \lambda_0(x^*) \cdot (1 + 5 \cdot \epsilon)$ **do**

17   Boost(k)   // See Algorithm 2

18   **if** $\hat{\lambda}_0(x^*) < \lambda_0(x^*) \cdot (1 - \epsilon)$ **then**

19   Iterate()   // See Algorithm 1

Boost subroutine (Algorithm 2): when *Boost(k)* is called then elements of $w_p(x^*, i)|P^*(i, k) > 0$ and $w_p(x^*, j)|C^*(j, k) > 0$, $w_c(x^*)$, $w_p(x^*)$ and $\lambda_0(x^*)$ are updated. Afterwards *UpdatePWeights(k)* and *UpdateCWeights(k)* are called. These subroutines check variables $\hat{w}_p(x^*, i)|P^*(i, k) > 0$ and $\hat{w}_p(x^*, j)|C^*(j, k) > 0$ and update them to enforce the requirements of Invariant C.4. *Boost(k)* might be called by subroutines *Iterate, UpdateP* and *UpdateC*. Each of these subroutines call *Iterate* if the last requirement of Invariant C.4 ($\hat{\lambda}_0(x^*) < \lambda_0 \cdot (1 - \epsilon)$) is violated. As Invariant C.4 can't compile (as it calls itself) until $\hat{\lambda}_0(x^*) \geq \lambda_0 \cdot (1 - \epsilon)$ all of Invariant C.4 will be satisfied after a boost operation.

UpdateP subroutine (Algorithm 3): *UpdateP(i, k, Δ)* doesn't affect constraint weights by define. The only variable it needs to update is $\hat{\lambda}(x^*, k)$ because of the change in $P^*(i, k)$. As argued previously, Invariant C.4 is restored by the *Boost* calls inside of *UpdateP*.

UpdateC subroutine (Algorithm 4): *UpdateC* makes changes to $w_c(x^*, j), w_c(x^*)$ and $\lambda_0(x^*)$ and updates the approximate variables accordingly to fit Invariant C.4. As argued previously the invariant is restored after proceeding *Boost* calls.

□

Claim C.2. *Whenever the algorithm halts without returning an answer the extended LP is infeasible.*

Proof. By Claim C.1 we can assume that Invariant C.4 is satisfied whenever the algorithm halts without returning an answer. Following the proof of Property 3.3 from Section B.1 if at any time for all $k \in [n]$ we have that $\lambda(x^*, k) < \lambda_0(x^*)$ then the linear program is infeasible. We will argue that Invariant C.4 implies $\forall k \in [n] : \lambda(x^*, k) < \lambda_0(x^*)$. Fix $k \in [n]$ for this proof.

By Invariant C.4 we always have that $\hat{\lambda}(x^*, k) \geq \lambda(x^*, k)$. Say whenever the algorithm checked if $\hat{\lambda}(x^*, k) \leq \lambda_0(x^*) \cdot (1 + 5 \cdot \epsilon)$ that the algorithm has checked the price of coordinate k. After a coordinates price is checked it is boosted until $\lambda(x^*, k) \geq \hat{\lambda}(x^*, k) > \lambda_0(x^*) \cdot (1 + 5 \cdot \epsilon)$. Each coordinates price is checked at least once at initialization. Let $\lambda_0^-(x^*)$ and $\lambda_0^+(x^*)$ represent the value of $\lambda_0(x^*)$ after the last time coordinate k-s price was checked and at compilation respectively. Let $\hat{\lambda}_0(x^*)^*$ represent the value of $\hat{\lambda}_0(x^*)$ at both of these time points (note that if $\hat{\lambda}_0(x^*)$ would have been updated between the last price checking of coordinate k and compilation a new call to *Iterate()* would have been made, hence k-s price would have been checked again). By Invariant C.4 we have that $\hat{\lambda}_0(x^*)^* \leq \lambda_0^-(x^*)$ and $\hat{\lambda}_0(x^*)^* \geq \lambda_0^+(x^*) \cdot (1 - \epsilon)$. Therefore, $\lambda_0^+(x^*) \leq \frac{\lambda_0^+(x^*)}{1-\epsilon}$.

$\lambda(x^*, k)$-s value might only have reduced since the last time k-s price was checked as at every instance it's price might increase it is checked. Therefore, we have that at that whenever the algorithm halts $\lambda(x^*, k) \geq \lambda_0(x^*) \cdot (1 + 5\epsilon) \cdot (1 - \epsilon) \geq \lambda_0(x^*)$ if $\epsilon \leq 4/5$ which concludes the claim.

□

Claim C.3. *At all times the algorithm satisfies that $P^* x^* \leq \mathbb{1} \cdot (1 + 200 \cdot \epsilon)$ (given $\epsilon < 1/10$).*

Proof. We will show that the following invariant is maintained throughout the run of algorithm:

$$\max_{i \in [m_p]} \{P_i^* x^*\} \leq \min_{i \in [m_c]} \{C_i^* x^*\} \cdot (1 + 50\epsilon) + 100\epsilon$$

Initially the inequality holds and the inequality will be affected by 3 kinds of changes: updates to P^*, C^* and x^*. Notice, that whenever C^* is updated only the right hand size of the inequality may change and it may only increase. Similarly, a decrease of P^* may only decrease the left hand side. However, the algorithm does increase the last column of P^* whenever an decrease of P^* occurs. Note however, that the last coordinate of x^* is is always 0 as coordinate $n + 1$ is never boosted (due to the last column of $C*$ being 0 $\lambda(x^*, n + 1)$ is unbounded hence coordinate $n + 1$ is never cheap).

It remains to argue that during Boost calls the inequality is maintained. Assume coordinate i is boosted by δ and at the start of the boosting process the inequality holds. Define $f_p(x^*) = \eta \cdot \log(\sum_{i \in [m_p]} \exp(\eta P_i^* x^*))$ and $f_c(x^*) = -\eta \cdot \log(\sum_{i \in [m_c]} \exp(-\eta C_i^* x^*))$ analogously to Section **??**. Following the proof of Property 3.4 we get that $f_p(x^* + \delta \cdot e_i) - f_p(x^*) \leq (f_c(x^* + \delta \cdot e_i) - f_c(x^*)) \cdot (1 + 50 \cdot \epsilon)$ (note that a different constant 50ϵ appears as the amount of slack the algorithm uses is different). As both function are monotonous in x^* and relaxing updates may only increase $f_c(x^*)$ and reduce $f_p(x^*)$ we have that at all times $f_c(x^*) \cdot (1 + 50 \cdot \epsilon) \geq f_p(x^*)$.

By the properties of $f_c(x^*)$ and $f_p(x^*)$ (stated in Section 3.1 as Equation 3.2 we can conclude that $\max_{i \in [m_p]} \{P_i^* x^*\} \leq \min_{i \in [m_c]} \{C_i^* x^*\} \cdot (1 + 50 \cdot \epsilon) + \epsilon \cdot (1 + 50 \cdot \epsilon)) \leq \min_{i \in [m_c]} \{C_i^* x^*\} \cdot (1 + 50 \cdot \epsilon) + 100 \cdot \epsilon$ at all times.

Let δ stand for the value of the last boost. By definition $C^* \delta \leq \mathbb{1} \cdot \epsilon$ hence at all times $\min_{j \in [m_c]} \{C_j^* x^*\} \leq 1 + \epsilon$. Therefore at all times $\max_{i \in [m_p]} \{P_i^* x^*\} \leq (1 + 50\epsilon) \cdot (1 + \epsilon) + 100\epsilon \leq 1 + 200\epsilon$ that is $P^* x^* \leq \mathbb{1} \cdot (1 + 200\epsilon)$.

$\square$

Corollary C.5. *If $\epsilon \leq 1/200$ then at all times $w_p(x^*) \leq \exp(3 \cdot \eta)$ and $w_x(x^*) \geq \exp(2 \cdot \eta)$, hence $\lambda_0(x^*) \leq \exp(5 \cdot \eta)$.*

Proof. $w_p(x^*) \leq m_p \cdot \exp(\eta \cdot \max_{i \in [m_p]} \{P_i^* x^*\}) \leq \exp(\eta(2 + 200 \cdot \epsilon)) \leq \exp(\eta \cdot 3)$. As argued in the proof of Claim C.3 we have that $\min_{j \in [m_c]} \{C_j^* x^*\} \leq 1 + \epsilon$ at all times, hence $w_c(x^*) \geq \exp(-\eta \cdot \min_{j \in [m_c]} \{C_j^* x^*\}) \geq \exp(-2 \cdot \eta)$..

$\square$

Lemma C.6. *Whenever the algorithm halts without returning an answer the extended linear program is infeasible. If the algorithm may only return an $O(\epsilon)$ approximate solution to the extended linear program.*

Proof. The lemma follows from Claim C.2 and Claim C.1

$\square$

C.4   Running Time

Claim C.4. *For any $k \in [n]$ Boost(k) may be called at most $O(\frac{\log^2(m + U/L)}{\epsilon^2}) = O(\frac{\log(m + U/L) \cdot \eta}{\epsilon})$ number of times.*

Proof. Fix a coordinate $k \in [n]$ for the rest of the proof.

Consider any call to the $\textsc{Boost}(x^*)$ subroutine during the course of our dynamic algorithm, which increases coordinate k of the vector $x^* \in \mathbb{R}_{\geq 0}^n$ by some amount $\delta > 0$. From Algorithm 2, observe that just before this specific call to the subroutine, either there exists a packing constraint $i \in [m_p]$ with $P^*(i,k) \cdot \delta \leq \epsilon/\eta$, or there exists a *not-too-large* covering constraint $j \in [m_c]$[17] with $C(j,k) \cdot \delta \leq \epsilon/\eta$. In the former (resp. latter) case, we refer to the index i (resp. j) as the *pivot* and the current value of $P(i,k)$ (resp. $C(j,k)$) as the *pivot-value* corresponding to this specific call to the subroutine $\textsc{Boost}(k)$.[18]

Due to Assumption C.2 we can assume that $U = 2^\tau \cdot L$ for some integer $\tau = O(\log n)$. Partition the interval $(L, U]$ into τ segments $\mathcal{I}_0, \ldots, \mathcal{I}_{\tau-1}$, where $\mathcal{I}_\ell = (L \cdot 2^\ell, L \cdot 2^{\ell+1}]$ for all $\ell \in \{0, \ldots, \tau - 1\}$.

Claim C.5. *Consider any $\ell \in \{0, \ldots, \tau - 1\}$. Throughout the duration of our dynamic algorithm, at most $16\eta/\epsilon$ calls with pivot-values in $\mathcal{I}_\ell$ and pivots in $[m_c]$ are made to the $\textsc{Boost}(k)$ subroutine.*

---

[17]We say that a covering constraint $j \in [m_c]$ is *not-too-large* iff $C_j^* x^* < 2$.

[18]Note that the values of the entries in the constraint matrix change over time due to the sequence of updates. The *pivot-value* of a call to $\textsc{Boost}(x,k)$ refers to the value of the concerned entry in the constraint matrix *just before* the specific call to $\textsc{Boost}(k)$.

Proof. Let $x_k \in \mathbb{R}_{\geq 0}$ denote the kth coordinate of the vector $x \in \mathbb{R}_{\geq 0}^n$. For ease of exposition, we say that a call to BOOST(k) is *covering-critical* iff its pivot is in $[m_c]$ and its pivot-value is in $\mathcal{I}_\ell$. We wish to upper bound the total number of covering-critical calls made during the course of our dynamic algorithm.

Consider any specific covering-critical call to BOOST(k) with pivot $j \in [m_c]$ and pivot-value $\alpha \in \mathcal{I}_\ell = \left(L \cdot 2^{\ell-1}, L \cdot 2^\ell \right]$. From the definition of Algorithm 2, it follows that this call increases x_k by $\delta = \epsilon/(\eta \alpha \cdot 4) \geq \epsilon/(\eta L 2^{\ell+2})$.

Suppose that T covering-critical calls have been made to the BOOST(x, k) subroutine, where $T > 16 \cdot \eta/\epsilon$. Let us denote these calls in increasing order of time by: $\Gamma_1, \Gamma_2, \ldots, \Gamma_T$. Thus, for all $t \in [T]$, we let Γ_t denote the tth covering-critical call made during the course of our dynamic algorithm. From the discussion in the preceding paragraph, each of these critical calls Γ_t increases x_k by at least $\epsilon/(\eta L 2^{\ell+2})$.

Since $x_k = 0$ initially and it increases monotonically over time, just before the last critical call Γ_T we have $x_k \geq (T-1) \cdot \epsilon/(\eta L 2^{\ell+2}) \geq (16\eta/\epsilon) \cdot \epsilon/(\eta L 2^{\ell+2}) = 1/(L2^{\ell-2})$. Let $j_T \in [m_c]$ be the pivot of the call Γ_T. Since the pivot-value of Γ_T lies in $\mathcal{I}_\ell$, it follows that $C(j_T, k) > L \cdot 2^{\ell-1}$ just before the call Γ_T. Thus, just before the call Γ_T, we have $C_{j_T} \cdot x \geq C(j_T, k) \cdot x_k > L2^{\ell-1} \cdot 1/(L2^{\ell-2}) = 2$. But if $C_{j_T} \cdot x > 2$ just before the call Γ_T, then j_T cannot be the pivot of Γ_T. This leads to a contradiction. Hence, we must have $T \leq 16 \cdot \eta/\epsilon$, and this concludes the proof of the claim. $\qquad \square$

Claim C.6. *Consider any $\ell \in \{0, \ldots, \tau - 1\}$. Throughout the duration of our dynamic algorithm, at most $16 \cdot \eta/\epsilon$ calls with pivot-values in $\mathcal{I}_\ell$ and pivots $i \in [m_p]$ are made to the BOOST(k) subroutine.*

Proof. (Sketch) As in the proof of Claim C.5, let $x_k \in \mathbb{R}_{\geq 0}$ denote the kth coordinate of the vector $x \in \mathbb{R}_{\geq 0}^n$. Say that a call to BOOST(k) is *packing-critical* iff its pivot is in $[m_p]$ and its pivot-value is in $\mathcal{I}_\ell$.

Suppose that T covering-critical calls have been made to the BOOST(k) subroutine, where $T > 16 \cdot \eta/\epsilon$. Let us denote these calls in increasing order of time by: $\Gamma_1, \Gamma_2, \ldots, \Gamma_T$. Thus, for all $t \in [T]$, we let Γ_t denote the tth packing-critical call made during the course of our dynamic algorithm. Let $i_T \in [m_p]$ be the pivot of the last packing-critical call Γ_T. Following the same argument as in the proof of Claim C.5, we conclude that $P_{i_T} \cdot x > 2$ just before the call Γ_T. By Claim C.3 this leads to a contradiction (assuming $\epsilon < 1/200$). Thus, it must be the case that $T \leq 16\eta/\epsilon$, and this concludes the proof of the claim. $\qquad \square$

Any call to the subroutine BOOST(k) has pivot in $[m_p] \cup [m_c]$ and pivot-value in $\mathcal{I}_\ell$ for some $\ell \in \{0, \ldots, \tau - 1\}$.[19] Accordingly, from Claim 3.1 and Claim 3.2, it follows that the total number of calls made to the BOOST(k) during the course of our dynamic algorithm is at most $2 \cdot \tau \cdot (16\eta/\epsilon) = O(\frac{\log^2(m+U/L)}{\epsilon^2})$. $\qquad \square$

Claim C.7. *All calls to UpdatePWeights and UpdateCWeights be handled in $O(\frac{\log^2(m+U/L) \cdot N}{\epsilon^2})$ total time.*

Proof. As each call to *Boost* calls the weight update functions once by Claim C.4 they will be called a total of $O(\frac{\log(m+U/L) \cdot \eta}{\epsilon} \cdot n)$ times. Fix $k \in [n]$. *UpdatePWeights(k)* and *UpdateCWeights(k)* will be called $O(\frac{\log(m+U/L) \cdot \eta}{\epsilon})$ times. Hence, the outer loops of both function will take $O(N \cdot \frac{\log(m+U/L) \cdot \eta}{\epsilon})$ total time to handle.

Without loss of generality focus on *UpdatePWeights*. For any $i \in [m_p]$ between any two updates to $\hat{w}_p(x^*, i)$ $w_p(x^*, i)$ must increase by a factor of $1 + \Omega(\epsilon)$. Due to Corollary C.5 this may happen at most $O(\log^2(m + U/L)/\epsilon^2)$ times. After updating $\hat{w}_p(x^*, i)$ the algorithm will spend $O(R_i^P)$ time updating $\hat{\lambda}_0(x^*)(x^*, k)$ values. Therefore, over all $i \in [m_p]$ the total work to maintain $\hat{\lambda}_0(x^*)(x^*, k)$ values will

---

[19]This holds because the pivot-values lie in the range $(L, U]$, and this range has been partitioned into subintervals: $\mathcal{I}_0, \ldots, \mathcal{I}_{\tau-1}$.

be in $O(\frac{\log^2(m+U/L)}{\epsilon^2})$. The proof proceeds similarly for the work *UpdateCWeights* does on maintaining $\hat{\lambda}_0(x^*)(x^*, k)$ values. $\qquad\square$

Claim C.8. *The total time spent by the algorithm through calls to the Boost subroutine is in $O(\frac{\log^2(m+U/L)\cdot N}{\epsilon^2})$*

Proof. Each call to *Boost(k)* requires 3 non-constant time tasks. Firstly an estimate of δ_k has to be sampled. By Claim C.11 and Claim C.4 these samplings will take $O(N \cdot \frac{\log(m+U/L)}{\epsilon^2})$ total time.

Afterwards the algorithm updates all $w_p(x^*, i)$ for $i \in [m_p]|P^*(i, k) > 0$ and $w_c(x^*, j)$ for $j \in [m_c]|C^*(j, k) > 0$ in $O(N_k^P + N_k^C)$ time. Over all calls to boost this will take total $O(N \cdot \frac{\log^2(m+U/L)}{\epsilon^2})$ time by Claim C.11.

Finally, the algorithm makes calls to *UpdatePWeights(k)* and *UpdateCWeights(k)* (see Algorithms 4 and 3). This will take total time $O(N \cdot \frac{\log^2(m+U/L)\cdot N}{\epsilon^2})$ by Claim C.7. $\qquad\square$

Claim C.9. *The total time by the UpdateP and UpdateC subroutines (outside of calls to Iterate and Boost) is bounded by $O(\frac{\log^2(m+U/L)\cdot N}{\epsilon^2} + t)$.*

Proof. The only action *UpdateP* does which is not in $O(1)$ is the while loop. Due to Claim C.4 the while loops will test a coordinates price at most $O(n \cdot \frac{\log^2(m+U/L}{\epsilon^2} + t)$ time over the total run of the algorithm.

Similarly, the while loops of *UpdateC* may also only have $O(n \cdot \frac{\log^2(m+U/L}{\epsilon^2} + t)$ iterations due to Claim C.4. The only other non-constant time operation of *UpdateC* is the process of updating $\hat{w}_c(x^*, j)$ and affected $\hat{\lambda}(x^*, k')$ values. Whenever $\hat{w}_c(x^*, j)$ is updated R_j^C values of $\hat{\lambda}(x^*, k')$ will be updated. As an update to $\hat{w}_c(x^*, j)$ increases its value by a multiplicative factor of at least $1 + \epsilon$ due to Assumption C.2 there can be at most $O(\log(N)/\epsilon)$ updates of $\hat{w}_c(x^*, j)$. Therefore, the maintenance of the slack parameters will take at most $O(\frac{N \cdot \log(N)}{\epsilon} + t)$ total time for *UpdateC*. This finishes the proof. $\qquad\square$

Lemma C.7. *Algorithm 1 runs in $O(\frac{N \cdot \log^2(m+U/L)}{\epsilon^2} + t)$ deterministic time.*

Proof. The time spent by calls to *Boost* or updates is handled in $O(\frac{N \cdot \log^2(m+U/L)}{\epsilon^2} + t)$ total time by Claim C.8 and Claim C.9. Between each call to iterate $\lambda_0(x^*)$ increases by a factor of $1/(1 - \epsilon)$. As $\lambda_0(x^*) \le \exp(5 \cdot \eta)$ by Corollary C.5 there will be at most $O(\eta/\epsilon) = O(\log(m + U/L)/\epsilon^2)$ calls to *Iterate()* which will take $O(\frac{N \cdot \log^2(m+U/L)}{\epsilon^2})$ to complete (apart from *Boost* calls). This completes the lemma. $\qquad\square$

C.5 Maintenance of an estimate of δ_k for Algorithm 2

To imitate the increments of the static algorithm described by Figure 9 when *Boost(k)* is called we would like to find $\delta = \max_\delta\{\max\{\max_i P^*(i, k) \cdot \delta, \max_{j|C_j^* x^* < 2} C^*(j, k) \cdot \delta\} = \epsilon/\eta$. Unfortunately, under dynamic relaxing updates returning the exact δ_k is too slow. Therefore, we will implement an efficient oracle which returns $\delta \in [\delta_k \cdot (1 - \epsilon), \delta_k]$ when queried in $O(1)$ time.

Max-Heap: Using folklore data structures we will assume that a max-heap on at most $m = m_c + m_p$ objects is a black-box tool which can handle the following actions:

- In $O(1)$ the max-heap returns the maximum-element stored in the heap

- In $O(\log(m))$ time an element can be removed from the heap

- In $O(\log(m))$ time an element can be added to the heap

Initialization: At initialization the oracle implements n max heaps $H_k : k \in [n]$, one for each coordinate of x^*. We will now describe the implementation of a specific heap H_k. H_k will consist of at most $N_k^C + N_k^P$ objects each representing a non-zero element of the k-th column of C^* or P^*. Each non-zero element of these two columns will be inserted into the heap with it's initial value and an identifier. Hence, the top element of H_k will represent the largest value in the k-th column of C^* and P^*.

Update to P^*: Assume that due to a relaxing update $P^*(i, k)$ reduced by Δ to some $P_+^*(i, k)$. The oracle will look-up the value $P^*(i, k)$ currently represented under in H_k, say this value is $P_-^*(i, k)$. If $P_-^*(i, k) > 2 \cdot P_+^*(i, k)$ then the oracle will remove $P_-^*(i, k)$ from H_k and re-enter it as $P_+^*(i, k)$. If $P_-^*(i, k) \le 2 \cdot P_+^*(i, k)$ then the oracle doesn't change anything in the heap H_k.

Update to C^*: Assume that due to a relaxing update $C^*(j, k)$ increased by Δ to some $C_+^*(j, k)$. If previously to the update $(C^*)_i x^* < 2$ however after the update $(C_*^+)_i x^* \ge 2$ the oracle proceeds over all non-zero entries of the j-th row of C^* (iterate over all $k' \in [m_c] | C^*(j, k') > 0$) of the form $C^*(j, k') > 0$ and removes them from their corresponding heaps. Otherwise the oracle proceeds similarly to its behaviour under a packing update: say that the value of $C^*(j, k)$ stored in H_k is $C_-^*(j, k)$. If $C_-^*(j, k) < 2 \cdot C_+^*(j, k)$ then the oracle removes the $C_-^*(j, k)$ node from the heap and re-enters it as the current value $C_+^*(j, k)$. If $C_-^*(j, k) \ge 2 \cdot C_+^*(j, k)$ the oracle rests.

Query of δ_k: Let the top element of heap H_k when the query is made be κ. The oracle will return value $\delta = \frac{\eta}{\epsilon \cdot \kappa/2}$. Afterwards, the oracle proceeds to adjust it's max-heaps to the boost of coordinate k. Let x_+^* and x_-^* represent the state of x^* after and before having been incremented by $\delta \cdot e_k$ through *Boost(k)* respectively. The oracle iterates through all $j \in [m_c]$ such that $C^*(j, k) > 0$. For any such j if $(C_-^*)_j x^* < 2$ however $(C_+^*)_j x^* > 2$ the oracle proceeds as follows: for all $k' \in [n] | C^*(j, k') > 0$ the oracle removes $C^*(j, k')$ (or the value stored as the representative of $C^*(j, k')$ which might be somewhat smaller) from heap $H_{k'}$.

Claim C.10. *If the oracle is queried to return an estimate of δ_k it returns a value in $[\delta_k/4, \delta_k]$*

Proof. Assume *Boost(k)* was called and the oracle is queried to return an estimate of δ_k. Observe, that at all time points the heap H_k contains an object corresponding to each non-zero elements of the k-th column of P^*. Furthermore, H_k contains an element for each $C^*(j, k) : j \in [m_c] | C_j^* < 2$. Fix an element of P^* represented in H_k, say $P^*(i, k)$ and let $P_-^*(i, k)$ stand for the value it is represented under in the heap H_k. Observe that by construction $P_-^*(i, k) \le P^*(i, k) \le P_-^*(i, k)$. Fix an element of C^* represented in H_k, say $C^*(j, k)$ and let it's representation in H_k have value $C_-^*(j, k)$. Similarly, observe by construction that $C_-^*(j, k) \cdot 2 \ge C^*(j, k) \ge C_-^*(j, k)$.

Define $\kappa = \max\{\max_{i \in [m_p]} P^*(i, k), \max_{j \in [m_c] | C_j^* x^* < 2} C^*(j, k)\}$. By the observations above we can conclude that the maximum element of H_k is in $[\kappa/2, \kappa \cdot 2]$. Therefore, the oracle returns a value in $[\frac{\epsilon}{4 \cdot \kappa \cdot \eta}, \frac{\epsilon}{\kappa \cdot \eta}] = [\delta_k/4, \delta_k]$. $\qquad\square$

Claim C.11. *Let $B_k : k \in [n]$ represent the number of times Algorithm 1 calls subroutine Boost(k). The total time it takes to maintain the oracle and return it's queries is in $O(\sum_{k \in [n]} B_k N_k^C + N \cdot \log^2(N) + t)$.*

Proof. Say that the oracle re-adjust a constraint parameter $P^*(j, k)$ or $C^*(i, k)$ whenever it removes their outdated value for heap H_k and inserts their current value in the constraint matrix. Whenever a replacement of a constraint parameter occurs it's value in H_k has to double if it's a member of C^* or halve if its a member of P^* respectively. By Assumption C.2 non-zero parameters of constraint matrices always lie in $[1/poly(N), poly(N)]$ therefore each constraint element may has its value re-adjusted at most $O(\log(N))$

48

times. This means that handling re-adjustments of all elements of the heaps maintained by the oracle can be completed in $O(N \cdot \log^2(N))$ total time.

When an update is made to P^* the oracle checks the value of a single element in the heap, therefore it's work outside of possibly re-adjusting said value can be completed in $O(1)$ time. This means $O(N \cdot \log^2(N) + t)$ upper bounds the work of the oracle handling packing constraint updates.

When an update is made to C^*, say to $C^*(j, k)$, the oracle first has to check weather $C_j^* x^*$ has exceeded 2 for the first time. Checking the value of $C_j^* x^*$ can be completed in $O(1)$ time (assuming the algorithm keeps track of each constraints progress at all times). The value of $C^*(j, k)$ may only exceed 2 once over the total run of the iteration. At this point the oracle removes R_j^C elements from it's heaps. All of these removals therefore will take at most $O(N \cdot \log(N))$ total time. Therefore, $O(N \cdot \log^2(N) + t)$ upper bounds the work of the oracle handling covering constraint updates.

When a coordinate k is boosted by Algorithm 2 the oracle has to iterate over all non-zero elements of the k-th column of C^* which takes $O(N_k^C)$ time. For all $j \in [m_c] | C^*(j, k) > 0$ the oracle calculates $C_j^* x^*$ in $O(1)$. Afterwards, the oracle may proceed to remove all elements in any such line of C^*, however each element may only be removed at most once. Therefore, all work of the oracle due to boosting queries can be upper bounded by $O(\sum_{k \in [n]} B_k N_k^C + N \cdot \log(N) + t)$. This finishes the proof. □

C.6 The Handling of Translation Updates

Using Theorem C.1 as a black-box statement we will show how the algorithm can handle translation updates in addition to element-wise updates without incurring any blow-up in the total running time. To re-iterate a mixed packing covering linear program defined by constraint matrices (P, C) is the problem of finding x such that $Px \leq \mathbb{1}, Cx \geq \mathbb{1}$. A relaxing translation update of the packing constraint could formalized as follows: for some coordinate $i : i \in [m_p]$ and $\gamma > 0$ we update the linear program such that we are looking for x satisfying $Px \leq \mathbb{1} + \gamma \cdot e_i, Cx \geq \mathbb{1}$. Similarly, a relaxing translation update to the covering matrix is an updated defined by some $j \in [m_c]$ and $\gamma > 0$ such that we update the linear program to $Px \leq \mathbb{1}, Cx \geq \mathbb{1} - \gamma \cdot e_j$.

Observation C.8. *Let $\Delta_P \in \Re_{\geq 0}^{m_p}$ and $\Delta_C \in \Re_{\geq 0}^{m_c}$ where $0 \leq \Delta_P \leq \mathbb{1} \cdot \epsilon$ and $0 \leq \Delta_C \leq \mathbb{1} \cdot \epsilon$. If x is an ϵ-approximate solution to the linear program defined by constraint matrices (P, C) then x is a $2 \cdot \epsilon$ approximate solution to the linear program defined by $(P + \Delta_P, C - \Delta_C)$.*

As demonstrated by Observation C.8 if the algorithm ignores translation updates until any constraint is translated by more than an ϵ factor the solution returned is still $O(\epsilon)$ approximate to the linear program. Once a constraint (say the i-th pacing constraint) is translated by at least an ϵ factor the algorithm can simulate this relaxation through R_i^P entry updates of the packing matrix by relaxing all elements in the i-th row of P by the same multiplicative factor. Assuming similarly as in Assumption C.2 that all input parameters lie in the $[1/poly(N), poly(N)]$ range we can be certain that each constraint will need to be updated at most $O(\log_{1+\epsilon}(poly(N)) = O(\frac{\log(N)}{\epsilon})$ times due to being translated by more than an ϵ factor. Hence, the total number of entry-wise updates required to handle all translation updates will be bounded by $O(N \cdot \frac{\log(N)}{\epsilon})$.

Therefore, the algorithm can handle translation updates with $O(N \cdot \frac{\log(N)}{\epsilon} + t)$ additional time (where t here stands for the number of translation updates) as the algorithm of Theorem C.1 runs in linear time with respect to the number of entry wise updates. This implies that Theorem C.1 can be extended to work for translation updates without incurring any loss in running time.

49

References

[ABD22]   Sepehr Assadi, Aaron Bernstein, and Aditi Dudeja. Decremental matching in general graphs. In *49th International Colloquium on Automata, Languages, and Programming (ICALP 2022)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2022. → cited on page 2

[ABFP13]   Yossi Azar, Umang Bhaskar, Lisa Fleischer, and Debmalya Panigrahi. Online mixed packing and covering. In *Proceedings of the twenty-fourth annual ACM-SIAM symposium on Discrete algorithms*, pages 85–100. SIAM, 2013. → cited on page 5

[ABL20]   Sepehr Assadi, Aaron Bernstein, and Zachary Langley. Improved bounds for distributed load balancing. *arXiv preprint arXiv:2008.04148*, 2020. → cited on page 6

[ACC⁺18]   Moab Arar, Shiri Chechik, Sarel Cohen, Cliff Stein, and David Wajc. Dynamic matching: Reducing integral algorithms to approximately-maximal fractional algorithms. In Ioannis Chatzigiannakis, Christos Kaklamanis, Dániel Marx, and Donald Sannella, editors, *45th International Colloquium on Automata, Languages, and Programming, ICALP 2018, July 9-13, 2018, Prague, Czech Republic*, volume 107 of *LIPIcs*, pages 7:1–7:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018. → cited on page 1

[AEM92]   Pankaj K. Agarwal, David Eppstein, and Jirí Matousek. Dynamic half-space reporting, geometric optimization, and minimum spanning trees. In *33rd Annual Symposium on Foundations of Computer Science, Pittsburgh, Pennsylvania, USA, 24-27 October 1992*, pages 80–89. IEEE Computer Society, 1992. → cited on page 1

[AG11]   Kook Jin Ahn and Sudipto Guha. Linear programming in the semi-streaming model with application to the maximum matching problem. In *International Colloquium on Automata, Languages, and Programming*, pages 526–538. Springer, 2011. → cited on page 6

[AG18]   Kook Jin Ahn and Sudipto Guha. Access to data and number of iterations: Dual primal algorithms for maximum matching under resource constraints. *ACM Transactions on Parallel Computing*, 4(4):17:1–17:40, 2018. → cited on page 6

[AHK12]   Sanjeev Arora, Elad Hazan, and Satyen Kale. The multiplicative weights update method: a meta-algorithm and applications. *Theory Comput.*, 8(1):121–164, 2012. → cited on page 3, 8, 35

[AJJ⁺22]   Sepehr Assadi, Arun Jambulapati, Yujia Jin, Aaron Sidford, and Kevin Tian. Semi-streaming bipartite matching in fewer passes and optimal space. In *Proceedings of the Thirty-Third Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 627–669. SIAM, 2022. → cited on page 6

[AP14]   Pankaj K Agarwal and Jiangwei Pan. Near-linear algorithms for geometric hitting sets and set covers. In *Proceedings of the thirtieth annual symposium on Computational geometry*, pages 271–279, 2014. → cited on page 3, 4

[ARW17]   Amir Abboud, Aviad Rubinstein, and R. Ryan Williams. Distributed PCP theorems for hardness of approximation in P. In *58th IEEE Annual Symposium on Foundations of Computer Science (FOCS)*, pages 25–36, 2017. → cited on page 32

[AZO19]   Zeyuan Allen-Zhu and Lorenzo Orecchia. Nearly linear-time packing and covering lp solvers. *Mathematical Programming*, 175(1):307–353, 2019. → cited on page 1

[BBCG22] Suman K Bera, Sayan Bhattacharya, Jayesh Choudhari, and Prantar Ghosh. A new dynamic algorithm for densest subhypergraphs. In *Proceedings of the ACM Web Conference 2022*, pages 1093–1103, 2022. → cited on page 6

[BBR04] Yair Bartal, John W Byers, and Danny Raz. Fast, distributed approximation algorithms for positive linear programming with applications to flow control. *SIAM Journal on Computing*, 33(6):1261–1279, 2004. → cited on page 1

[BG95] Hervé Brönnimann and Michael T Goodrich. Almost optimal set covers in finite vc-dimension. *Discrete & Computational Geometry*, 14(4):463–479, 1995. → cited on page 3, 4

[BGS20] Aaron Bernstein, Maximilian Probst Gutenberg, and Thatchaphol Saranurak. Deterministic decremental reachability, scc, and shortest paths via directed expanders and congestion balancing. In *2020 IEEE 61st Annual Symposium on Foundations of Computer Science (FOCS)*, pages 1123–1134. IEEE, 2020. → cited on page 2, 5

[BHN19] Sayan Bhattacharya, Monika Henzinger, and Danupon Nanongkai. A new deterministic algorithm for dynamic set cover. In David Zuckerman, editor, *60th IEEE Annual Symposium on Foundations of Computer Science (FOCS) 2019, Baltimore, Maryland, USA, November 9-12, 2019*, pages 406–423, 2019. → cited on page 1

[BHR18] Aaron Bernstein, Jacob Holm, and Eva Rotenberg. Online bipartite matching with amortized replacements. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 947–959. SIAM, 2018. → cited on page 5

[BK21] Sayan Bhattacharya and Peter Kiss. Deterministic rounding of dynamic fractional matchings. In Nikhil Bansal, Emanuela Merelli, and James Worrell, editors, *48th International Colloquium on Automata, Languages, and Programming, (ICALP)*, pages 27:1–27:14, 2021. → cited on page 1

[BN09a] Niv Buchbinder and Joseph Naor. Online primal-dual algorithms for covering and packing. *Mathematics of Operations Research*, 34(2):270–286, 2009. → cited on page 5

[BN+09b] Niv Buchbinder, Joseph Seffi Naor, et al. The design of competitive online algorithms via a primal–dual approach. *Foundations and Trends® in Theoretical Computer Science*, 3(2–3):93–263, 2009. → cited on page 5

[Bsh16] Nader H Bshouty. Derandomizing chernoff bound with union bound with an application to k-wise independent sets. *arXiv preprint arXiv:1608.01568*, 2016. → cited on page 3

[CH20] Timothy M Chan and Qizheng He. Faster approximation algorithms for geometric set cover. *arXiv preprint arXiv:2003.13420*, 2020. → cited on page 3, 4

[Cla93] Kenneth L Clarkson. Algorithms for polytope covering and approximation. In *Workshop on Algorithms and Data Structures*, pages 246–252. Springer, 1993. → cited on page 3, 4

[CQ18] Chandra Chekuri and Kent Quanrud. Randomized mwu for positive lps. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 358–377. SIAM, 2018. → cited on page 1, 3

[Epp91] David Eppstein. Dynamic three-dimensional linear programming. In *32nd Annual Symposium on Foundations of Computer Science, San Juan, Puerto Rico, 1-4 October 1991*, pages 488–494. IEEE Computer Society, 1991. → cited on page 1

[Fle00]   Lisa K Fleischer. Approximating fractional multicommodity flow independent of the number of commodities. *SIAM Journal on Discrete Mathematics*, 13(4):505–520, 2000. → cited on page 3

[FS97]   Yoav Freund and Robert E Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of computer and system sciences*, 55(1):119–139, 1997. → cited on page 3

[FS99]   Yoav Freund and Robert E Schapire. Adaptive game playing using multiplicative weights. *Games and Economic Behavior*, 29(1-2):79–103, 1999. → cited on page 3

[GGK13]   Albert Gu, Anupam Gupta, and Amit Kumar. The power of deferral: maintaining a constant-competitive steiner tree online. In *ACM Symposium on Theory of Computing Conference (STOC)*, pages 525–534. ACM, 2013. → cited on page 5

[GK02]   Naveen Garg and Rohit Khandekar. Fast approximation algorithms for fractional steiner forest and related problems. In *The 43rd Annual IEEE Symposium on Foundations of Computer Science, 2002. Proceedings.*, pages 500–509. IEEE, 2002. → cited on page 3

[GK07]   Naveen Garg and Jochen Könemann. Faster and simpler algorithms for multicommodity flow and other fractional packing problems. *SIAM Journal on Computing*, 37(2):630–652, 2007. → cited on page 3

[GK14]   Anupam Gupta and Amit Kumar. Online steiner tree with deletions. In *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 455–467. SIAM, 2014. → cited on page 5

[GKKP17]   Anupam Gupta, Ravishankar Krishnaswamy, Amit Kumar, and Debmalya Panigrahi. Online and dynamic algorithms for set cover. In Hamed Hatami, Pierre McKenzie, and Valerie King, editors, *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2017, Montreal, QC, Canada, June 19-23, 2017*, pages 537–550. ACM, 2017. → cited on page 1, 2, 5

[GKL21]   Anupam Gupta, Gregory Kehne, and Roie Levin. Random order set cover is as easy as offline. *arXiv preprint arXiv:2111.06842*, 2021. → cited on page 3, 5

[GKS14]   Anupam Gupta, Amit Kumar, and Cliff Stein. Maintaining assignments online: Matching, scheduling, and flows. In *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 468–479. SIAM, 2014. → cited on page 5

[GN12]   Anupam Gupta and Viswanath Nagarajan. Approximating sparse covering integer programs online. In *International Colloquium on Automata, Languages, and Programming*, pages 436–448. Springer, 2012. → cited on page 5

[GP13]   Manoj Gupta and Richard Peng. Fully dynamic (1+ e)-approximate matchings. In *2013 IEEE 54th Annual Symposium on Foundations of Computer Science*, pages 548–557. IEEE, 2013. → cited on page 2

[HIPS19]   Niklas Hjuler, Giuseppe F. Italiano, Nikos Parotsidis, and David Saulpic. Dominating sets and connected dominating sets in dynamic graphs. In Rolf Niedermeier and Christophe Paul, editors, *36th International Symposium on Theoretical Aspects of Computer Science, STACS 2019, March 13-16, 2019, Berlin, Germany*, volume 126 of *LIPIcs*, pages 35:1–35:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. → cited on page 2

[HLLT06]   Nicholas JA Harvey, Richard E Ladner, László Lovász, and Tami Tamir. Semi-matchings for bipartite graphs and load balancing. *Journal of Algorithms*, 59(1):53–78, 2006. → cited on page 6

[HWC17]   Shuguang Hu, Xiaowei Wu, and TH Hubert Chan. Maintaining densest subsets efficiently in evolving hypergraphs. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*, pages 929–938, 2017. → cited on page 6

[IMR⁺17]   Piotr Indyk, Sepideh Mahabadi, Ronitt Rubinfeld, Jonathan Ullman, Ali Vakilian, and Anak Yodpinyanee. Fractional set cover in the streaming model. In *20th International Workshop on Approximation Algorithms for Combinatorial Optimization Problem (APPROX 2017)*, 2017. → cited on page 6

[IPZ01]   Russell Impagliazzo, Ramamohan Paturi, and Francis Zane. Which problems have strongly exponential complexity? *Journal of Computer and System Sciences*, 63(4):512–530, 2001. → cited on page 2

[JJST22]   Arun Jambulapati, Yujia Jin, Aaron Sidford, and Kevin Tian. Regularized box-simplex games and dynamic decremental bipartite matching. In Mikolaj Bojanczyk, Emanuela Merelli, and David P. Woodruff, editors, *49th International Colloquium on Automata, Languages, and Programming, ICALP 2022, July 4-8, 2022, Paris, France*, volume 229 of *LIPIcs*, pages 77:1–77:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. → cited on page 2, 5

[KY14]   Christos Koufogiannakis and Neal E. Young. A nearly linear-time PTAS for explicit fractional packing and covering linear programs. *Algorithmica*, 70(4):648–674, 2014. → cited on page 1, 3

[Lit88]   Nick Littlestone. Learning quickly when irrelevant attributes abound: A new linear-threshold algorithm. *Machine learning*, 2(4):285–318, 1988. → cited on page 3

[LN93]   Michael Luby and Noam Nisan. A parallel approximation algorithm for positive linear programming. In *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*, pages 448–457, 1993. → cited on page 1

[Mat93]   Jirí Matousek. Linear optimization queries. *J. Algorithms*, 14(3):432–448, 1993. → cited on page 1

[OvL80]   Mark H. Overmars and Jan van Leeuwen. Dynamically maintaining configurations in the plane (detailed abstract). In Raymond E. Miller, Seymour Ginsburg, Walter A. Burkhard, and Richard J. Lipton, editors, *Proceedings of the 12th Annual ACM Symposium on Theory of Computing, April 28-30, 1980, Los Angeles, California, USA*, pages 135–145. ACM, 1980. → cited on page 1

[OvL81]   Mark H. Overmars and Jan van Leeuwen. Maintenance of configurations in the plane. *J. Comput. Syst. Sci.*, 23(2):166–204, 1981. → cited on page 1

[PST95]   Serge A Plotkin, David B Shmoys, and Éva Tardos. Fast approximation algorithms for fractional packing and covering problems. *Mathematics of Operations Research*, 20(2):257–301, 1995. → cited on page 1, 3

[Qua20]   Kent Quanrud. Nearly linear time approximations for mixed packing and covering problems without data structures or randomization. In *3rd Symposium on Simplicity in Algorithms (SOSA)*, pages 69–80, 2020. → cited on page , 1, 3, 4, 6, 22, 23

[Tre98]   Luca Trevisan. Parallel approximation algorithms by positive linear programming. *Algorithmica*, 21(1):72–88, 1998. → cited on page 1

[Waj20]   David Wajc. Rounding dynamic matchings against an adaptive adversary. In Konstantin Makarychev, Yury Makarychev, Madhur Tulsiani, Gautam Kamath, and Julia Chuzhoy, editors, *Proccedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing, STOC 2020, Chicago, IL, USA, June 22-26, 2020*, pages 194–207. ACM, 2020. → cited on page 1

[WRM15]   Di Wang, Satish Rao, and Michael W Mahoney. Unified acceleration method for packing and covering problems via diameter reduction. *arXiv preprint arXiv:1508.02439*, 2015. → cited on page 1

[You14]   Neal E Young. Nearly linear-work algorithms for mixed packing/covering and facility-location linear programs. *arXiv preprint arXiv:1407.3015*, 2014. → cited on page , 1, 3, 4, 6, 22, 23, 29, 36

[ZN01]   Edo Zurel and Noam Nisan. An efficient approximate allocation algorithm for combinatorial auctions. In *Proceedings of the 3rd ACM conference on Electronic Commerce*, pages 125–136, 2001. → cited on page 1

[ZO15]   Zeyuan Allen Zhu and Lorenzo Orecchia. Nearly-linear time positive LP solver with faster convergence rate. In *Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing (STOC)*, pages 229–236. ACM, 2015. → cited on page 1, 3