Dynamic Algorithms for Packing-Covering LPs via Multiplicative Weight Updates

Sunil Vittal
sv2954@princeton.edu

Owen Yang
oy3975@princeton.edu

Devan Shah
ds6237@princeton.edu

December 16, 2023

1 Introduction

Let $A \in \mathbb{R}_{\geq 0}^{m \times n}$, $b \in \mathbb{R}_{\geq 0}^{m}$, and $c \in \mathbb{R}_{\geq 0}^{n}$. A **covering LP** $\mathbb{C}$ is an LP of the form

$$\min_{x \in \mathbb{R}_{\geq 0}^{n}} \{c^\top x \mid Ax \geq b\}. \tag{1}$$

A **packing LP** $\mathbb{P}$ is the dual LP of $\mathbb{C}$, which takes the form

$$\max_{y \in \mathbb{R}_{\geq 0}^{m}} \{b^\top y \mid A^\top y \leq c\}. \tag{2}$$

An ε-approximation for a covering-packing LP is a solution (x, y) satisfying $Ax \geq b$, $c^\top x \leq (1+\varepsilon)\,\mathrm{OPT}$, $A^\top y \leq c$, and $b^\top y \geq \frac{\mathrm{OPT}}{1+\varepsilon}$. The authors consider a dynamic setting of a series of updates by an adversary changing one entry of A, b, or c. Thus, the goal is to acquire an ε-approximation of a solution to either the covering LP or the dual packing LP as fast as possible after these adversarial actions.

Updates to an LP can be classified into **relaxing** updates and **restricting** updates. An update is restricting to $\mathbb{C}$ (and relaxing to $\mathbb{P}$) if it decreases an entry of A or increases an entry of b or c. We may analogously define updates relaxing $\mathbb{C}$ and restricting $\mathbb{P}$ as increasing an entry of A or decreasing an entry of b or c. A sequence of updates to an LP is called **partially dynamic** if they all restrict $\mathbb{C}$, or they all relax $\mathbb{C}$.

The main result achieved by the paper "Dynamic Algorithms for Packing-Covering LPs" by Bhattacharya, Kiss, and Saranurak, is the following theorem, where N denotes the maximum number of non-zero entries in the input LP throughout the sequence of updates:

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

2 Covering-Packing LPs

Covering-Packing LPs represent a large subset of interesting LP problems, with notable special cases being bipartite matching and set cover.

To note the connection to set cover, consider a universe $[m]$ with n sets $S_1, S_2, \ldots S_n \subset [m]$. For each S_i, construct the indicator vector $s^i \in \mathbb{R}^m$ which satisfies $s^i_j = 1$ if $j \in S_i$ and 0 otherwise. For an element $j \in [m]$ and a collection $\bigcup_{i \in A} S_i$ with $A \subset [n]$, we have that

$j \in \bigcup_{i \in A} S_i \iff 1 \leq \sum_{i \in A} s_j^i = \sum_{i \in [n]} x_i s_j^i$ where x is an indicator variable for A (i.e. $x_i = 1$ iff $i \in A$). Thus, we can model set cover by the LP:

$$\min_{x \in \mathbb{R}^n_{\geq 0}} \{\mathbb{1} \cdot x \mid Ax \geq \mathbb{1}\}$$

Where $A_{ji} = (s_j^i)$ and thus $Ax \geq \mathbb{1}$ is equivalent to $\sum_i x_i s_j^i \geq 1, \forall j \in [m]$, and thus equivalent to the constraint $[m] \subset \bigcup_{i \in A_x} S_i$ where $e \in A_x$ iff $x_e = 1$, and the goal is to minimize $\mathbb{1} \cdot x = |A_x|$ and thus we encode the aim of finding the smallest collection of sets that contain all elements.

Rather than considering the optimization problems of Eq. 1 and Eq. 2, we can reduce these problems to a sequence of approximation feasibility subproblems.

For the purposes of rescaling, we can construct matrix $A' = (\frac{A_{ij}}{b_j c_j})_{ij} \in \mathbb{R}^{m \times n}_{\geq 0}$, to reach that:

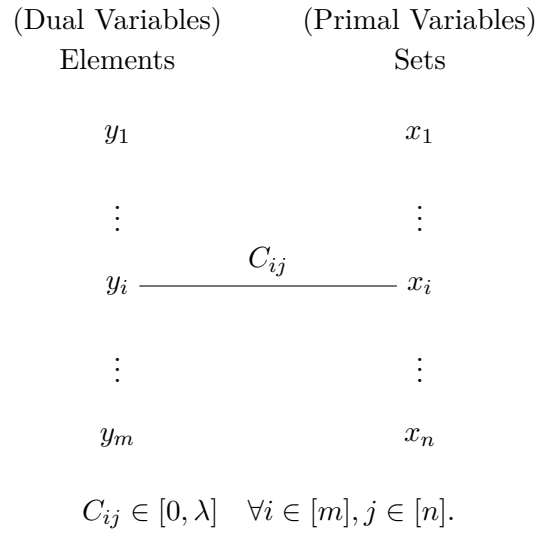
$$\min_{x \in \mathbb{R}^n_{\geq 0}} \{c^\top x \mid Ax \geq b\} = \min_{x' \in \mathbb{R}^n_{\geq 0}} \{||x'||_1 \mid A'x' \geq 1\}$$
$$\max_{y \in \mathbb{R}^m_{\geq 0}} \{b^\top y \mid A^\top y \leq c\} = \max_{y' \in \mathbb{R}^m_{\geq 0}} \{||y'||_1 \mid A'^\top y' \leq 1\}$$

And now to solve the RHS LPs, we can reduce this problem to one of feasibility and binary search, i.e. for the covering LP, is there a solution $x' \in \mathbb{R}^n_{\geq 0}$ to $||x'||_1 \leq \lambda$ and $A'x' \geq 1$, which after a final rescaling $A'' = \frac{1}{\lambda} A'$, is equivalent to the feasibility problem of, is there a solution $x'' \in \mathbb{R}^n_{\geq 0}$ with $||x''||_1 = 1$ to $A''x'' \geq 1$. For the remainder of the paper and for bounds on future arguments, we consider the following feasability equivalent of Covering-Packing LPs:

For $C \in \mathbb{R}^{m \times n}_{\geq 0}$,
Does there exist either $x \in \mathbb{R}^n_{\geq 0}$ s.t $\sum_{j \in [n]} x_j = 1$ and $C \cdot x \geq \mathbb{1}$
or a $y \in \mathbb{R}^m_{\geq 0}$ s.t $\sum_{i \in [m]} y_i = 1$ and $C^\top \cdot y \leq \mathbb{1}$?

By the binary-search approach for converting feasability to optimization, an ϵ-approximate solution to this feasibility problem may used to solve the generic case with an additional factor of $\Theta\left(\frac{\log(\frac{nU}{L})}{\epsilon}\right)$ time, where L and U are the smallest and largest nonzero entries respectively of C. This holds true for both the static and partially dynamic cases.

Note that we can still view the feasibility version of covering-packing LPs as a variant of set cover. Consider the below complete bipartite graph.



$$C_{ij} \in [0, \lambda] \quad \forall i \in [m], j \in [n].$$

The primal covering LP has the constraint on the element, and so we are examining each element and ensuring the weighted sum of the edges to it exceeds $1 - \epsilon$ for the ϵ -approximate covering, which corresponds to the sets amply covering the element. For more elaboration on this bipartite graph intuition, we refer the reader to (5).

3 Whack-a-Mole Algorithm

The Whack-a-Mole algorithm is a Multiplicative Weights Update (MWU) algorithm, so solutions are built in, say t, iterations, but due to the dependency of the solution from round to round, the authors state that an MWU approach will likely be useless in a fully dynamic setting. In particular, they claim that in an MWU-based algorithm, there can only be a propagation of changes if the updates are partially dynamic. In other words, the constraint of partially dynamic updates allows solutions from previous rounds to remain close to the feasible solution in future rounds, which is critical for the results we present. As a result, prior changes remain in the proper direction despite updates, which is critical for quickly finding $x^{(t+1)}$ after updated inputs.

The authors describe this challenge using the language of two players simultaneously solving two related problems, based on the previous reduction from covering-packing LPs to a more tractable form. Given a matrix $C \in \{0, 1\}^{m \times n}$, we consider a feasibility problem such that we must find a vector $x \in \mathbb{R}^n_{\geq 0}$ satisfying $\|x\|_1 = \mathbb{1}^\top x \leq 1$ and $Cx \geq 1$. We also consider the dual problem, which is finding $y \in \mathbb{R}^n_{\geq 0}$ satisfying $\mathbb{1}^\top y \geq 1$ and $C^\top y \leq 1$. We aim to find an ϵ-approximate solution in either x or y. They view MWU problems with a Whacking player and Greedy player.

1. The Whacking player starts with $x = \mathbb{1}/n$. The player performs *whacking* on the constraint i, if constraint i is violated, which occurs when $C_i x < 1$. In this case, we update x_j via $(1 + \varepsilon)x_j$ forall j when $C_{ij} = 1$.

2. The Greedy player starts with the zero vector. With $y = 0$, whenever $C_i x < 1$, perform $y_i = y_i + 1$.

3. We terminate this game if either the Whacking player can no longer whack any more constraints, in which case we output x, or if T rounds have passed, in which case we output $y^* := \frac{y}{T}$.

One observes that the Whacking perspective corresponds to a Covering LP while the Greedy perspective corresponds to a Packing LP. Indeed, in the former, we aim to minimize $\mathbb{1}^\top x$ with

the constraints $C^\top x \geq 1,\ x \in \mathbb{R}^n_{\geq 0}$, and for the latter, we will show analogously that y^* satisfies $\mathbb{1}^\top y^* = 1$ and $C^\top y^* \leq (1+\varepsilon)\mathbb{1}$. This general framework extends to the partially dynamic setting, as either all updates relax or restrict the LP. As a result, if the updates are all restricting, we look at the Whacking player's perspective. Note that if a constraint could have been whacked before, it can still be whacked as the update is restricting. Therefore, we could have chosen to have taken those past actions before or now and they would still be equally valid. Conversely, if the updates are relaxing, we take the role of the Greedy player as the updates we have made previously are still valid choices for updates.

4 Whack-a-Mole MWU Algorithm

We consider the following problem, equivalent to finding an ϵ-approximation of either the covering LP or dual packing LP: Given a matrix $C \in [0, \lambda]^{m \times n}$ where $\lambda > 0$, either return a vector $x \in \mathbb{R}^n_{\geq 0}$ with $\mathbb{1}^\top x \leq 1 + \Theta(\varepsilon)$ and $Cx \geq (1 - \Theta(\varepsilon)) \cdot \mathbb{1}$, or return a vector $y \in \mathbb{R}^m_{\geq 0}$ with $\mathbb{1}^\top y \geq 1 - \Theta(\varepsilon)$ and $C^\top y \leq (1 + \Theta(\varepsilon)) \cdot \mathbb{1}$.

4.1 Static Setting

Algorithm 1 The Whack-a-Mole MWU Algorithm

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 $x^1 \leftarrow \frac{\hat{x}^1}{\|\hat{x}^1\|_1}$.
for $t = 1$ to T **do**
    EITHER
        Conclude that $(C \cdot x^t)_i \geq 1 - \epsilon$ for all $i \in [m]$.
        Terminate the FOR loop, and RETURN (x^t, NULL).
    OR
        Find a covering constraint $i_t \in [m]$ such that $(C \cdot x^t)_{i_t} < 1$.
        $\hat{x}^{t+1} \leftarrow \text{WHACK}(i_t, \hat{x}^t)$. ▷ See Algorithm 2.
        $x^{t+1} \leftarrow \frac{\hat{x}^{t+1}}{\|\hat{x}^{t+1}\|_1}$.
        Let $y^t \in \Delta^m$ be the vector where $(y^t)_{i_t} = 1$ and $(y^t)_i = 0$ for all $i \in [m] \setminus \{i_t\}$.
end for
$y \leftarrow \frac{1}{T} \cdot \sum_{t=1}^{T} y^t$.
RETURN (NULL, y).

Algorithm 2 WHACK$(i, \hat{x})$

for all $j \in [n]$ **do**
    $\hat{z}_j \leftarrow (1 + \epsilon \cdot \frac{C_{ij}}{\lambda}) \cdot \hat{x}_j$.
end for
RETURN $\hat{z}$

Algorithm Examples We provide video examples of this algorithm at: `https://github.com/devs-cs/MWUVisualization`

We will initially consider a static algorithm that gives intuition for the dynamic case. If the condition x is approximately feasible is true, we have that the algorithm is satisfied. We will next show the correctness of the algorithm.

Theorem 2 *After $T = \frac{\lambda \ln n}{\epsilon^2}$ rounds, if x is not approximately feasible, we must have that y is an approximately feasible solution to the packing LP. Thus, we will either return a solution to the covering LP or packing LP.*

Proof: To derive bounds from the Multiplicative Weights Setting, consider the n experts labeled $\{1, 2, \ldots, n\}$. At each round $t \in T$, we pick an expert according to the probability distribution $\frac{w^t}{||w^t||_1}$, initiated as $w^1 = \vec{1}$. The costs for each decision are decided as follows. Choose $i_t \in [m]$ such that the constraint is violated. We then define that, for expert j, the cost is $p_j^t = \frac{1}{\lambda} \cdot C_{i_t j} \in [0, 1]$. We then perform the update $w_j^{t+1} = w_j^t(1 + \epsilon \cdot p_j^t)$, $\forall j \in [n]$.

From (1), by the Multiplicative Weights Update algorithm, we have that for any $j \in [n]$,

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

Thus, as $p_j^T \geq 0$, and dividing by T, we have that:

$$(1 - \epsilon) \cdot \frac{\sum_t (p^t)_j}{T} \leq \frac{\sum_t (p^t)^\top \cdot x^t}{T} + \frac{\ln(n)}{\epsilon T}$$

Note that the above setting and updates are identical to that performed in the Whack-a-Mole MWU algorithm. Associate each covering constraint to an expert, and correspond the vector $\hat{x}^t$ to w, noting that the update conditions and initial values are identical. As i_t denotes a violated constraint, we have that $(Cx^t)_{i_t} < 1$ and thus $(p^t)^\top \cdot x^t = (1/\lambda) \cdot (Cx^t)_{i_t} \leq 1/\lambda$, and thus $\frac{\sum_t (p^t)^\top \cdot x^t}{T} \leq \frac{1}{\lambda}$. Additionally, note that $\frac{\ln(n)}{\epsilon \cdot T} = \frac{\epsilon}{\lambda}$, and thus we can bound the RHS of the above equation by $\frac{1}{\lambda} + \frac{\epsilon}{\lambda} = \frac{1+\epsilon}{\lambda}$.

Now let us consider the LHS of the equation. Note that for each $j \in [n]$, we have by the definition of y^t that

$$(p^t)_j = \frac{C_{i_t j}}{\lambda} = \frac{1}{\lambda} \cdot (C^\top y^t)_j.$$

Then writing $y := \frac{1}{T} \sum_{t=1}^{T} y^t$, we have

$$\frac{\sum_t (p^t)_j}{T} = \frac{\sum_t (C^\top y^t)_j}{\lambda T} = \frac{1}{\lambda} (C^\top y)_j.$$

Thus,

$$(1 - \epsilon) \frac{1}{\lambda} (C^\top y)_j \leq \frac{1 + \epsilon}{\lambda} \implies (C^\top y)_j \leq \frac{1 + \epsilon}{1 - \epsilon} = 1 + \Theta(\epsilon).$$

Thus, the vector y approximately satisfies the packing LP, and we are done.

Key Takeaway #1

We can view the algorithm as representing a two player game. We have the *Whack-A-Mole* player and the *Greedy* player:

1. **Whack-a-Mole Player:** Starts with $x = \mathbb{1}/n$ and if x violates a constraint $i \in [m]$ (i.e $C_i x < 1$), update x via MWU to move towards constraint satisfaction. Terminate and return x if no constraint is violated.
2. **Greedy Player:** Starts with $y = \mathbf{0}$ and for any $i \in [m]$ s.t. $C_i x < 1$, increment y_i by 1. Terminate after T rounds and return $y^* = y/T$

Leveraging the bounds of the MWU regret property, we receive that either the Whack-a-Mole player returns a solution x to the covering LP or the Greedy Player returns a solution y to the dual covering LP.

Implementing the Whack-a-Mole Algorithm

The above algorithm can be implemented in a runtime that is "approximately linear", or more precisely $O(N \cdot \frac{\log n}{\epsilon^2} \cdot \log^2(\frac{\lambda \log n}{\epsilon}))$ for N being the number of nonzero entries of $C \in [0, \lambda]^{m \times n}$. Instead of updating $\hat{x}$ one step at a time, we may split the algorithm into phases where the sum of the weights increase by a factor of at most $(1 - \frac{\epsilon}{2})^{-1}$. The main purpose of the phases is to bound the number of updates that can happen in each phase, as well as the number of total phases.

Since in each phase the total sum of weights $\|\hat{x}_t\|_1$ remains approximately the same, the resulting approximation $\tilde{x}_t = \frac{\hat{x}_t}{W}$, where we now normalize by the sum of the weights at the beginning of the phase, rather than $\frac{\hat{x}_t}{\|\hat{x}_t\|_1}$ is still close to the actual solution. With this rounding, we now have that within a phase, enforcing one constraint does not affect whether a separate constraint is satisfied, as applying WHACK on a single coordinate of $\hat{x}_t$ will not change $\tilde{x}_t$ for other coordinates. This allows us to then WHACK constraints in arbitrary order and be satisfied that, within the given phase, if we ensure a constraint is satisfied, it will remain satisfied in that phase.

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

$\hat{x}^1 \leftarrow \mathbb{1}, t \leftarrow 1$, and $T \leftarrow \lambda \ln(n)/\epsilon^2$
LOOP
$W \leftarrow \|\hat{x}^t\|_1$
for all $i \in [m]$ **do**
    if $(C \cdot \frac{\hat{x}^t}{W})_i < 1 - \epsilon/2$ **then**
        $\delta \leftarrow$ ENFORCE$(i, t, \hat{x}^t, W)$. ▷ See Algorithm 4.
        $t \leftarrow t + \delta$
        if $t = T$ **then**
            Terminate the LOOP and return (NULL, y), where $y := \frac{1}{T} \cdot \sum_{t'=1}^{T} y^{t'}$.
        end if
        if $\|\hat{x}^t\|_1 > (1 - \epsilon/2)^{-1} \cdot W$ **then**
            Go to step (03). ▷ Initiate a new phase.
        end if
    end if
end for
Terminate the LOOP and return (x^t, NULL), where $x^t := \frac{\hat{x}^t}{\|\hat{x}^t\|_1}$.

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

$\delta \leftarrow$ STEP-SIZE$(i, t, \hat{x}^t, W)$. ▷ See Algorithm 5.
for $t' = t$ to $(t + \delta - 1)$ **do**
    $\hat{x}^{t'+1} \leftarrow$ WHACK$(i, \hat{x}^{t'})$. ▷ See Algorithm 2.
    $i_{t'} \leftarrow i$.
    Let $y^{t'} \in \Delta^m$ be the vector where $(y^{t'})_i = 1$ and $(y^{t'})_{i'} = 0$ for all $i' \in [m] \setminus \{i\}$.
end for
return δ.

One optimization over the naive approach of just iterating one repetition of WHACK is to perform is to binary search on the number of times a constraint needs to be whacked until it is either satisfied or the phase is over. Then, since all the WHACK operation does is multiply each entry of x by a constant depending on i and j, we may just do this exponentiation all at once, reducing this particular subroutine from a linear to logarithmic runtime.

First, to show the correctness of the algorithm, note that it will definitely run the WHACK subroutine at most T times, which we call rounds. Then it will just return the greedy player's solution, exactly the same as in the previous description of the algorithm. We will only whack a constraint which is not approximately satisfied, i.e. is less than $1 - \frac{\epsilon}{2}$. Since the sum of the weights only increases when whacking

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

For every integer $k \geq 1$, let $z^k \in \mathbb{R}^n_{\geq 0}$ be such that $(z^k)_j = (1 + \epsilon \cdot \frac{C_{ij}}{\lambda})^k \cdot (\hat{x}^t)_j$ for all $j \in [n]$.
if $\left(C \cdot \frac{z^{T-t}}{W}\right)_i < 1$ **then**
    $\delta \leftarrow T - t$.
else
    Using binary search, compute the smallest integer $\delta \in [T - t]$ such that $\left(C \cdot \frac{z^\delta}{W}\right)_i \geq 1$.
end if
return δ.

a constraint, we have that in a given iteration whacking constraint i from time t to $t + \delta$,

$$\forall t' \in [t, t + \delta), C \cdot \frac{\hat{x}^{t'}}{\|\hat{x}^{t'}\|_1} \leq C \cdot \frac{\hat{x}^{t'}}{W} < 1.$$

The last inequality follows from $1 - \frac{\epsilon}{2}$ in a given phase (and since it's being whacked, it is less than $1 - \frac{\epsilon}{2}$).

Now if the algorithm terminates after t^* rounds and returns the whacking solution rather than the greedy solution, we are guaranteed that for all constraints, $C \cdot \frac{\hat{x}^{t^*}}{W} \geq 1 - \frac{\epsilon}{2}$, where W is the sum of the weights at the beginning of the last phase. Furthermore, we have that $W \geq (1 - \frac{\epsilon}{2})\|\hat{x}^{t^*}\|_1$. Here lies the significance of choosing $1 - \frac{\epsilon}{2}$ rather than $1 - \epsilon$ as our bound for checking constraints: defining $x^{t^*} := \frac{\hat{x}^{t^*}}{\|\hat{x}^{t^*}\|_1}$, we now find that

$$C \cdot x^{t^*} \geq (1 - \frac{\epsilon}{2}) C \cdot \frac{\hat{x}^{t^*}}{W} \geq (1 - \frac{\epsilon}{2})^2 = 1 - \epsilon + \frac{\epsilon^2}{4} > 1 - \epsilon$$

for all constraints, so the solution x^{t^*} satisfies the same property $C \cdot x^{t^*} \geq (1 - \epsilon)\mathbb{1}$ as in the previous description of the algorithm.

Now, we prove some bounds on the running times of STEP-SIZE and ENFORCE.

Lemma 1 *STEP-SIZE$(i, t, \hat{x}^t, W)$ can be implemented in $O(N_i \cdot \log^2(\frac{\lambda \log n}{\epsilon}))$ time where N_i is the number of nonzero entries in C_i (the ith row of C).*

Proof: It takes $O(N_i \log \kappa) < O(N_i \log T)$ time to compute $(C \cdot \frac{z^\kappa}{W})_i$ for any κ, since all we need to do is exponentiate to the κ power for each nonzero entry. Since this quantity increases as κ increases, we can binary search for $\delta \in [1, T - t]$, this takes at most $\log T$ iterations, for a total of $O(N_i \log^2 T)$ time. Now setting $T = \frac{\lambda \log n}{\epsilon^2}$ gives the desired bound.

Lemma 2 *ENFORCE$(i, t, \hat{x}^t, W)$ can be implemented in $O(N_i \cdot \log^2(\frac{\lambda \log n}{\epsilon}))$ time.*

The key idea here is that the STEP-SIZE subroutine dominates the runtime.
Proof: Once we have computed the value of δ using STEP-SIZE, it only takes $\log \delta$ exponentiations to perform a WHACK δ times for one index. Since WHACK does nothing if $C_{ij} = 0$, we in fact only need $O(N_i \log \delta)$ time to do this. Furthermore, since all the $y^{t'}$s are equal, we may just multiply the first one by δ to get their sum. Since $\delta \leq T$, we find that the time to compute STEP-SIZE dominates the running time of ENFORCE, as desired.

The other respective half of the bound is to bound the number of phases that can happen. Essentially, the reason why there shouldn't be too many phases is because we restrict the number of allowed WHACKs, and each WHACK can't increase the norm of $\hat{x}$ by a lot. Therefore, since in every phase $\|\hat{x}\|_1$ increases by a set factor, we can get a bound on the total number of phases. More formally, we have the following:

Lemma 3 $\|\hat{x}^t\|_1 \leq n^{\frac{1}{\epsilon}+1}$ *for all t.*

Proof: For any given t we have that

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

where $x^t = \frac{\hat{x}^t}{\|\hat{x}^t\|_1}$. The last inequality here was proven in the previous section about correctness of the algorithm. Then we have that

$$\left\|\hat{x}^{t+1}\right\|_1 < (1+\frac{\epsilon}{\lambda})\left\|\hat{x}^t\right\|_1 \implies \left\|\hat{x}^T\right\|_1 < (1+\frac{\epsilon}{\lambda})^T\left\|\hat{x}^0\right\|_1 = (1+\frac{\epsilon}{\lambda})^T n \leq e^{\frac{\log n}{\epsilon}} n = n^{\frac{1}{\epsilon}+1}.$$

Remark 1 *The authors' computation and statement of this lemma contains a minor typo, stating the upper bound is $n^{1/\varepsilon}$, but as the computation illustrates, it is actually $n^{O(1/\varepsilon)}$.*

Lemma 4 *There are at most $O(\frac{\log n}{\epsilon^2})$ phases.*

Proof: Since in each phase, the norm changes by a factor of at least $(1-\frac{\epsilon}{2})^{-1}$, there are at most $\log_{(1-\frac{\epsilon}{2})^{-1}}(n^{\frac{1}{\epsilon}+1}) = \frac{(\frac{1}{\epsilon}+1)\log n}{\log\left((1-\frac{\epsilon}{2})^{-1}\right)} = \frac{(\frac{1}{\epsilon}+1)\log n}{-\log\left(1-\frac{\epsilon}{2}\right)} \leq \frac{2}{\epsilon}\cdot\left(1+\frac{1}{\epsilon}\right)\log(n) = O\left(\frac{\log n}{\epsilon^2}\right)$.

Theorem 3 *The total runtime of Algorithm 3 is $O(N\cdot\frac{\log n}{\varepsilon^2}\cdot\log^2(\frac{\lambda\log n}{\varepsilon}))$ where N is the total number of nonzero entries in the matrix $C\in[0,\lambda]^{m\times n}$*

Proof: For a particular phase, we must enforce all the constraints. Recalling that ENFORCE takes $O(N_i\cdot\log^2(\frac{\lambda\log n}{\varepsilon}))$ time to implement, we enforce every constraint, taking a total of

$$\sum_{0\leq i\leq m} O\left(N_i\cdot\log^2\left(\frac{\lambda\log n}{\varepsilon}\right)\right) = O\left(\sum_i N_i\cdot\log^2\left(\frac{\lambda\log n}{\varepsilon}\right)\right) = O\left(N\cdot\log^2\left(\frac{\lambda\log n}{\varepsilon}\right)\right)$$

Since there are $O(\frac{\log n}{\epsilon^2})$ phases, the total runtime is $O(N\cdot\frac{\log n}{\varepsilon^2}\cdot\log^2(\frac{\lambda\log n}{\varepsilon}))$ as desired.

Key Takeaway #2

By partitioning our updates into phases determined by the amount $||\hat{x}||_1$ increases, we can bound the total amount of phases knowing that $||\hat{x}||_1 \leq n^{\Theta(1/\epsilon)}$. We also have that, for each phase, as $||\hat{x}||_1$ does not change greatly, whacking one constraint will not affect whether another constraint is approximately satisified, and thus we can more efficient WHACK constraints, leading to improved runtime guarantees.

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

5.1 Dynamic Algorithm Overview and Implementation

We next modify the prior static algorithm for the dynamic setting.

We are given a constraint matrix $C\in[0,\lambda]^{m\times n}$, and subsequently the constraint matrix C has restricting updates decreasing values of entries C_{ij}. This algorithm only handles the restricting case, and the next section will consider relaxing updates. As before, we maintain either a vector $\tilde{x}$ such that $\sum_i \tilde{x}_i \leq 1+\Theta(\epsilon)$ and $C\tilde{x}\geq(1-\Theta(\epsilon))\cdot\mathbb{1}$ or y such that $\sum_i y_i = 1$ and $C^T y \leq (1+\Theta(\epsilon))\cdot\mathbb{1}$.

At preprocessing, we run the static Whack-a-Mole MWU algorithm. If the algorithm returns y, note that updates decreasing C_{ij} will maintain y as an approximately feasible solution to the dual, and thus we can simply maintain y without processing any updates.

Thus, we only consider the case the algorithm returns vector $x^t := \hat{x}^t/\|\hat{x}^t\|_1$. To account for a restricting update to C_{ij}, we run Algorithm 3 steps $(5)-(12)$. We have that if any constraint in the LP is violated from this update, it must be the constraint $i\in[m]$. We thus check whether the current solution $\tilde{x}^t := \hat{x}^t/W$ approximately satisfies constraint i. If not, we will enforce the constraint by repeatedly whacking it. This process will result in one of the following 3 cases:

- **Case 1:** If W is no longer an accurate estimate of $||\tilde{x}^t||_1$, we will then begin a new phase. Accordingly, a new phase will begin whenever $\|x_t\|_1$ increases by a multiplicative factor of $(1-\varepsilon/2)^{-1}$.
- **Case 2:** If $t=T$, we may then return the approximately feasible solution y to the dual packing LP. We no longer have to update y as it will remain approximately feasible as updates continue to be restricting, i.e. relaxing to the packing LP.
- **Case 3:** If we are in neither of the above cases, then $\tilde{x}^t$ is an approximate solution for the Covering LP as we can repeatedly whack a constraint, provided we remain in the same phase.

5.2 Dynamic Algorithm Guarantees

It is straightforward to check that Lemma 3 still holds even if there are restricting updates. This is because the key observation that $(Cx^t)_{i_t} < 1$ for all t still holds true in the dynamic case, since decreasing an entry of C can only decrease this value.

We now show that this algorithm will still return either x or y with the same precision as in the static case. Because all entries of $\tilde{x}^t$ are monotonically increasing with t, we have that for all constraints i that are not being enforced, the value of $(C\tilde{x}^t)_i$ still at least $1-\epsilon$. Also, by the same proofs as before, after repeatedly whacking constraint i' if necessary, we find that $(C\tilde{x}^t)_{i'}$ is also at least $1-\epsilon$. We have that at the beginning of a phase, by definition $\|\tilde{x}^t\|_1 = 1$. Then at the end of the phase we must have $\|\tilde{x}^t\|_1 \leq (1-\frac{\epsilon}{2})^{-1} < 1+\epsilon$, so this other bound still holds. Therefore, the bulk of the proof of correctness lies in the following claim:

Lemma 5 *If the dynamic algorithm returns a vector y, then $\|y\|_1 = 1$ and $C^\top y \leq \mathbb{1} \cdot (1+4\epsilon)$.*

Proof: Although we had the analogues guarantee in the static setting, as we now have that the matrix C has decrementing terms, we must show this property is maintained. We map Whack-a-Mole back to the expert setting as we did for Theorem 2. Let C^t stand for the state of C at the time of the t-th call to WHACK. Let us reconsider the equation we get from considering experts, which I include for ease here:

$$(1-\epsilon)\cdot\frac{\sum_t (p^t)_j}{T} \leq \frac{\sum_t (p^t)^\top \cdot x^t}{T} + \frac{\ln(n)}{\epsilon T} \quad \forall j \in [m] \tag{3}$$

As the algorithm chooses a violated constraint to whack each round, we again have that $(C^t x^t)_{i_t} < 1$, and thus we can bound the RHS of by $\frac{1+\epsilon}{\lambda}$ identically as we did before.

As $(p^t)_j = \frac{1}{\lambda}\cdot C^t_{i_t j} = \frac{1}{\lambda}\cdot((C^t)^\top y^t)_j$, we thus derive:

$$\frac{1}{T}\sum_{t=1}^{T}(p^t)_j = \frac{1}{T}\sum_{t=1}^{T}((C^t)^\top y^t)_j \geq \frac{1}{T}\sum_{t=1}^{T}((C^T)^\top y^t)_j = \frac{1}{T}((C^T)^\top y)_j$$

with the inequality holding as the elements in C only decrease over time. Thus substituting into Eq. 3 our bounds for the LHS and RHS, we reach

$$\frac{1-\epsilon}{\lambda}\cdot\left((C^T)^\top y\right)_j \leq \frac{1}{\lambda}\cdot(1+\epsilon) \implies \left((C^T)^\top y\right)_j \leq 1+4\epsilon \quad \forall j \in [m].$$

We now provide a key bound on how many times a constraint can be enforced total in the entire running of the dynamic algorithm. This has major implications: after a very large number of updates when all constraints have been enforced the maximum number of times, no more constraints will ever be updated. This provides a very strong bound the total update time of this dynamic algorithm that depends only linearly on the number of updates.

Lemma 6 *Throughout the entire duration of the dynamic algorithm, a constraint i can be enforced at most $O(\frac{\log n}{\varepsilon^2}\cdot\log\frac{\lambda\log n}{\varepsilon})$ times.*

Proof: We say the step-size of a constraint is δ if it has been whacked δ times within an enforcement. Note that $\delta \in [1,T]$, so we discretize this interval into $O(\log T)$ intervals of powers of 2. Define the *rank* of an enforcement κ to be the unique integer such that $\delta \in [2^{\kappa-1}, 2^\kappa)$.

For every fixed κ with $1 \leq \kappa \leq \log T$, we will demonstrate that the constraint $i \in [m]$ has at most $O\left(\frac{log(n)}{\epsilon^2}\right)$ many enforcements of rank κ. Intuitively, the upper bound of 2^κ gives us a good bound on how much $\|\hat{x}\|_1$ increases per WHACK, and the lower bound of $2^{\kappa-1}$ ensures that there are many WHACKs in each enforcement. Since we have the bound of $\|\hat{x}\|_1 \leq n^{\frac{1}{\epsilon}+1}$, this allows us to bound the number of WHACKs and consequently the number of enforcements.

Formally, suppose that there are $\gamma+1$ enforcements total for some integer γ. If $\gamma = 0$ then clearly we are done. Otherwise, for the second-to-last enforcement, at say, time t_γ, we have that $C\cdot\frac{\hat{x}^t}{W}$ is at most $1-\frac{\epsilon}{2}$ at the beginning of the phase and at least 1 at the end of the phase (we do not use the last because the phase could potentially end too early for the latter bound to hold). Therefore, $C\cdot\frac{\hat{x}^t}{W}$ increases by a factor of at least $(1-\frac{\epsilon}{2})^{-1} \geq 1+\frac{\epsilon}{2}$. Then there is some coordinate j^* that increases by at least a factor

of $1+\frac{\epsilon}{2}$. However, since the step size is at most 2^κ, we also have that the coordinate can increase by at most a factor of $\left(1+\epsilon\frac{C_{ij^*}^{t\gamma}}{\lambda}\right)^{2^\kappa}$, i.e.

$$\left(1+\epsilon\frac{C_{ij^*}^{t\gamma}}{\lambda}\right)^{2^\kappa} \geq 1+\frac{\epsilon}{2}.$$

Since the step sizes of any previous enforcements is at least $2^{\kappa-1}$, we find that in all previous enforcements, the coordinate must have changed by a factor of at least

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

Since the initial value of $\hat{x}_{j^*}^t$ is 1, we find that after γ of these rank κ enforcements, $\hat{x}_{j^*}^t \geq (1+\frac{\epsilon}{2})^{\frac{\gamma}{2}}$. Then using Lemma 3 we find that

$$(1+\frac{\epsilon}{2})^{\frac{\gamma}{2}} \leq \left\|\hat{x}^t\right\|_1 \leq n^{\frac{1}{\epsilon}+1} \implies \frac{\gamma}{2}\log\Big(1+\frac{\epsilon}{2}\Big) \leq \left(\frac{1}{\epsilon}+1\right)\log n = O\left(\frac{\log n}{\epsilon^2}\right)$$

where the final bound is analogous to the proof of Lemma 4.

Since there are at most $\log T$ possible values of κ, we find that the number of total enforcements is at most $O(\frac{\log n}{\epsilon^2}\log T) = O\left(\frac{\log n}{\epsilon^2}\log\Big(\frac{\lambda\log n}{\epsilon}\Big)\right)$ as desired.

This gives us the promised upper bound on the total update time, which is linear in the (sufficiently large) number of updates:

Theorem 4 *The algorithm will handle any sequence of τ restricting updates to C in $O\left(\tau + N\cdot\frac{\log n}{\epsilon^2}\cdot\log^3(\frac{\lambda\log n}{\epsilon})\right)$ time, where N is the number of initially nonzero entries of C.*

Proof: For each update to some element C_{ij}, we must first decide whether constraint i needs to be enforced, and then enforce it if necessary. For the former, we maintain variables $\hat{z}_j$ for every $j\in[n]$ that will take on values $(1+\epsilon)^\kappa$ for some nonnegative integer κ. At all times t, we check that $\hat{z}_j$ is within a $1+\epsilon$ factor of $\hat{x}_j^t$ and update it if not. This gives us a good estimate for coordinate j, so by maintaining the matrix $C\cdot\frac{\hat{z}}{W}$, all entries are within a $1+\epsilon$ factor of the "true" value, $C\cdot\frac{\hat{x}}{W}$. This allows us to check whether constraint i is approximately satisfied in $O(1)$ time by simply subtracting out $(C_{ij}^{t-1}-C_{ij}^t)\cdot\frac{\hat{z}_j}{W}$ and checking whether it is still greater than $1-\frac{\epsilon}{2}$.

Now because every coordinate of $\hat{x}$ is at most $n^{\frac{1}{\epsilon}+1}$ by Lemma 3, we must have that $\hat{z}_j$ can be updated at most $\log_{1+\epsilon}(n^{\frac{1}{\epsilon}+1}) = O(\frac{\log n}{\epsilon^2})$ times. When this value changes, we spend $O(N_j)$ time to update the jth column of the matrix $C\cdot\frac{\hat{z}}{W}$, where N_j is the number of nonzero entries in column j. Therefore, the total time spent on updating the matrix is at most $O(\sum_{j\in[n]}N_j\cdot\frac{\log n}{\epsilon^2}) = O(N\cdot\frac{\log n}{\epsilon^2})$. Therefore, the total time spent on this subtask is $O(\tau + N\cdot\frac{\log n}{\epsilon^2})$.

For the latter, we have by Lemma 2 that we can enforce a constraint $i\in[m]$ in time $O\left(N_i\cdot\log^2\left(\frac{\lambda\log(n)}{\epsilon}\right)\right)$, and thus by the bound provided by Lemma 6, we have that, for any constraint $i\in[m]$, the total amount of time spent enforcing the constraint 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 over all constraints gives a total runtime of at most $O\left(N\cdot\frac{\log(n)}{\epsilon^2}\cdot\log^3\left(\frac{\lambda\log(n)}{\epsilon}\right)\right)$. Combining with the first subtask, we have a total runtime of $O\left(\tau + N\cdot\frac{\log(n)}{\epsilon^2}\cdot\log^3\left(\frac{\lambda\log(n)}{\epsilon}\right)\right)$ as desired.

Key Takeaway #3

At a high level, this algorithm works under restricting updates as any constraints violated on C^t would have been violated under C^{t+1}, and thus past whacking actions remain valid. Since we retain the bound that $\|\hat{x}\|_1 \leq n^{\frac{1}{\epsilon}+1}$ and we only enforce constraints that require us to increase $\|\hat{x}\|_1$ by a noticeable factor, we can bound the total amount of enforcements that can occur on each constraint.

Moreover, as C is decreasing element wise, we can retain guarantees that if $t=T$, the associated value of y will solve the packing LP, and thus the termination conditions of the static

algorithm work in the dunamic setting. Therefore, in the long run, we are able to use the static algorithm with minor changes for the dynamic setting to achieve constant update time.

6 Greedy MWU Algorithm

In the prior sections, we were observing the case where we are given a sequence of restricting updates, in which we discussed that we would like to take the perspective of the Whack-a-Mole player. Now, we consider the case where we are given a sequence of relaxing updates, essentially reversing the problem. In fact, the exposition and results will feel very similar. Throughout this section, we deal with *Positive* LPs, which ask the following: Find $x \in \mathbb{R}^n_{\geq 0}$ such that $Px \leq a$ and $Cx \geq b$ where $P \in \mathbb{R}^{m_p \times n}_{\geq 0}, a \in \mathbb{R}^{m_p}_{\geq 0}, C \in \mathbb{R}^{m_c \times n}_{\geq} 0, b \in \mathbb{R}^{m_c}_{\geq 0}$, where m_p is the number of rows for P and m_c is the number of rows of C.

We desire to find an $\varepsilon-$approximate solution to such LPs which is either an x satisfying $Px \leq (1+\varepsilon)a$ and $Cx \geq (1-\varepsilon)b$, or $\perp$ to indicate an infeasible solution. An update will be relaxing if it increases an entry of C or a, or decreases an entry of P or b. Note that on initial rounds, the program will return $\perp$, but we desire to return a feasible x after a certain point in time. After this, x will continue to be an approximate solution in future updates. As more definitions, we will say an update is an entry update if it changes P or C. Meanwhile, an update is said to be a translating update if it changes a or b. Let L and U be upper and lower bounds on the values of P, C, a, b. Let N be an upper bound on the total number of nonzero entries in P, C.

The authors focus on the setting where all the relaxing updates are entry updates since if we are happy with an ε-approximate solution to the input LP, we can ignore translation updates to the constraints as long as a_i/b_j (just the coordinates of a, b) does not increase/decrease by more than a factor of $1+\varepsilon$. Moreover, we observe that we can simulate a translation update by applying as many entry updates as there are nonzero entries in P_i, C_j(ith row of P, jth row of C) to C and P. In particular, if we need to scale a_i, b_j by a factor of $\alpha \geq 1$, we can scale up P_i, C_j by that same factor. As a result, it suffices to consider only entry updates, and for convenience we scale a, b to 1. Then, our constraints become, $Px \leq 1$, $Cx \geq 1$.

6.1 Static Greedy MWU Algorithm

The authors attribute this static algorithm to (3) and (4), but we summarize the exposition contained in (2). We can define two functions to phrase this Positive LP question in an alternative way. Define

$$f_p(x) = \frac{1}{\eta} \log \sum_{i=1}^{m_p} exp(\eta P_i x) \tag{4}$$

$$f_c(x) = \frac{1}{\eta} \log \sum_{i=1}^{m_c} exp(-\eta P_i x) \tag{5}$$

$$\eta = \frac{\log(m_p + m_c + U/L)}{\varepsilon} \tag{6}$$

which satisfies

$$\max_{i \in [m_p]} P_i x \leq f_p(x) \leq \max_{i \in [m_p]} P_i x + \varepsilon$$

$$\min_{j \in [m_c]} C_j x - \varepsilon \leq f_c(x) \leq \min_{j \in [m_c]} C_j x$$

for all x, so the algorithm desires to find x satisfying $f_p(x) \leq 1$ and $f_c(x) \geq 1$ as such x is immediately an approximate solution by the above two equations. We define a coordinate $k \in [n]$ to be cheap with respect to a point x if and only if $\langle \nabla f_p(x), e_k \rangle \leq (1 + \Theta(\varepsilon)) \cdot \langle \nabla f_c(x), e_k \rangle$, where e_k is the kth standard basis vector. We can make this more precise with more notation for coming observations.

We associate a weight to each LP constraint. For all $i \in [m_p], j \in [m_c]$, we define $w_p(x, i) := exp(\eta P_i x)$ and $w_c(x, j) := exp(-\eta C_j x)$. Moreover, we define $w_p(x) = \sum_{i=1}^{m_p} w_p(x, i)$ and $w_x(x) := \sum_{j=1}^{m_c} w_c(x, j)$. The cost of a coordinate k at a given x is then

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

and k is said to be cheap w.r.t x if and only if $\lambda(x, k) \leq (1 + \Theta(\varepsilon)) \cdot w_p(x)/w_c(x)$. We now have the following algorithm.

Algorithm 6 A static algorithm for solving a positive LP (P, C).

INITIALISE $x \leftarrow 0 \in \mathbb{R}^n_{\geq 0}$.
while $\min_{j\in[m]} C_j \cdot x < 1$ **do**
  if there is no cheap coordinate at x **then**
    return that the LP is infeasible.
  else
    Let $k \in [n]$ be a cheap coordinate.
    $x \leftarrow \text{BOOST}(x, k)$.
  end if
end while
return x.

Algorithm 7 BOOST(x, k).

Find the maximum δ such that $\max_{i\in[m]_P} P(i,k)\cdot\delta \leq \epsilon/n$ and $\max_{j\in[m]_C; j\neq x} C(j,k)\cdot\delta \leq \epsilon/n$.
return $x + \delta e_k$.

The algorithm starts with $x = 0$. In every iteration of the WHILE loop, we either conclude that the input LP is infeasible, or we find a cheap coordinate and boost x a small amount in the direction of that cooridnate. This algorithm has the following properties, which we state without proof:

- **Property 1:** If there is no cheap cooridnate at x, then $LP(P, C)$ is infeasible.
- **Property 2:** Let $\varepsilon < 1/10$ and let k be a cheap cooridnate at y and update z with BOOST(y, k). Then $f_p(z) - f_p(y) \leq (1-\Theta(\varepsilon))\cdot(f_c(z) - f_c(y)) \leq \Theta(\varepsilon)$. Note that as $x = 0$ initially, $f_p(x), f_c(x) = 0$ as well, so Property 2 gives way to the following invariant.
- **Property 3:** We always have $f_p(x) \leq (1 + \Theta(\varepsilon))f_c(x)$.Moreover, each call to BOOST additivly increases $f_c(x)$ by at most $\Theta(\varepsilon)$.

Lemma 7 *If Algorithm 6 returns an x, then x is an $\Theta(\varepsilon)$ approximate solution to the input LP. Otherwise, the algorithm correctly declares that the input LP is infeasible*

Proof: If the algorithm terminates because it found an x, note that in the prior iteration of the loop, we had $f_p(x) \leq (1 + \Theta(\varepsilon))f_c(x)$ and $f_c(x) \leq \min_{j\in[m_c]} C_j x < 1$. Note that the loop terminates when $f_c(x) \geq 1$, by Property 3, we have $1 \leq f_c(x) \leq 1 + \Theta(\varepsilon)$. Moreover, we also have $\max_{i\in[m_p]} P_i x \leq f_p(x) \leq (1 + \Theta(\varepsilon))f_c(x) \leq (1 + \Theta(\varepsilon))^2 \leq 1 + \Theta(\varepsilon)$ as ε is small, so x is a $\Theta(\varepsilon)-$approximate solution to the Input LP. If we haven't found an x, Property 1 states that the input LP is infeasible.

6.2 Dynamic Greedy MWU

This section contains many propositions with long and technical proofs, so we will omit proofs and instead opt to state these propositions and offer broad descriptions of the overall algorithm.

Like the dynamic Whack-a-Mole, we start initialization by running the static Greedy algorithm. If this outputs an approximate solution, we are done since this output is a still a solution after any future relaxing updates. After pre-processing, the dynamic algorithm addresses relaxing updates to the covering constraints. After a relaxing update, we keep boosting x along cheap coordinates until we can't anymore or $\min_{j\in[m_j]} C_j \geq 1$. In the former, we output this LP is infeasible and in the latter, we output that x is an approxmate solution.

This dynamic algorithm still has Property 3, and as a result, we end up with $\max_{i\in[m_p]} P_i x \leq 1 + \Theta(\varepsilon)$ as we proved earlier. From here, we observe that a relaxing update to C or boosting x can only increase $w_p(x, i)$ for packing constraint i or decrease $w_c(x, j)$ for covering constraint j. It follows that the cost of any coordinate can only increase after BOOST. Moreover, if $C(j, k)$ is relaxed, the cost of $k' \neq k$ relative to x can only increase, but the cost of k can either increase or decrease.

With this information, we now describe the algorithm more. We maintain $x \in \mathbb{R}^n_{\geq 0}, w_p(x, i), w_c(x, j)$ for all $i \in [m_p], j \in [m)c], w_p(x), w_c(x)$, and all the values of $P_i x, C_j x$. We can define phases in the following manner. We calculate a ratio $\gamma(x) = w_p(x)/w_c(x)$. Note that this value is 1 at the start of the

algorithm since $x = 0$. We will say a new phase starts whenever $\gamma(x)$ increase by a factor of $1 + \Theta(\varepsilon)$. This factor is intuitive due to Property 3, so that a phase is spanned by calls to BOOST or a mixture of calls to BOOST with some relaxing updates to C.

Initialize $\gamma^0(x)$ to be the starting value of $\gamma(x)$. We will say a cooridnate $k \in [n]$ is cheap if and only if $\lambda(x, k) \leq (1 + \Theta(\varepsilon))\gamma^0(x)$. With this classification, maintain a list of cheap coordinates. At the start of the algorithm we initialize this list E to be $[n]$, and then call BOOST-ALL(x, E), which essentially forces the particular k in WHILE loop iterations to no longer be cheap. Moreover, the monotonicity of costs implies that k will never be cheap again within the same phase!

For handling a relaxing update during a phase, note that after BOOST-ALL, our list E is empty, so a relaxing update to an entry $P(i, k)$ could only lead to k as a possible cheap coordinate. If k does become cheap, we set $E = \{k\}$ and perform BOOST-ALL(x, E), so that we keep boosting along k. Note that by properties described earlier, boosting k won't contribute to other coordinates becoming cheap. As a result, we've handled the update accordingly.

Algorithm 8 BOOST-ALL(x, E).

while $E \neq \emptyset$ **do**
  Consider any $k \in E$.
  $E \leftarrow E \setminus \{k\}$.
  while $\lambda(x, k) \leq (1 + \Theta(\epsilon)) \cdot \gamma^0(x)$ **do**
    $x \leftarrow \text{BOOST}(x, k)$. ▷ See Algorithm 7.
    if $\min_{j \in [m]_C} C_j \cdot x \geq 1$ **then**
      return x.
    end if
    if $\gamma(x) > (1 + \Theta(\epsilon)) \cdot \gamma^0(x)$ **then**
      Terminate the current phase.
    end if
  end while
end while

From this description, the following are the main results about this dynamic greedy MWU algorithm.

Lemma 8 *Fix any coordinate $k \in [n]$. The dyanmic algorithm described above calls the subrountine BOOST(x, k) at most $O(\frac{\log^2(m_c+m_p+U/L)}{\varepsilon^2})$*

Lemma 9 *The dynamic algorithm described above has the most $O(\frac{\log(m_c+m_p+U/L)}{\varepsilon^2})$ phases.*

Combining these results with the static algorithm runtime from preprocessing of $O(N \cdot \frac{\log(m_c+m_p)}{\varepsilon^2})$, we have the following.

Theorem 5 *We can maintain an $\varepsilon-$approximate solution to a positive LP undergoing a sequence of t relaxing updates in $O(N \cdot \frac{\log^2(m_c+m_p+U/L)}{\varepsilon^2} + t)$ total update time*

7 The Full Algorithm

The main change here is that relaxing updates can occur to P or C. Note that the prior section considering the case where relaxing updates occured to C, but if a relaxing update occurs to P, $P(i, k)$ gets reduced for some i, k, so $w_p(x, i)$ decreases, breaking the monotonicity of weights from earlier. The authors form a new LP to work around this shift in circumstance.

The Extended LP

1. Define $P^* \in \mathbb{R}_{\geq 0}^{m_p \times (n+1)}$, where $P^*(i,k) = P(i,k)$ and $P^*(i,n+1) \geq 0$ for all $i \in [m_p], k \in [n]$.
2. Define $C^* \in \mathbb{R}_{\geq 0}^{m_p \times (n+1)}$, where $C^*(j,k) = C(j,k)$ and $C^*(j,n+1) = 0$ for all $j \in [m_c], k \in [n]$.
3. LP(P,C) is feasible if and only if the extended LP(P^*,C^*) is feasible. One notices this as the vector x solves LP(P,C) leads to a soluton to LP(P^*,C^*) via $(x,0)$. Conversely, just ignore the $(n+1)$ dimension entries to get a solution to LP(P,C).

Initialization: Set P^* to be the zero matrix and we start with $x^* = (x_1^*, x_2^*, \ldots, x_n^*, x_{n+1}^*) = (0,\ldots,0,1)$, so that $f_p(x^*) = f_c(x^*) = 0$. We desire to modify x^* so that it is a $\Theta(\varepsilon)-$approximation to the extended LP(P^*,C^*). These updates are along the same theme as the Greedy MWU algorithms by finding cheaper coordinates and boosting x^*. Moreover, we assume that changes to P, C are reflected in the corresponding indices in P^*, C^* respectively.

For the extended LP, after any relaxing update to an entry $P(i,k)$ of P, we increase the value of $P^*(i,n+1)$ such that $P_i^* x^*$ remains unchanged. We do this so that $w_p(x^*)$ remains unchanged and this manipulation allows us to find a solution to the extended LP. We will call these pseudo-updates.

Recall the initial Problem from Whack-a-Mole section: Given a matrix $C \in [0,\lambda]^{m \times n}$ where $\lambda > 0$, either return a vector $x \in \mathbb{R}_{\geq}^n 0$ with $\mathbb{1}^\top x \leq 1 + \Theta(\varepsilon)$ and $Cx \geq (1-\Theta(\varepsilon)) \cdot \mathbb{1}$, or return a vector $y \in \mathbb{R}_{\geq 0}^m$ with $\mathbb{1}^\top y \geq 1 - \Theta(\varepsilon)$ and $C^\top y \leq (1+\Theta(\varepsilon)) \cdot \mathbb{1}$. Now we desire to maintain a solution when C is experiencing relaxing updates, so each update increases a value of C. The idea is essentially to run the Greedy MWU dynamic algorithm on the positive LP defined by $\mathbb{1}^\top x \leq 1$ and $Cx \geq \mathbb{1}$. When this algorithm has finished processing an update, we have two cases

1. The algorithm returns a solution x satisfying satisfying the conditions of the problem. We accept this x and terminate the algorithm as this vector continues to be an approximate solution after future updates.
2. The algorithm declares that the positive LP is infeasible since there is no cheap coordinate to boost on. We define a vector $\hat{(}y)$ consisting of $\hat{y}_j = w_c(x,j)/w_c(x)$ (basically a vector containing normalized weights of every column). We desire to show that $\hat{y}$ is a feasible solution to the dual packing LP.

   Indeed, this is not hard to see, as $\hat{y}$ consists of normalized weights of the covering constraints, so $\mathbb{1}^\top \hat{y} = 1$. Now, consider a coordinate k which is not cheap. It follows from the definition of cheapness, that

$$\frac{1}{\sum_{j \in [m_c]} \hat{y}_j C(j,k)} \geq 1 + \Theta(\varepsilon)) \implies (C^\top \hat{y})(k) = \sum_{j \in [m_c]} \hat{y}_j C(j,k) \leq 1$$

   so it follows that $C^\top \hat{y} \leq \mathbb{1}$.

Keeping this information in mind, the algorithm maintains an estimated weight $(1-\varepsilon)w_c(x) \leq w_c^*(x) \leq (w_c(x))$. If we end up in the second case, we return $y = (y_j = \frac{w_c(x,j)}{w_c^*(x)})$. We observe $\hat{y}_j \leq y_j \leq (1+\Theta(\varepsilon))\hat{y}_j$, and so we get $\mathbb{1}^T y \geq 1$ and $C^T y \leq (1+\Theta(\varepsilon))\mathbb{1}$ as a result of the feasibility of $\hat{y}$ for the dual packing LP.[1] Note that storing y is not computationally expensive since we store $w_c(x,j)$ for all j and we don't have to change $w_c^*(x)$ while we are within the same phase.

Indeed, recall $\gamma(x) = w_p(x)/w_c(x)$ does not change by more than a factor of $1+\Theta(\varepsilon)$ within a given phase. Moreover, $w_p(x^*)$ is unchanged!, so checking this condition is the same as checking the change of $w_c(x)$. Then if $w_c(x)$ has changed by a factor of $1+\Theta(\varepsilon)$, we begin a new phase. We also recall that there are $O(\frac{\log(m_c+m_p+U/L)}{\varepsilon^2})$ phases in the dynamic Greedy MWU algorithm and as there are m_c columns, it takes $O(m_c)$ time to update y in a new phase. We can combine this with Theorem 4, giving the following theorem.

Theorem 6 *We can determinisitically maintain a solution that satisfies $\mathbb{1}^\top x \leq 1 + \Theta(\varepsilon)$ and $Cx \geq (1-\Theta(\varepsilon)) \cdot \mathbb{1}$ or a solution to its dual when C undergoes t relaxing entry updates in*

$$O\left(t + N \cdot \frac{\log^2(m_c + U/L)}{\varepsilon^2} + m_c \cdot \frac{\log(m_c+U/L)}{\varepsilon^2}\right) = O\left(t + N \cdot \frac{\log^2(m_c+U/L)}{\varepsilon^2}\right)$$

[1] The paper maintains that $w_c(x) \leq w_c^*(x) \leq (1+\varepsilon)(w_c(x))$, leading to a false conclusion, so we have tweaked estimate range, to match more with the proof.

We can then perform a similar reduction from this setting to Packing and Covering LPs to acquire the main result.

Theorem 7 *Consider any sequence of t relaxing updates to a covering LP, where each update either increases an entry of C or decreases an entry of a, b. Let N be the maximum number of non-zero entries in C, let L/U be the lower/upper bound on the minimum/maximum value of a non-zero entry of C, a, b. We can deterministically maintain an $\varepsilon-$approximate optimal solution to the Covering LP in $O(t \cdot \frac{\log(nU/L)}{\varepsilon} + N \cdot \frac{\log^2(m+U/L)}{\varepsilon^2} \cdot \log(nU/L))$ total time. This also holds for such a solution to a packing LP with restricting updates.*

References

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

[2] Sayan Bhattacharya, Peter Kiss, and Thatchaphol Saranurak. Dynamic Algorithms for Packing-Covering LPs via Multiplicative Weight Updates. arXiv preprint arXiv: 2207.07519, 2022.

[3] 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.

[4] Neal E Young. Nearly linear-work algorithms for mixed packing/covering and facility-location linear programs. arXiv preprint arXiv:1407.3015, 2014

[5] Simons institute and Sayan Bhattacharya. Dynamic Graphs and Algorithm Design. `https://www.youtube.com/watch?v=QucpI6Ww8EM`