Online Unrelated-Machine Load Balancing and Generalized Flow with Recourse

Ravishankar Krishnaswamy
rakri@microsoft.com
Microsoft Research India
India

Shi Li
shil@buffalo.edu
University at Buffalo
USA

Varun Suriyanarayana
vs478a@cornell.edu
Cornell University
USA

ABSTRACT

We consider the recourse version of the classical online load-balancing problem on unrelated machines, where the algorithm is allowed to re-assign prior jobs. We give a \((2 + \epsilon)\)-competitive algorithm for the problem with \(O_\epsilon(\log n)\) amortized recourse per job. This is the first \(O(1)\)-competitive algorithm for the problem with non-trivial recourse, and the competitive ratio nearly matches the long-standing best-known offline approximation guarantee. We also show an \(O(\log \log n/\log \log \log n)\)-competitive algorithm for the problem with \(O(1)\) amortized recourse. The best-known bounds from prior work are \(O(\log \log n)\)-competitive algorithms with \(O(1)\) amortized recourse due to Gupta et al., for the special case of the restricted assignment model.

Along the way, we design an algorithm for the recourse version of the online generalized network flow problem (also known as the network flow problem with gains). We have a graph with costs and capacities on the edges, and sources arrive online. Upon arrival of a source, we need to send unit flow from the source. In contrast to standard network flow, every edge \(uv\) in the network has a \(\gamma\)-factor gain \(\gamma_{uv} > 0\), meaning that \(\gamma\)-units of flow sent from \(u\) across \(uv\) becomes \(\gamma_{uv}\) units of flow when it reaches \(v\). In the recourse version, the algorithm can undo prior flow sent on an edge by incurring a linear cost. We present an online algorithm for the problem with recourse at most \(O(1/\epsilon)\) times the offline optimum cost flow for the instance when edge capacities are scaled by a factor of \(\frac{1}{\epsilon^2}\). This marks an improvement over prior work in two ways: the known algorithms only apply to standard network flow (i.e., unit gains), and secondly, the guarantees held against an offline flow when edge capacities are scaled by a factor of \((2 + \epsilon)\). As an immediate corollary of this, we also obtain an improved algorithm for the online \(b\)-matching problem with reassignment costs.

CCS CONCEPTS

• Theory of computation → Online algorithms.

1 INTRODUCTION

Load balancing is one of the fundamental problems in online algorithms with several real-world motivations. Its clean formulation has also led to the development of several techniques in online algorithms. In this paper we study the power of recourse/re-assignments for the online load balancing problem on unrelated machines. In the OLBwR (Online Unrelated Machine Load Balancing with Recourse) problem, we are given a set \(M\) of \(m\) machines, and \(n\) jobs \([n]\) arrive online. Job \(i\) \([n]\) arrives at time \(t\), if assigned to machine \(i\), would induce a load of \(p_{it}\) on the machine. The goal is to assign jobs to machines to minimize the maximum load of any machine, which is the sum of loads of jobs assigned to it. The algorithm can also re-assign prior jobs, and we separately track the recourse to be the total number of re-assignments over the course of arrivals. The above is a very natural problem to study, since jobs/demands typically arrive in an online manner, and moreover, real-world systems (see, e.g. [1]) often migrate jobs between servers to achieve better balance. However, since migrating jobs is a disruptive operation, we seek to minimize the total number of movements while also ensuring nearly balanced assignments at all times.

Definition 1.1 (\(\alpha\)-competitive, \(\beta\)-amortized recourse algorithms). Let \(\text{Opt}\) denote the maximum load of any machine in an optimal assignment for the first \(t\) jobs. An algorithm is said to be \(\alpha\)-competitive with \(\beta\)-amortized recourse if the maximum load over all machines of its assignment is most \(\alpha\ \text{Opt}\), and the total number of re-assignments done through the first \(t\) job arrivals is at most \(\beta\).

This problem has received significant attention since the 1990s. The classical online load balancing problem has the same model as OLBwR, with the restriction that the online algorithm’s assignments are irrevocable, i.e., no recourse is allowed. A series of works introduced several elegant ideas culminating with tight \(\Theta(\log m)\)-competitive algorithms [3], with matching lower bound for any randomized online algorithm. The power of recourse has also been studied for the load balancing problem, albeit from the perspective of handling job departures [16], where it is evident that no online algorithm can have non-trivial competitive ratios without recourse. To the best of our knowledge, [9] were the first to study the...

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

STOC ’23, June 20–23, 2023, Orlando, FL, USA
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-9913-5/23/06...
https://doi.org/10.1145/3564246.3585522

775
the power of recourse to get improved guarantees for job arrivals (the same setting as ours), and showed that $O(1)$-amortized recourse can yield $O(\log \log mn)$-competitive algorithms for online load balancing for the restricted assignment problem (where each job can only be assigned to a subset $N(j)$ of machines, but it has the same processing time of $p_j$ on any of these machines). This bound represents an exponential improvement when compared to the $\Omega(\log m)$ lower bounds on the competitive ratio for the same model when no recourse is allowed. The central focus of this paper is in trying to understand if we can get similar improvements for the OLBwR on unrelated machines. We obtain the following results:

**Theorem 1.2.** For any constant $\epsilon > 0$, there is an efficient deterministic $(2 + \epsilon)$-competitive algorithm for OLBwR on unrelated machines with $O\left(\frac{\log n \log(1/\epsilon)}{\epsilon}\right)$-amortized recourse.

Note that we are able to get a competitive ratio bound nearly matching the best known offline approximation factor [11, 15] for this classic problem, with a small recourse. We also design another algorithm with a different competitive ratio-recourse tradeoff.

**Theorem 1.3.** There is an efficient randomized algorithm for OLBwR on unrelated machines which is $O(\log n / \log \log n)$-competitive and has $O(1)$-amortized recourse. The approximation ratio holds with high probability, and the recourse bound holds in expectation with high probability.

Our algorithms work by maintaining a $(1 + \epsilon)$-competitive fractional assignment online, and then rounding it in an online manner while ensuring that both steps do not modify the solution too much. Prior to our work, no algorithm that maintained constant-competitive fractional solutions with bounded recourse was known. A crucial observation central to our result is that the fractional assignments for unrelated machines can be seen as a special case of the generalized flow (aka min-cost flow with gains) problem. Here, we are given a directed graph $G = (V, E)$ with cost $c(e)$ and capacities $\mu_e$ on the edge set $E$. The generalization from standard flow comes from the fact that the amount of flow exiting an edge is now a scalar multiple of the amount entering it. This is captured by gain factors $g_e$ on the edges, which represent the extent to which one unit of flow originating at one end-vertex gets scaled when it reaches the other end-vertex. We are given a set $S$ of sources, each of which wants to send one unit of flow, and a sink $r$ which can absorb the flows. The goal is to maintain minimum cost flows which can send unit from the sources while ensuring flow conservation at all vertices $s \in V \setminus S \cup \{r\}$. Since we seek to maintain online fractional solutions for OLBwR, we introduce and study the OGNF (Online Generalized Network Flow) problem. Here, the sources arrive one by one, and the algorithm needs to pay the incremental cost of sending one unit flow from the new source, on top of the existing flow. Unlike classical offline network flows, undoing a flow also counts positively to our overall cost since it is a form of recourse. The goal is to minimize the total cost in comparison with the offline min-cost generalized flow. Informally, we obtain the following result, and state the formal version in Theorem 2.1.

**Theorem 1.4.** There is an efficient deterministic $O\left(\frac{1}{\epsilon}\right)$-competitive algorithm for OGNF when the algorithm can violate edge capacities by a factor of $1 + \epsilon$.

Theorem 2.1 represents a two-fold advancement over prior work for this problem [9]. Firstly, [9] presents $O(1)$-competitive algorithms with $(2 + \epsilon)$-factor capacity violations, while our results improve this to $(1 + \epsilon)$ violation, and secondly, our algorithms can handle arbitrary gain factors, while the GKS-algorithms only applies to classical network flow setting with unit gains.

We now discuss our final algorithmic contribution. Since our results improve the capacity violation even for the regular flow problem, this immediately translates to an improvement for the so-called Online b-Matching with Reassignment Costs (ObMwRC) problem. Here we have a bipartite graph $G = (L \cup R, E)$. Right vertices $v \in R$ each have capacities $b_v \in \mathbb{Z}_{\geq 0}$, and the goal is to assign the left vertices while respecting the capacities. It is guaranteed that $G$ has a valid $b$-matching: a matching where each $u \in L$ is matched once and every $v \in R$ is matched at most $b_v$ times. In the online problem, the left vertices arrive one by one. When a left-vertex $u \in L$ arrives, it specifies its neighbors $N(u) \subseteq R$ in $G$, and a reassignment cost $c_u \geq 0$. Each time we re-assign $u \in L$ to a different vertex $v \in R$, we incur a cost of $c_u$. We need to always maintain an assignment which violates the capacities by a small amount while minimizing the total re-assignment small cost.

**Theorem 1.5.** There is an efficient deterministic algorithm for ObMwRC which (i) maintains a matching of all the left vertices, (ii) ensures that each $v \in R$ is matched at most $[(1 + \epsilon)b_v]$ times, and (iii) has $O\left(\frac{1}{\epsilon}\right)$ amortized recourse.

This improves over the prior factor of violation $2 + \epsilon$ [9].

Finally, one may ask what we can do in the fully-dynamic model for the online load balancing problem with recourse. That is, jobs may arrive and depart. In Appendix B, we show that in this case, even the offline algorithm which knows the whole sequence of the job arrivals and departures ahead of time needs to incur an amortized recourse of $n^{O(1/\epsilon)}$, in order to achieve an $\alpha$-competitive ratio. Thus, to circumvent the negative result, one needs to consider a different measurement for recourse for the fully dynamic model.

### 1.1 Our Techniques, at a High Level

**Online Generalized Flow.** [9] studied the special case with unit gains, and analyzed a natural algorithm which, for each new source arrival, sends unit flow along the shortest path from the source to the sink in the residual graph. However, unlike the residual graphs in offline flow problems where backward arcs have negative costs, the GKS-algorithm forces both forward as well as backward arcs to have the same non-negative cost $c_e$ to accurately reflect the recourse incurred. For the analysis, they define a quantity height(s) to be the cost of the shortest path to the sink in the residual graph. This captures the augmentation cost of $s$ when it arrives. A crucial property they show is that height(s) is non-decreasing over the course of future arrivals. They use the final height-values to exhibit a good dual solution to the offline LP of the min-cost flow instance. These steps allow them to relate the online and offline costs.

How do we extend these ideas to the generalized flow problem? Intuitively, when a new source arrives, a natural strategy might be to augment along the shortest path in the residual graph. While this is a reasonable idea, observe that the concept of augmenting paths is very different when there are gains. Indeed, we could potentially...
augment along a path in the residual graph from the source to a cycle which does not even contain the sink, provided the product of gains on the cycle is < 1 (these are called flow-absorbing cycles in literature). Keeping this in mind, we generalize the notion of height above to be the minimum cost way to send one unit flow out of the source, as opposed to the cost of the shortest path from source to sink. We show that with this modification, we can seamlessly apply the duality-based proof technique of [9]. Next, in order to improve the capacity violation factor to 1 + e, we introduce a subtle difference to edge costs in the residual graph: we set the cost of backward arcs to 0 in the residual graph instead of c_e as in [9]. Up to a factor of 2 in recourse, the two definitions are equivalent. However, this minor modification in defining the heights helps us perform a tighter analysis to get the improved factor.

**Load Balancing.** Inspired by the algorithm for online restricted assignment scheduling of [9], we follow a two-step procedure, where we first show how to maintain good fractional solutions with bounded makespan and bounded recourse, and then we devise a method which can round the fractional solution online, again with recourse comparable to that of the fractional solution. For the first step, we can in fact view the fractional load-balancing problem as an instance of the generalized flow problem, with the different p_{ij}-values acting like gain-factors. We can then immediately use Theorem 1.4 to maintain (1 + e)-competitive fractional solutions with O(1/e)-recourse. To round this solution, we devise an online low-recourse adaptation of the Shmoys-Tardos (ST) algorithm in [15] for the offline generalized assignment problem. Indeed, the ST algorithm crafts a bipartite matching instance based on a fractional solution x* to the GAP instance, such that (a) there exists a feasible matching in the resulting instance, and (b) any feasible matching can be mapped back to a 2-approximate schedule.

Our online rounding algorithm loosely follows the above outline, where jobs correspond to left vertices, and we create roughly as many right vertices for each machine as its fractional number of jobs assigned to it. If we try to strictly abide by the above construction, the bipartite matching instance could change substantially even when the fractional solution x* only changes by a bit. To overcome this, we maintain a two-level partition of the right-vertices associated with each machine i. They are first partitioned into buckets, each of which contain roughly Θ(1/e) units of fractional allocation (based on decreasing order of p_{ij} values), and then every bucket is further partitioned into segments, each of which contains 1 − Θ(e) units of fractional allocation. We update these buckets and segments not after every change to the fractional solution, but only when certain trigger events occur, and we use this to bound the total changes made to the bipartite matching instance, which in turn lets us bound the overall recourse of our schedule.

Our L(n) = O((log log n)/log log n)-competitive algorithm with O(1)-amortized recourse in Theorem 1.3 uses many ideas from the L(m)-competitive online rounding algorithm of [12]. The offline version of our algorithm works as follows. Let x_{ij} be the fractional extent to which job j is assigned to machine i. A job j is "heavy on machine i" if p_{ij} > T^*/log n and light otherwise, where T^* is the optimal offline makespan. Then a job is "big" if \sum_{i \in B(j)} x_{ij} ≥ 1/2 where B(j) is the set of machines on which job j is heavy. If job j is not big then we call it a "small" job. For small jobs, independent rounding already gives an O(1)-approximation ratio. For big jobs j, with first round x_{ij} values so that each x_{ij} becomes either 0 or at least 1/log n. This makes the support of x for big jobs sparse. Then we attempt to assign a big job to a machine randomly using the new x_{ij} values. The assignment fails if the target machine is too overloaded. Finally, we use the deterministic 2-approximation rounding algorithm to round the failed jobs. To analyze the recourse, we use two crucial lemmas: A job j fails with 1/poly log(n) probability, and with high probability, a connected component induced by failed jobs has size poly log(n). Therefore, even if we reassign everything in such a connected component, the recourse is small.

### 1.2 Related Work

It is well known that without any form of recourse, the best possible competitive ratio for online load balancing is O(log n) [6]. When arrivals and departures are allowed, [5] give a lower bound of O(\sqrt{n}). Philip and Westbrook [13] considered the same case and showed O(log n)-competitive algorithms with O(1)-recourse for the special case of unit-size restricted assignment (aka online matching). Westbrook [16] subsequently designed O(1)-competitive algorithms with O(log n)-recourse for the same setting. The case of unrelated machines in the fully-dynamic model was considered in [4] where the authors designed O(log n)-competitive algorithms with O(log n)-recourse. In the special case of identical machines, [10] gave a lower bound \sqrt{n} and an upper bound of 1.923 is known from [2]. [14] presented a class of (1 + e)-competitive algorithms with reassignments allowed, where the mitigation factor grows as e gets smaller. When there are only arrivals, but recourse is allowed, [9] showed that one can achieve a O(log log nm) competitive ratio with O(1)-amortized recourse in the restricted assignment setting. [7] extended this result to general re-assignment costs. Both papers also gave constant approximations with similar amount of reassignments in the special case of OGNF with all gains γ_e = 1.

There is also significant work on designing offline algorithms for generalized flow problems (see e.g., Chapter 6 of [17]), and these algorithms have also been used to design fast approximation algorithms for makespan minimization [8]. We believe our work is the first to address the online version of generalized flow.

**Organization.** In Section 2 we present our results for online generalized flow and prove Theorem 1.4. Using this algorithm, in Section 3 we design the (2 + e)-competitive rounding algorithm for load balancing to prove Theorem 1.2. In Section 4, we present the O((log log n)/log log n)-competitive algorithm for the same problem with O(1) amortized recourse, proving Theorem 1.3. In Appendix B we give the lower bound on the recourse required in the fully dynamic model. Missing proofs can be found in Appendix A.

**Notations** For a real number a, we use (a)_e to denote max{a, 0}. For a graph H and a vertex v in H, we let N_H(v) to denote the set of neighbors of v in H. If H is directed, we use δ^+_H(v) and δ^−_H(v) to denote the sets of outgoing and incoming edges of v in H respectively. When there is no ambiguity, we omit the subscript. For online load balancing, we shall identify jobs with time steps: jobs are denoted as [n], where job t ∈ [n] arrives at time t.
2 THE ONLINE GENERALIZED NETWORK FLOW (OGNF) PROBLEM

In the generalized network flow problem, we are given a digraph $G = (V, E)$ with sources $S \subseteq V$ and a sink $t \in V \setminus S$, where the sources $S$ do not have incoming edges and the sink $t$ does not have outgoing edges in $G$. We are given vectors $\mu, y \in \mathbb{R}^E_+ \in \mathbb{R}^E_+$ where for every $e \in E$, $\mu_e, y_e$ and $c_e$ denote the capacity, gain and cost of the edge $e$ respectively. Every source $s \in S$ has $a_s > 0$ units of supply. As in the ordinary network flow problem, our goal is for each $s \in S$ to send $a_s$ units flow in the network satisfying the flow conservation and edge capacity constraints, so as to minimize the cost. The generalization comes from the gain vector $y$: when a vertex $u$ sends $\theta$ unit flow along an edge $e = uv$, $\theta$ will receive $y_{uv}$ units flow from the edge. Therefore, the problem can be formulated as the following LP, where we assume $a_0 = 0$ for every $v \in V \setminus \{t\}$.

$$\min \sum_{e \in E} c_e x_e \quad \text{s.t.} \quad (1)$$

$$\sum_{e \in \delta^-(v)} x_e - \sum_{e \in \delta^+(v)} y_e x_e = a_v \quad \forall v \in V \setminus \{t\}$$

$$0 \leq x_e \leq \mu_e \quad \forall e \in E$$

In the LP, $x_e$ for an edge $e = uv$ in $E$ indicates the amount of flow sent by $u$ through $e$. Both the capacity $\mu_e$ and the per-unit cost $c_e$ are defined with respect to the flow on the sender’s side of $e$. We call any $x \in \mathbb{R}^E_+$ satisfying the constraints in the LP a valid flow for the instance. Notice that unlike the ordinary network flow problem, we cannot infer the flow into the sink $t$, and so we leave $a_t$ undefined and do not impose the flow conservation constraint on $t$.

**Online Generalized Network Flow Problem (OGNF).** We are initially given an instance of the problem $(G = (V, E), S, \tau, \mu, c, y)$ with $S = \emptyset$. In each time step $t = 1, 2, \ldots, T$, a new source $s_t \notin V$ arrives. We assume $a_{s_t} = 1$, as this suffices for our purpose. Along with $s_t$, we are given the outgoing edges of $s_t$, as well as their $\mu$, $c$ and $y$ values. After the arrival of $s_t$, we add it to $S$ and $V$, and its outgoing edges to $E$. Note that no source will ever have incoming edges. We need to maintain a valid flow $x^{(t)} \in \mathbb{R}^E_+$ for the instance at any time $t$. The cost incurred is defined as $\sum_{e \in E} c_e (x_e^{(t)} - x_e^{(t-1)})$, where we assume undefined variables have value $0$. Our goal is to design an online algorithm with a small cost.

We elaborate more on the definition of our cost: [9] defined the cost incurred at step $t$ to be $\sum_{e \in E} c_e |x_e^{(t)} - x_e^{(t-1)}|$ in their model to accurately capture the notion of recourse for flow problems. In this model, decreasing the flow value along an edge would incur a positive cost. This is in contrast to classical offline algorithms for flow where decreasing the flow value of an edge would incur a negative cost, thereby ensuring that the final cost of the flow would always be equal to the sum of costs incurred by each update. Our model is in between these models, where we charge flow increases with positive cost but omit charging flow decrements. However, note that our results also translate to results for the [9] cost model within a factor of 2 in the cost — for any decrement in the flow value on an edge, we must have paid the cost when we increase the flow value.

We prove the following main theorem in this section, which is a more formal description of Theorem 1.4:

**Theorem 2.1.** Given any $\varepsilon > 0$, there is an efficient deterministic algorithm for OGNF with capacities $\mu$, such that the following holds. The cost incurred by the algorithm at any time is at most $\frac{1 + \varepsilon}{\varepsilon} = O(\frac{1}{\varepsilon})$ times the cost of the optimum flow for the general network flow instance with capacities $\mu$.

Implicit in this theorem statement is the requirement that the given OGNF instance is feasible. Feasibility of the instance can be checked efficiently by verifying that LP (1) is feasible.

**Definition 2.2 (Residual Graph).** Let $(G, S, \tau, \mu, c, y)$ be a generalized network flow instance. Let $x \in \mathbb{R}^E_+$ be a vector such that $x_e \in [0, \mu_e]$ for every $e \in E$ (it is not required that $x$ is a valid instance). Then the residual graph $G^x = (V, E^x)$ for $x$ is defined as the graph containing vertices $V$ and the set $E^x$ of edges, each $e \in E^x$ with parameters $\mu^x_e, c^x_e$ and $y^x_e$. They are defined as follows.

- For every $uv \in E$ with $x_{uv} < \mu_{uv}$, we have a forward edge $uv \in E^x$ with $\mu^x_{uv} = \mu_{uv} - x_{uv}, c^x_{uv} = c_{uv}$ and $y^x_{uv} = y_{uv}$.
- For every $uv \in E$ with $x_{uv} > 0$, we have a backward edge $vu \in E^x$ with $\mu^x_{vu} = y_{uv}x_{uv}, c^x_{vu} = 0$ and $y^x_{vu} = 1/y_{uv}$.
- We call a cycle $C \in G^x$ flow absorbing if $\prod_{e \in C} y^x_e < 1$, flow generating if $\prod_{e \in C} y^x_e > 1$, and unit gain if $\prod_{e \in C} y^x_e = 1$.

The definition can be viewed as an extension of the residual graph for network flow without gains to the generalized network flow problem. We make two remarks here. First, a backward edge has cost 0 instead of a negative cost, due to the cost defined in our online model. Second, $\theta$ units of flow sent via $wo$ on the $u$’s side transforms into $y_{wu}0$ units of flow on the $v$’s side. This gives the definition of $\mu^x_{wu}$ and $y^x_{wu}$ for a backward edge $vu \in E^x$.

2.1 The Online Algorithm

Our algorithm is simple: whenever a new source $s_t$ arrives, we simply find the cheapest cost augmentation in the residual graph $G^{x^{(t-1)}}$. Formally, we solve LP (1) with $a_{s_t} = 1, a_0 = 0$ for all $v \notin \{s_t, t\}$ and costs, capacities, and gains as defined in Definition 2.2 for $G^{x^{(t-1)}}$, and let $f^t$ denote the computed flow. We then simply augment the existing flow with this optimal LP solution by setting $x^{(t)} = x^{(t-1)} + f^t$. It should be clear that $x^{(t)}$ is indeed feasible for LP (1) with $a_{s_t} = 1$ for $s_1 \ldots s_t$ and 0 for all non-sink vertices.

2.2 Analysis of the Online Algorithm

Before we get to the analysis, we define the useful concepts of (fractional) augmenting paths.

**Definition 2.3 (Fractional Augmenting Paths).** Let $(G, S, \tau, \mu, c, y)$, $x, G^x = (V, E^x)$ and vectors $\mu^x, y^x$ and $c^x$ be defined as in Definition 2.2. Let $s \in V \setminus \{t\}$ (it may be possible that $s \notin S$). A fractional augmenting path from $s$ in $G^x$ is a vector $f \in \mathbb{R}^E_+$ such that the excess flow $\sum_{e \in \delta^+(v)} f_e - \sum_{e \in \delta^-(v)} f_e$ equals 1 for $v = s$, and equals 0 for $v \in V \setminus \{s, t\}$. The cost of such an augmenting path $f$ is defined as cost$(f) := \sum_{e \in E^x} c^x_e f_e$.

Notice that the definition does not involve the capacities $\mu^x_e$ as long as $e$ exists in $E^x$, $f_e$ can take any number in $\mathbb{R}_{\geq 0}$.

**Definition 2.4 (Augmenting Paths).** Take all notations in Definition 2.3 and assume $f$ is a fractional augmenting path from $s$ in $G^x$. 
We simply say f is an augmenting path (without the word "fractional") if additionally it satisfies one of the following conditions:

(2.4a) The support of f is a path from s to t in $G^x$.
(2.4b) The support of f is a flow absorbing cycle C in $G^x$ containing s but not t.
(2.4c) The support of f is the union of a flow absorbing cycle C in $G^x$ not containing s and t, and a path in $G^x$ from s to C that is internally disjoint from C.

Lemma 2.5. Consider a generalized network flow instance $(G, S, T, \mu, c, \gamma)$ and $x \in \mathbb{R}^E_+$ satisfying $x_e \in [0, \mu_e]$ for every $e \in E$. Let $v \in V \setminus \{s, t\}$. Then f, the minimum-cost fractional augmenting path from v in $G^x$, assuming it exists, can be achieved at an augmenting path.

The proof of the lemma is deferred to the appendix.

To help with the analysis, we present a different algorithm which incrementally sends flow along the cheapest augmenting paths until we send unit flow out of the new source $s_t$, and we update the residual graph after each augmentation. The crucial difference between these versions is that the one-shot LP based approach essentially implicitly can undo mistakes of these individual cheapest augmentations, which the following algorithm in the analysis cannot exploit, due to backward arcs having zero cost as opposed to negative cost. However, this algorithm will simplify the proof substantially albeit incurring exponential running time.

Now given a (fractional) augmenting path f from s in $G^x$, and a real number $\theta > 0$, we define the operation of augmenting x by $\theta$ units using f as follows:

- For every forward edge uw $\in E^x$ with $f_{uw} > 0$, we update $x_{uw} \leftarrow x_{uw} + \theta \cdot f_{uw}$.
- For every backward edge uv $\in E^x$ with $f_{uv} > 0$, we update $x_{uv} \leftarrow x_{uv} - \theta \cdot f_{uv}/y_{uv}$.

Claim 2.6. Let f be an augmenting path from s in $G^x$. Then, augmenting x by $\theta > 0$ units using f does not change $\sum_{e \in \delta^+(s)} x_e - \sum_{e \in \delta^-(s)} y_e x_e$ for every $v \in V \setminus \{s, t\}$, and increases $\sum_{e \in \delta^+(s)} x_e - \sum_{e \in \delta^-(s)} y_e x_e$ by $\theta$.

We can now formally describe the alternate online algorithm useful for analysis in Algorithm 1.

**Algorithm 1** Online algorithm for generalized network flow

1. Let $x^{(0)}$ be the all-0 vector over edges of the initial graph $G$
2. for every $t \leftarrow 1$ to $T$ do
3. update the instance to include the source $s_t$
4. let $x \leftarrow x^{(t-1)}$, adding 0-coordinates for the incoming edges of $s_t$
5. while $\sum_{e \in \delta^+(s_t)} x_e < 1$ do
6. find the cheapest augmenting path f from $s_t$ in the residual graph $G^x$
7. let $\theta > 0$ be the biggest number such that after augmenting x using f by $\theta$ units, we still have $x_e \in [0, \mu_e]$ for every $e \in E$ and $\sum_{e \in \delta^+(s_t)} x_e \leq 1$
8. augment x using f by $\theta$ units
9. $x^{(t)} \leftarrow x$

Lemma 2.7. If $x^{(t-1)}$ is a feasible flow in LP (1) with the first $t-1$ sources only, then $x^{(t)}$ is a feasible flow in LP (1) with the first $t$ sources only.

Proof. By claim 2.6, augmenting x using cheapest augmenting paths f from $s_t$ does not change $\sum_{e \in \delta^+(s)} x_e - \sum_{e \in \delta^-(s)} y_e x_e$ for any $v \notin \{s, t\}$ and so, $x^{(t)}$ satisfies all excess flow constraints. Since $\theta$ is chosen at each step to ensure that $x_e \in [0, \mu_e]$, $x^{(t)}$ satisfies non-negativity and capacity constraints as well and is therefore feasible. □

[9] define the concept of height of any vertex to be the cost of the shortest path to the sink r in the residual graph at any point in time, and then track it for bounding the competitive ratio. Analogously, we define a height for a vertex as the cost of the cheapest augmenting path from the vertex in the residual graph – notice the lack of sink in the definition. We show that the incremental cost incurred by the ith source is at most the height of $s_t$ at time t. Moreover, we also show that heights can only increase during the course of the algorithm, implying that the cost incurred by the algorithm is at most the total height of sources at the end. To complete the analysis, we show that the heights (after all arrivals) define a dual solution for the final flow instance with capacities scaled down by a factor of $1 + \epsilon$, and upper bound its cost by $O(1/\epsilon)$ times the offline optimum cost of the instance.

Definition 2.8. Let $t \in [0, T]$ and $v \neq r$ be a vertex in the network $G$ at time t. The height of v at time t, denoted as height$_t(v)$, is defined as the cost of the cheapest augmenting path from v in the residual graph $G^x_t$. By Lemma 2.5, this is also the cost of the cheapest fractional augmenting path. We define height$_t(s_t) = 0$.

Monotonicity of Heights. We show heights are non-decreasing over the course of the algorithm. The proof the following lemma is deferred to Appendix A.

Lemma 2.9. Let $t \in [T]$ and $v$ be a vertex in the graph $G$ at time $t = 1$. Then height$_t(v) \geq$ height$_{t-1}(v)$.

Lemma 2.10. The cost incurred by the whole algorithm is at most $\sum_{t=1}^T$ height$_t(s_t)$.

Proof. Notice that the cost incurred by sending 1 unit of flow from the source $s_t$ in time t can be upper bounded by height$_t(s_t)$, which in turn is at most height$_T(s_t)$ by Lemma 2.9. □

Bounding total heights using duality. Let $C^*$ be the cost of the optimum flow for the generalized network flow instance at time T, with capacities scaled by a factor of $\frac{1}{1 + \epsilon}$. We consider the dual of LP (1):

$$\max \sum_{v \in V \setminus \{r\}} a_{uv} y_u - \sum_{e \in E} \frac{\mu_e x_e}{1 + \epsilon}$$

s.t.

$$-z_{uv} + y_u - \gamma_{uv} y_u \leq c_{uv} \quad \forall u \in E$$
$$z_e \geq 0 \quad \forall e \in E$$
$$y_r = 0$$

Notice that we do not have a constraint in LP (1) for r; for convenience we also introduce a dual variable $y_r$ and let $y_r = 0$. For a fixed y vector, the optimal choice for $z_{uv}$ is $(y_u - \gamma_{uv} y_u - c_{uv})$.
Also \(a_v = 0\) for every \(v \in V \setminus S \setminus \{\tau\}\) and \(a_v = 1\) for every \(t \in [T]\).
Therefore, the dual LP can be rewritten as
\[
\max \sum_{t \in [T]} y_t - \sum_{uv \in E} \frac{\mu_{uv}(y_u - y_v + c_{uv})}{1 + \varepsilon} \quad \text{s.t. } y_t = 0.
\]

**Lemma 2.11.** \(\sum_{t=1}^T \text{height}_T(s_t) \leq \frac{1 + \varepsilon}{\varepsilon} C^*\).

**Proof.** We bound the sum of the heights of sources at termination. To do this we show that \((y_t := \text{height}_T(s_t))_{t \in V}\) is a feasible dual solution. Then we bound the cost of this feasible dual in relation to \(C^*\), therefore giving us a bound on the competitive ratio.

Let \(x = x^{(T)} \in \mathbb{R}^E_+\) be the final flow we obtained. By breaking edges, we can assume every edge \(e \in E\) has either \(x_e = 0\) or \(x_e = \mu_e\). Any edge \(uv \in E\) with \(x_{uv} = 0\) exists in the residual graph \(G^\circ\). Therefore \(y_u - y_v + c_{uv}\) as \(y\) corresponds to the heights, and sending 1 unit flow from \(u\) can be achieved by sending 1 unit flow from \(u\) to \(v\), and then sending \(y_{uv}\) units flow from \(v\). Since for such edges \((y_u - y_v + c_{uv})\) is saturated i.e., \(y_u - y_v + c_{uv} = 0\). Let \(E'\) be the set of edges with \(x_{uv} = \mu_{uv}\). Then,
\[
\sum_{uv \in E} \frac{\mu_{uv}(y_u - y_v + c_{uv})}{1 + \varepsilon} \leq \sum_{uv \in E'} \frac{x_{uv}}{1 + \varepsilon} (y_u - y_v y_0) = \frac{1}{1 + \varepsilon} \sum_{e \in E'} y_e x_e = \frac{1}{1 + \varepsilon} \sum_{t \in [T]} y_{s_t}.
\]
The inequality holds because \(y_u > y_v + c_{uv}\) only occurs when the edge \(uv\) is not in the residual graph, which in turn only occurs when the edge \(uv\) is saturated i.e., \(uv \in E'\). We can drop the \(c_{uv}\) because \(c_{uv} \geq 0\). We can assume that For every \(uv \in E'\), \(x_{uv} = \mu_{uv}\) and \(y_0 \leq \frac{y_u}{\mu_{uv}}\) since the backward edge \(uv\) exists in \(G^\circ\) and it has gain \(\frac{1}{\mu_{uv}}\) and cost 0. So we have the inequality. The first equality is by that \(x_{uv} = 0\) for \(uv \in E \setminus E'\) and rearranging the terms. The second equality is by the balance condition for \(x\) and \(y_t = 0\).

So the objective value of the dual solution \(y\) is at least \((1 - \frac{1}{1 + \varepsilon}) \sum_{t \in [T]} y_t\), which implies \(\sum_{t \in [T]} y_{s_t} \leq C^*\). Multiplying both sides by \(\frac{1}{1 + \varepsilon}\) proves the lemma.

Thus, combining Lemmas 2.10 and 2.11 proves Theorem 2.1.

### 3 ONLINE UNRELATED MACHINE LOAD BALANCING WITH RECOURSE

We now turn our attention to one of our main results, that of maintaining \((2 + \varepsilon)\)-approximate solutions for online unrelated machine load balancing with \(O(\log n)\) amortized recourse per job, as stated in Theorem 1.2. We restate the problem setting. There is a set \(M\) of \(m\) machines, and \(n\) jobs indexed by \([n]\). We have a bipartite graph \(G = (M \cup [n], E)\) between machines and jobs, where \(ij \in E\) indicates that the job \(j\) can be assigned to machine \(i\). When \(j\) is assigned to \(i\), it incurs a load of \(p_{ij}\) on machine \(i\). The goal is to assign jobs to machines so as to minimize the maximum load over all machines, also called makespan in the scheduling literature. In the online version, jobs arrive one by one: job \(j \in [n]\) arrives at time \(j\), along with its incident edges in \(G\) and their \(p_{ij}\) values. We need to maintain a solution for the arrived jobs at any time. We allow the algorithm to re-assign prior jobs from time to time, and separately track the recourse of the algorithm.

**Known vs Unknown** \(T^*\). We first define a useful quantity \(T^*\), which is the smallest value of \(T\) for which the following LP is feasible: \(\sum_{j \in [n]} x_{ij} = 1\) for every \(j \in [t]\), \(\sum_{j \in [n]} p_{ij} x_{ij} \leq T\) for every \(i \in M\), and \(x_{ij} = 0\) if \(p_{ij} > T\). We refer to this as the optimal fractional makespan.

Our online algorithms will assume knowledge of \(T^*\). While we can use a standard guess-and-double approach to eliminate this assumption, we would lose an additional constant factor in the competitive ratio. Since we are allowed recourse, we can do better, as follows. Suppose there is algorithm \(A\) that achieves \(C : T^*\) makespan when \(T^*\) is given. We now design a simple procedure which can also achieve \((1 + O(\varepsilon))C\)-competitive solutions with bounded recourse, even when \(T^*\) is not given. We break our procedure into epochs, where a new epoch occurs when the optimum fractional makespan increases by a factor of at least \(1/\varepsilon\). Each epoch is further partitioned into many phases, when a new phase starts if the optimum fractional makespan increases by a factor of at least \(1 + \varepsilon\). When a new epoch \(g\) starts with bound \(T^*\) on the optimal fractional makespan, we simply re-construct an offline solution for all the jobs in \((g - 2)\)-th epoch with makespan at most \(2eT^*\), say using the 2-approximation algorithm [11] for offline load balancing. We then freeze the assignment of these jobs according to this offline solution, i.e., we won't change the assignment of these jobs in the future. Note that the total load due to all the frozen jobs on any machine is at most \(O(\varepsilon)T^*\). On the other hand, whenever a new phase starts with optimum fractional makespan \(T^*\), we re-run algorithm \(A\) with the revised estimate \(T^*\) and reintroduce all the unfrozen jobs (i.e., all jobs of the current and one previous epoch), thereby causing recourse for all these jobs. From the guarantee of \(A\), the makespan induced by these jobs will be at most \(C \cdot T^*\), giving us the desired guarantee. As for recourse, note that each job can be unfrozen for at most \(O(\log_4 e)\) phases across two epochs, which bounds the recourse.

**Overall Algorithm Overview.** We maintain a \((1 + \varepsilon)\)-competitive fractional solution with \(O(\frac{1}{\varepsilon})\)-amortized recourse using a reduction to online generalized network flow. We then round the fractional solution with low recourse by creating an intermediate bipartite matching instance based on the fractional solution.

#### 3.1 Maintaining Fractional Solutions Online

In the first step, we reduce the online load balancing problem to the online generalized network flow problem. We use \(G' = (V', E')\) to denote the digraph for the network flow problem. Initially, \(V' = M \cup \{\tau\}\), where \(\tau\) is the sink. There is an edge \((it) \in E'\) with \(\mu_{it} = 1 + \varepsilon\). \(c_{it} = 0\) and \(y_{it} = 1\). For each arriving job \(j\), we add \(j\) to \(V'\) and the source set. For every \(ij \in E\), we add a directed edge \(ji\) to \(E'\) with \(\mu_{ji} = \infty, c_{ji} = 1\) and \(y_{ji} = p_{ij}\).

**Lemma 3.1.** We can maintain fractional solutions \(x^{(t)} \in [n]\) online such that the following conditions hold:

- For every \(t \in [n]\), \(x^{(t)} \in [0, 1]^E\) is constructed at time \(t\), and is a fractional solution for jobs \([t]\) of makespan at most \((1 + \varepsilon)T^*\): \(\sum_j x_{ij} = 1\) for all jobs \(j \in [t]\), \(x_{ij} = 0\) for every \(ij \in E\) with \(j > t\), and \(\sum_j p_{ij} x_{ij} \leq (1 + \varepsilon)T^*\) for all \(i \in M\).

- The total fractional recourse until any time \(t\) is bounded, i.e., \(\sum_{t'=1}^t |x^{(t')} - x^{(t'-1)}| \leq O(\frac{1}{\varepsilon}) \cdot t\).
3.3 Online Rounding of Fractional Solutions

We now describe the online algorithm that rounds the fractional solutions $x^{(1)}, x^{(2)}, \ldots, x^{(t)}$ with a $(2 + O(\varepsilon))$-approximation ratio, which shall complete the proof of Theorem 1.2. To do this, we create an online bipartite matching instance $H = (L \cup R, E_H)$ with vertex updates on the right side. We will ensure that (a) the instance has sufficient expansion by exhibiting a fractional matching $y$ in $H$, (b) we can recover a good schedule for our original load balancing instance with only a small overhead in the objective function, and (c) the total number of vertex updates to $H$ is bounded in terms of the fractional recourse of the input $\sum_t |x^{(t)} - x^{(t-1)}|$.

In the graph $H$, the left vertices correspond to jobs, and each machine is associated with a set of right vertices, which will dynamically change over time based on the fractional solution. To this end, we fix a machine $i \in M$, and describe the parts of $H$ pertaining to the machine $i$. Many notations in the section depend on $i$, but for simplicity we omit $i$ when the context is clear.

Let $x = x^{(t)}$ be the fractional load balancing solution at $t$. We sort all the jobs $j \in N(i)$ in non-increasing order of $p_{ij}$ values, and let $N(i) = \{j_1, j_2, \ldots, j_{|N(i)|}\}$ be labeled such that $p_{ij_1} \geq p_{ij_2} \geq \cdots \geq p_{ij_{|N(i)|}}$. We now define three kinds of intervals which we will maintain over time, and which will help us construct the graph $H$. We map each job $j_k$ to a job interval $I_k$ of length $x_{k}$. $I_1$ is mapped to $I_1 \defeq [0, x_{j_1}]. J_2$ is mapped to $I_2 \defeq [x_{j_1}, x_{j_1} + x_{j_2}]. J_3$ is mapped to $I_3 \defeq [x_{j_1} + x_{j_2}, x_{j_1} + x_{j_2} + x_{j_3}].$ and so on. Let $X = \sum_{j \in N(i)} x_{j}$ be the fractional number of jobs assigned to $i$; that is, the total length of all job intervals. When $x$ gets updated, $X$ and the job intervals also change accordingly.

We next define a two-level partition of the interval $[0, X]$. In the first level, we partition $[0, X]$ into a set of bucket-intervals, or simply buckets. We guarantee that all buckets (except the last) have length between $\frac{1}{2}$ and $\frac{2}{3}$. In the second level, each bucket-interval is further partitioned into segment-intervals, or simply segments. All the segments except the last one in a bucket have length between $1 - 3\varepsilon$ and $1 - \varepsilon$, and the last segment length has at most $1 - \varepsilon$.

The 2-level partition of $X$ for $i$ determines the portion of the graph $H$ and the fractional matching vector $y$ for $i$. For every segment $s$, we have a vertex $x_{s} \in R$ for the segment. For every job $j_k \in N(i)$ such that $x_{j_k} \cap I_{o} \neq \emptyset$, we have $y_{j_k} x_{s} \in E_H$. Moreover, for analysis purposes, we define a fractional solution for the bipartite matching instance $y_{j_k} x_{s}$ to be the length of $x_{j_k} \cap I_{o}$.

Claim 3.4. At any time $t$, for every $j_k \in E_H$, we have

\[ \sum_{s} y_{j_k} x_{s} = x_{j_k}. \]

Proof. This holds since all the segments form a partition of $[0, X]$, and the job interval for $j_k$ is inside $[0, X]$. □

Therefore, if we consider the whole graph $H$, $y$ defines a fractional matching between $L = \{y\}$ and $R$ at time $t$: every $y \in R$ is covered to an extent of $1$ and every $v \in R$ is matched by an extent of at most $1 - \varepsilon$. This implies the following claim:

Claim 3.5. At any time $t$, we have $|N_{H}(A)| \geq |A| / 1+\varepsilon$ for every $A \subseteq L$.

\[ \frac{1}{\alpha} \]

We note one technicality here: all the intervals we defined are closed. So it is possible that the intersection of segment $I_k$ and $L_y$ is only one point. In this case we have $J_k x_{s} \in E_H$ and $y_{j_k} x_{s} = 0$. It is allowed for the bipartite matching algorithm to match $J_k$ to $x_{s}$.\[ \frac{1}{\alpha} \]
Dynamically Maintaining the Buckets and Segments. We now describe how to maintain the 2-level partition of $[0, X]$, as the fractional solution $x$ changes. Initially, $X = 0$ and there is one bucket and segment of 0-length at 0. We show how to handle two operations: increasing and decreasing some $x_{ij}$ value. The two operations are sufficient for our algorithm: when a new job $j$ with $ij \in E$ arrives, we create a 0-length job interval for the job at the appropriate position, and then we increase $x_{ij}$ from 0 to the desired number. The changes to $x$ can also be handled using the two operations.

First, consider the case where we need to increase $x_{ij}$, for some $o$. We find the first segment seg (from left to right) that internally intersects the job interval $I_o$. We then increase $x_{ij}$, and the lengths of $I_o$ and seg continuously at the same rate. As a result, the length of the bucket buc containing seg also increases. Also, the segments after seg and the buckets after buc will be shifted to the right continuously. We run the procedure until $x_{ij}$ is increased enough, or one of the following events happens.

- The length of seg reaches $1 - \epsilon$. In this case, we re-divide the bucket buc into segments: all the segments except the last segment in buc have length exactly $1 - 2\epsilon$, and the last bucket has length at most $1 - 2\epsilon$.
- The length of buc reaches $\frac{1}{2}$. In this case, we divide buc into two buckets of length $\frac{1}{2}$ each. Then we re-divide each of the two buckets into segments of length $1 - 2\epsilon$ as in the previous case. If we have not increased $x_{ij}$ enough after handling the event, we continue running the procedure.

Now suppose we need to decrease $x_{ij}$ by some amount. We find the first segment seg that intersects the job interval $I_o$ internally. Let buc be the bucket containing seg. Similarly, we decrease $x_{ij}$, and the lengths of $I_o$ and seg continuously at the same rate, until we decreased $x_{ij}$ enough, or seg does not intersect internally with $I_o$ anymore, or one of the following two events happen. As before, the segments after seg and the buckets after buc will be shifted to the left continuously.

- The length of seg drops to $1 - 3\epsilon$ and is not the last segment of buc. In this case, again we re-divide buc into segments of length $1 - 2\epsilon$ as before.
- The length of buc drops to $\frac{1}{2}$ and buc is not the only bucket. In this case, we merge buc with either its previous or the next bucket, depending on which one exists. If the merged bucket has length at most $\frac{1}{2}$, then we keep it. Otherwise, we divide the bucket into 2 equal-length buckets, each of which has length between $\frac{1+\epsilon}{2}$ and $\frac{2}{\epsilon}$. In any case, we divide each of the newly created buckets into segments of length $1 - 2\epsilon$.

Again, if we have not decreased $x_{ij}$ enough after handling the event, we continue the decreasing operation.

Completing the Rounding Algorithm. We maintain the dynamically-changing bipartite graph $G$ and use the bipartite-matching algorithm described in Section 3.2 and Theorem 3.2 to maintain a bipartite matching. If a job $j$ is assigned to a segment for machine $i$, we then assign job $j$ to machine $i$ in the load-balancing instance.

3.4 Analysis

We now analyze the recourse and competitive ratio of the algorithm.

### Analysis of Recourse

First, we analyze the recourse.

**Lemma 3.6.** The number of vertex updates on segments for $i$ by any time $t$ is at most $O\left(\frac{2}{\epsilon^2} \cdot \sum_{\ell=1}^{t} \sum_{j \in N(i)} |x_{ij}^{(\ell)} - x_{ij}^{(\ell-1)}| \right)$.

**Proof.** We fix the machine $i$ in this proof, and consider two different causes for vertex updates for machine $i$.

First, vertex updates may happen when a new bucket is created. When a new bucket is created, its length is between $\frac{1-\epsilon}{2}$ and $\frac{1}{2}$. When a bucket is created due to an increasing operation, then the new buckets have length $\frac{1-\epsilon}{2}$. Consider a decreasing operation. The merged bucket will have length between $\frac{1}{2}$ and $\frac{1}{2} - \epsilon$. If it has length at most $\frac{1}{3}$, then no splitting happens and its length is between $\frac{2}{3}$ and $\frac{1}{3}$. Otherwise, the two new buckets will have length between $\frac{1-\epsilon}{2}$ and $\frac{1}{2} - \epsilon$. So, it takes $\frac{2\epsilon}{2}$ fractional recourse on $(x_{ij})$ to create a new bucket. Moreover, when a new bucket is created, $O(1/\epsilon)$ segments will be inserted and deleted. Therefore, creation of new buckets incur at most $O\left(\frac{2}{\epsilon^2} \cdot \sum_{\ell=1}^{t} \sum_{j \in N(i)} |x_{ij}^{(\ell)} - x_{ij}^{(\ell-1)}| \right)$ vertex updates.

Then we consider vertex updates due to re-division of a bucket into segments. Whenever a re-division happens, all segments have lengths exactly $1 - 2\epsilon$ except for the last one, which has length at most $1 - 2\epsilon$. The next re-division happens if the length of some segment increases to $1 - \epsilon$, or the length of some segment that is not the last one decreases to $1 - 3\epsilon$. So, it takes $\epsilon$ fractional recourse on $(x_{ij})$ for the next re-division to happen. When a re-division happens, we make $O(1/\epsilon)$ vertex updates. Therefore, re-divisions incur at most $O\left(\frac{2}{\epsilon^2} \cdot \sum_{\ell=1}^{t} \sum_{j \in N(i)} |x_{ij}^{(\ell)} - x_{ij}^{(\ell-1)}| \right)$ vertex updates.

Notice that these are the only case where a vertex update happens. We need to mention that when the length of the intersection of seg and $I_o$ decreases to 0, no vertex update happens, since still there is an edge between $j_o$ and $\sigma_{seg}$ in $H$. $\square$

Now we can complete the analysis of the recourse of the online rounding algorithm. Note that the total recourse made by the load balancing algorithm is at most the number of reassignments made by the bipartite-matching algorithm. By Theorem 3.2 and Claim 3.5, the recourse of the latter by time $t$ is at most $O\left(\frac{2}{\epsilon^2} \cdot (t + \#(\text{vertex updates by } t)) \right)$. By Lemma 3.6 and summing up the bounds over all machines $i$, the number of vertex updates by time $t$ is at most $O\left(\frac{1}{\epsilon^2} \cdot \sum_{\ell=1}^{t} \sum_{j \in N(i)} |x_{ij}^{(\ell)} - x_{ij}^{(\ell-1)}| \right)$, which is at most $O\left(\frac{1}{\epsilon^2} \cdot \right)$ by Lemma 3.1. Therefore the total recourse by time $t$ made by the load balancing algorithm is at most $O\left(\frac{1}{\epsilon^2} \cdot \log \left(\frac{1}{\epsilon} \right) \right)$. Recall that the extra $O\left(\frac{\log (1/\epsilon)}{\epsilon^2} \right)$ factor comes from making the assumption that $T^*$ is known.

**Analysis of Competitive Ratio.** It remains for us to analyze the competitive ratio of the algorithm. Again we fix a machine $i$.

**Lemma 3.7.** The load of machine $i$ at any time is at most $(2 + O(\epsilon))T^*$.

**Proof.** Fix a time $t$ and let $x = x^{(t)}$. We now use $\varphi_{ik}$ to denote the $k$-th segment associated with machine $i$ from left to right, over all the buckets. Let $J_{ik}$ be the set of jobs in $N(i)$ whose job intervals internally intersect the segment $\varphi_{ik}$. at the time. We use $\pi_{ijk}$ to indicate if $j$ is assigned to $\varphi_{ik}$ in the solution for bipartite matching instance at time $t$.

Notice that all segments except for last segments of buckets have length between $1 - 3\epsilon$ and $1 - \epsilon$. The last segment of a bucket has
We elaborate more on the first inequality. For every segment of the first bucket can be upper bounded by 1/ε times the budget from the previous bucket, as the bucket has length at least 1/ε. So, if k is the first segment of any bucket, then the upper bound holds as the total value for the previous segment is at least \( \pi \). Notice every \( j \) in the summation has \( p_{ij} \). If \( k \) is not the first segment of any bucket, then the upper bound holds as the total value for the previous segment is at least 1-3ε. The \( p_{ij} \) for the first segment of the first bucket can be upper bounded by \( T^* \); the \( p_{ij} \) for the first segment of other buckets can be bounded by \( \epsilon \) times the budget from the previous bucket, as the bucket has length at least 1/\( \epsilon \).

The total load \( i \) receives at time \( t \) is then

\[
\sum_k \sum_{j \in J_k} \pi_{ijk} p_{ij} \leq \frac{1}{1-\epsilon} \sum_k \sum_{j \in J_k} y_{jk} \cdot \pi_{ijk} + T^* + \epsilon \cdot \sum_k \sum_{j \in J_k} y_{jk} \cdot \pi_{ijk} \\
= (1 + O(\epsilon)) \sum_k \sum_{j \in J_k} y_{jk} \cdot \pi_{ijk} + T^* \\
= (1 + O(\epsilon)) \sum_{j \in N(i)} x_{ij} p_{ij} + T^* \leq (1 + O(\epsilon)) T^* + T^* \\
= (2 + O(\epsilon)) T^*.
\]

We elaborate more on the first inequality. For every \( j \in J_k \) with \( \pi_{ijk} = 1 \), we try to upper bound \( p_{ij} \) by \( \frac{1}{1-\epsilon} \sum_j y_{jk} \cdot p_{ij} \). Notice every \( j \) in the summation has \( p_{ij} \). If \( k \) is not the first segment of any bucket, then the upper bound holds as the total value for the previous segment is at least 1-3ε. The \( p_{ij} \) for the first segment of the first bucket can be upper bounded by \( T^* \); the \( p_{ij} \) for the first segment of other buckets can be bounded by \( \epsilon \) times the budget from the previous bucket, as the bucket has length at least 1/\( \epsilon \).

The total load \( i \) receives at time \( t \) is then

\[
\sum_k \sum_{j \in J_k} \pi_{ijk} p_{ij} \leq \frac{1}{1-\epsilon} \sum_k \sum_{j \in J_k} y_{jk} \cdot \pi_{ijk} + T^* + \epsilon \cdot \sum_k \sum_{j \in J_k} y_{jk} \cdot \pi_{ijk} \\
= (1 + O(\epsilon)) \sum_k \sum_{j \in J_k} y_{jk} \cdot \pi_{ijk} + T^* \\
= (1 + O(\epsilon)) \sum_{j \in N(i)} x_{ij} p_{ij} + T^* \leq (1 + O(\epsilon)) T^* + T^* \\
= (2 + O(\epsilon)) T^*.
\]

We elaborate more on the first inequality. For every \( j \in J_k \) with \( \pi_{ijk} = 1 \), we try to upper bound \( p_{ij} \) by \( \frac{1}{1-\epsilon} \sum_j y_{jk} \cdot p_{ij} \). Notice every \( j \) in the summation has \( p_{ij} \). If \( k \) is not the first segment of any bucket, then the upper bound holds as the total value for the previous segment is at least 1-3ε. The \( p_{ij} \) for the first segment of the first bucket can be upper bounded by \( T^* \); the \( p_{ij} \) for the first segment of other buckets can be bounded by \( \epsilon \) times the budget from the previous bucket, as the bucket has length at least 1/\( \epsilon \).

The last equality is by Claim 3.4. The last inequality comes from that the total fractional load on machine \( i \) is at most \( (1 + \epsilon)T^* \).

**4 AN \( O \left( \frac{\log n}{\log \log n} \right) \)-COMPETITIVE ALGORITHM WITH O(1)-AMORTIZED RECURSE**

In this section, we describe the \( O \left( \frac{\log n}{\log \log n} \right) \)-competitive algorithm for unrelated machine load balancing (OLBwR) with O(1)-amortized recourse. Notice that we can, compute fractional solution using the algorithm in Section 2 with \( \epsilon = 1 \). This only loses an \( O(1) \) factor on the competitive ratio. Using Lemma 3.1, we construct a sequence \( x^{(1)}, x^{(2)}, \ldots, x^{(n)} \) of fractional solutions online, each \( x^{(t)} \) being a fractional schedule for jobs \( \{t\} \). We assume the makespans of the fractional solutions are at most \( T^* \) by scaling up \( T^* \). For every \( t \in [n] \), we have \( \sum_{t'=1}^{t-1} |x^{(t')} - x^{(t-1)}| = O(1) \cdot t \). We prove the following theorem, which in turn proves Theorem 1.2.

**THEOREM 4.1.** There is a randomized online rounding algorithm that succeeds with high probability. Conditioned on its success, the makespan of the schedule at any time is \( O \left( \frac{\log n}{\log \log n} \right) \cdot T^* \) and the expected recourse by any time \( t \) is \( O(1) \cdot t \).

For ease of exposition, for every time step, we modify the fractional solution returned by the flow-based algorithm by adding a dummy machine \( i_L \) to \( M \) and assigning all jobs that have not arrived yet to \( i_L \) in both fractional schedule and output integral schedule. That is, we assume \( x_{i_L,j}^{(t)} = 1 \) for every \( j > t \). After this transformation, every \( x^{(t)} \) is a schedule for the whole job set \( [n] \). We assume all jobs have processing time 0 on the dummy machine; this will not create an issue as our algorithm will never assign a job \( j \) to a machine \( i \) at time \( t \) if \( x_{ij}^{(t)} = 0 \).

**4.1 Main Ideas**

Our algorithm uses the framework of the \( O \left( \frac{\log m}{\log \log m} \right) \)-competitive online rounding algorithm of [12]. In their setting, the fractional assignment of a job \( j \) never changes after its arrival. As a result, their algorithm does not need to incur a recourse.

We give a high-level overview of the rounding algorithm in [12]. They describe the algorithm in the offline setting, and one can easily make it online. We say an edge \( ij \in E \) is big if \( p_{ij} > \frac{T}{\log m} \), and small otherwise. A job \( j \) is big if at least 1/2 fraction of the job is assigned via big edges in the fractional solution, and small otherwise. For big (small) jobs \( j \), we only consider its heavy (light) edges. Small jobs can be assigned by independent rounding; with high probability they incur only an \( O(1) \cdot T^* \) load on every machine. Therefore we turn our attention to big jobs, which are assigned in three steps:

- **Step (b1):** The algorithm does an initial rounding to make the support of \( x \) sparse: If some \( x_{ij} \) for a big job \( j \) has \( x_{ij} \in (0, \frac{1}{\log m}) \), then it rounds \( x_{ij} \) to 0 or \( \frac{1}{\log m} \) randomly, preserving the expectation of \( x_{ij} \). After this step, every such \( x_{ij} \) is either 0 or at least \( \frac{1}{\log m} \). This guarantees that the support graph for \( x \) restricted to big jobs have degree \( O(\log^2 m) \).
- **Step (b2):** The algorithm attempts to assign every big job \( j \) to a machine \( i \) randomly, using the new \( x_{ij} \) values as probabilities. The assignment fails if the target machine is overloaded.
- **Step (b3):** The crucial theorem proved in [12] is that the following event happens with high probability: In the sub-graph of the support of \( x \) induced by the failed jobs and all machines, every connected component has size at most poly \( \log m \). This allows the algorithm to apply a deterministic \( O \left( \frac{\log m}{\log \log m} \right) \)-competitive online rounding procedure for each component.

We use a similar framework as that of [12], with the following main differences. First, we generate a set of global random seeds that are used in our randomized rounding procedure for each time step. They will correlate schedules at different time steps. Second, in step (b3), we can run the simple offline 2-approximation algorithm for each connected component, as recourse is allowed in our setting. Finally, a small difference is that our competitive ratio is \( O \left( \frac{\log n}{\log \log n} \right) \) as we need to apply union bounds over \( n \) time steps.

**4.2 Description of Algorithm and Proof of Competitive Ratio**

We now formally describe our algorithm. We say an edge \( ij \in E \) is heavy if \( p_{ij} \geq \frac{T}{\log n} \), and light otherwise. For every \( j \in [n] \), we let \( M_{j}^{\text{heavy}} \) (resp.) be the set of machines \( i \in N(j) \) with \( ij \) being heavy (light resp.).

**Generating Global Random Seeds.** We choose a threshold \( \beta \in \left[ \frac{1}{2}, \frac{3}{4} \right] \) uniformly at random to define big and small jobs.

**DEFINITION 4.2.** Given \( \beta \) and a fractional solution \( x \in [0,1]^E \), we say a job \( j \) scheduled in \( x \) is big if \( \sum_{i \in M_{j}^{\text{heavy}}} x_{ij} > \beta \) and small otherwise. So, if \( j \) is small, then \( \sum_{i \in M_{j}^{\text{light}}} x_{ij} \geq 1 - \beta \). Let \( j_{\text{big}} \) and \( j_{\text{small}} \) be the sets of big and small jobs respectively.

When a job \( j \) arrives, for every heavy edge \( ij \in E \), we independently choose a threshold \( \delta_{ij} \in [0,1/\log n] \) uniformly at random; this will be used in the initial rounding step (step (b1)) for big jobs. We also generate an infinite sequence of pairs \((h_{i}^{1}, \theta_{i}^{1}), (h_{i}^{2}, \theta_{i}^{2}), \ldots, \)
where for every $o \in \mathbb{Z}_{>0}$, $h^o_j$ is a random machine in $N(j)$, and $\theta^o_j$ is a random real number in $[0, 1]$; all the parameters are independently generated. The sequence will serve as the random seeds for the acceptance-rejection sampling method, to assign small jobs, and to assign big jobs in step (b2).

Once we generated the global random seeds, it is convenient to describe the algorithm in the offline setting: For every time step $t$, we round the fractional solution $x^{(t)}$ to obtain an integral assignment, using the global random seeds. A recourse occurs when the assignment of a job $j$ at time $t$ is different from that at time $t - 1$. For the rest of this subsection we focus on a single time step $t$, and the fractional solution $x := x^{(t)} \in [0, 1]^E$. Big and small jobs are defined w.r.t the global seed $\beta$ and this fractional solution $x$.

**Step (s): Assigning Small Jobs.** For every $j \in j_{\text{small}}$, we find the smallest $o \in \mathbb{Z}_{>0}$ such that $h^o_j \in M^\text{light}_j$ and $\theta^o_j \leq x^j h^o_j$, and we assign $j$ to $h^o_j$. This is equivalent to the following procedure. We draw a histogram for job $\theta E$ we assign $j$ to $h^o_j$. This is equivalent to the following procedure.

We say the vector $x^{(t)}$ is an approximate LP solution, where by Lemma 4.4 the two constraints are each violated by a factor of 10.

This finishes the description of the algorithm. With high probability, the algorithm maintains a schedule of makespan at most $O\left(\frac{\log \log n}{\log \log \log n}\right) \cdot T^*$ at any time: Small jobs, successful and failed big jobs respectively incur a load of $O(1) \cdot T^*$, $O\left(\frac{\log \log n}{\log \log \log n}\right) \cdot T^*$ and $O(1) \cdot T^*$ on each machine.

### 4.3 Bounding the Recourse

In this section, we bound the recourse of the algorithm case by case. Below we fix a time step $t \geq 1$, and use $x^0$ and $x$ to denote the fractional solution at time $t - 1$ and $t$ respectively. For every job $j$, we let $x^j_{\text{big}}$ denote the vector $(x^j_{i,j})_{i \in N(j)}$, i.e., the fractional assignment of $j$ at time $t$. Define $x^j_{\text{big}}$ similarly for time $t - 1$. Define $x^0, x^t, x^0_{\text{big}}, x^t_{\text{big}}$ similarly for the vector $x^t \in [0,1]^E$ obtained in step (b2). In all the proofs, we assume $c$ is a sufficiently large constant.

**Type-1 recourse:** recourse from switches between small and big jobs. We say $j$ incurs a type-1 recourse at time $t$ if $j$ is small at time $t - 1$, and big at time $t$, or the other way around.

**Lemma 4.5.** Let $j \in [n]$. In expectation over the randomness of $\beta$, the probability that $j$ incurs a type-1 recourse at time $t$ is at most $2 \cdot |x_j - x^j_{\text{big}}|/|x_j - x^j_{\text{big}}|$. 

### Proof.

Let $p^0_{\text{big}}$ be the fraction of job $j$ assigned via heavy edges, in the fractional solution $x^0$; define $p_{\text{big}}$ similarly for $x$. Then we have $|p_{\text{big}} - p^0_{\text{big}}| \leq 2 |x_j - x^j_{\text{big}}|$. If there is a switch between $j$ being small and $j$ being big, then $\beta$ must be between $p_{\text{big}}$ and $p^0_{\text{big}}$. This happens with probability at most $4 \cdot |p_{\text{big}} - p^0_{\text{big}}| \leq 2 \cdot |x_j - x^j_{\text{big}}|$ since $\beta$ is uniformly chosen from $[\frac{1}{2}, \frac{3}{4}]$.

**Type-2 recourse:** recourse from small jobs. We say a recourse incurred by $j$ at time $t$ is of type-2 if $j$ is small at both time steps $t - 1$ and $t$, but assigned to different machines in the two time steps.

---

There is a slight difference between our algorithm and that of [12] in this step. In [12], for a marked machine $i$, the jobs assigned to $i$ before it is marked are not failed. This is needed for their no-recourse setting.
Lemma 4.6. For every $j \in [n]$, the probability that $j$ incurs a type-2 recourse at time $t$ is at most $4 \cdot |x_j - x_j'|$.

Proof. We fix a $\beta$ for which $j$ is small in both time steps $t - 1$ and $t$. Consider the point $(h_i', \theta_i')$ from the random sequence for $j$. We say the point is good if it is accepted in both time step $t - 1$ and $t$; we say the point is bad if it is accepted only in exactly one of the two time steps. The point is neutral if it is not accepted in either time step. The probability that $j$ incurs a type-2 recourse at time $t$ is at most $\Pr[(h_i', \theta_i') \text{is bad}] \leq \frac{|x_j - x_j'|^2/N(j)}{t/4(N(j))} = 4 \cdot |x_j - x_j'|$. De-conditioning on $\beta$ gives the lemma. □

Type-3 recourse: recourse from different target machines in step (b2). We say a recourse incurred by $j$ at time $t$ is of type-3 if $j$ is big at both time steps $t - 1$ and $t$, but it is temporarily assigned to different machines in step (b2) of the two time steps.

Lemma 4.7. For every $j \in [n]$, the probability that $j$ incurs a type-3 recourse at time $t$ is at most $10 \cdot |x_j - x_j'|$.

Proof. First we condition on the value of $\beta$ such that $j$ is big at both time steps $t - 1$ and $t$. Conditioned on $\delta_j$ values, the probability that $j$ incurs a type-3 recourse at time $t$ is at most $10 \cdot |x_j' - x_j''|$ using the same argument from the previous lemma. De-conditioning on $\delta_j$ values, the probability is at most $10 \cdot |x_j - x_j'|$; this holds since $E_h_{ij} [\delta_j | x_j' - x_j''|] = |x_j - x_j'|$ for some $i \in M_j^{\text{heavy}}$. Finally, de-conditioning on $\beta$ gives the lemma. □

Type-4 recourse: recourse from failed jobs. We say a recourse incurred by a job $j$ at time $t$ is of type-4, if it is not of type-1, 2 or 3. In this case, $j$ is big in both time steps $t - 1$ and $t$, and we temporarily assign the same machine in both steps (b2), and it fails in at least one of the two time steps. Moreover, the connected component in $G'[M \cup J^{\text{fail}}]$ containing $j$ is not the same in the two time steps (we assume the condition holds if $j$ does not fail in the other time step).

Let $C$ be the set of connected components in $G'[M \cup J^{\text{fail}}]$ in time step $t$; define $C^o$ similarly for time step $t - 1$. So we need to count the number of jobs in the components in $C^o \setminus C$, where $C^o \cap C$ denotes the symmetric difference between $C^o$ and $C$, i.e., the set of components appearing exactly one of $C^o$ and $C$. Due to the symmetry, it suffices to consider the components in $C^o \setminus C$.

The key theorem we use is that any $C \in C^o$ is small with high probability. This is proved in [12] and used to bound their competitive ratio; while in our case, we apply the theorem to bound the recourse. The crucial properties we use to prove this are the facts that $G^o$ has degree $O(\log^2(n))$, and that every machine is marked with probability $1/\poly log(n)$, with a sufficiently large exponent in the poly log(n) factor. We omit its proof here as it is identical to that in [12], except for minor differences in parameters. See Theorem 4.5 in [12].

Theorem 4.8. With probability at least $1 - \frac{1}{n^7}$, every $C \in C^o$ contains at most $O(\log^7(n))$ machines.

With the theorem, we can now focus on bounding $|C^o \setminus C|$. The change of components from $C^o$ to $C$ are caused by the following two types of events:

- Some job $j$ has changed its target machine in step (b2) from time $t - 1$ to $t$, and in at least one of the two time steps, the target machine is marked; we also say this event happens if $j$ is small in the other time step. This event may add/remove $j$ to/from $J^{\text{fail}}$, switch up to 2 machines between marked and unmarked from time $t - 1$ to $t$. We say this is a type-a event incurred by job $j$.
- For some $j$ that fails in both time steps $t - 1$ and $t$, and some $i \in M_j^{\text{heavy}}$, we have $x_{ij}' \neq x_{ij}''$. We say this is a type-b event incurred by the pair $ij$. This will add/remove the edge $ij$ to/from the graph $G'$. Notice that if no events of the two types occur, then $C^o = C$, as marked machines, failed jobs, and $x_j$ values incident to failed jobs do not change from time $t - 1$ to time $t$.

Lemma 4.9. Fix a job $j \in [n]$. The probability that $j$ incurs a type-a event at time $t$ is at most $O\left(\frac{1}{\log^4(n)} \cdot |x_j - x_j'|\right).

Proof. The probability that $j$ is assigned to two different target machines in step (b2) in time steps $t - 1$ and $t$ is $O(1 \cdot |x_j - x_j'|).$ Conditioned on this event, the probability that the target machine is marked in the correspondent time step is at most $O\left(\frac{1}{\log^4(n)}\right)$ when $c$ is big enough. □

Lemma 4.10. Fix a heavy edge $ij \in E$. The probability that $ij$ incurs a type-b event is at most $O\left(\frac{1}{\log^4(n)} \cdot |x_{ij} - x_{ij}'|\right)$.

Proof. We say a job $j$ at time $t$ is heavy if $|x_{ij} - x_{ij}'|$. The probability that $j$ incurs a type-b event is at most $O\left(\frac{1}{\log^4(n)} \cdot |x_{ij} - x_{ij}'|\right)$.

Each type-a event can change at most $O(\log^2(n))$ components in $C^o$: it may change the marking status of two machines, and a machine is incident to $O(\log^2(n))$ jobs in the support of $x^o$. Each type-b event can change at most at most 2 components in $C^o$. Therefore, the expected number of components in $C^o \setminus C$ is at most $\sum_{j \in [n]} O\left(\frac{1}{\log^4(n)} \cdot |x_j - x_j'| \right) \cdot O(\log^2(n)) = O\left(\frac{1}{\log^4(n)} \cdot \sum_{j \in [n]} |x_j - x_j'| \right)$.

By Theorem 4.8, in expectation, the type-4 recourse is at most $O\left(\frac{1}{\log^7(n)} \cdot \sum_{j \in [n]} |x_j - x_j'| \right) \cdot O(\log^7(n)) \cdot O(\log^2(n)) = O\left(\frac{1}{\log^{14}(n)} \cdot \sum_{j \in [n]} |x_j - x_j'| \right)$.

Therefore, we have proved that the expected recourse time at time $t$ is at most $O(1) \sum_{j \in [n]} |x_j - x_j'| = |x - x'|$. Summing up the bound over all time steps $t'$ from 1 to $t$, we obtain that the recourse by time $t$ is at most $\sum_{t'=1}^t |x(t') - x(t'-1)| = O(1) \cdot t$. This finishes the proof of Theorem 4.1.

A MISSING PROOFS

Lemma 2.5. Consider a generalized network flow instance $(G, c, \mu, c, \gamma, \rho)$, and $x \in \mathbb{R}_{\geq 0}$ satisfying $x_x \in [0, \mu_c]$ for every $e \in E$. Let $\nu \in V \setminus \{e\}$. Then $\rho$, the minimum-cost fractional augmenting path from $\nu$ in $G^o$, assuming it exists, can be achieved at an augmenting path.

Proof. We consider any solution $f$ to the linear system defined by the constraints in Definition 2.3, with objective cost $(f)$. As all edge costs are non-negative, we can assume the optimum solution.
is achieved when all values in $f$ are bounded. Therefore, it must be achieved at a vertex point defined by some tight inequalities in the linear system. Let $G' = (V', E')$ be the subgraph of $G^s$ containing the support of $f$, and the vertices incident to these edges. We can assume $G'$ contains $s$ and is connected. Every $v \in V' \setminus \{s, x\}$ has at least one incoming edge and one outgoing edge; $s$ has at least one outgoing edge. As $f$ is a vertex solution, we have $|E'| \leq |V' \setminus \{t\}|$. We now consider two cases depending on whether $t \in V'$ or not.

First assume $t \in V'$. Then we have $|E'| \leq |V'|-1$. Therefore, we have $|E'| = |V'| - 1$ and $G'$ is a path from $s$ to $t$ in $G^s$ (case (2.4a)). It remains to consider the case when $t \notin V'$, which implies $|E'| \leq |V'|$. It can only happen that $|E'| = |V'|$. Since every vertex in $V'$ except $s$ has at least one incoming and one outgoing edge. Two sub-cases may happen in this case. It may be that $s$ also has an incoming edge in $G'$, in which case $G'$ is a cycle containing $s$ (case (2.4b)). It may also be that $s$ does not have an incoming edge in $G'$, but some other vertex in $G'$ has 2 incoming edges. In this case, $G'$ contains a cycle and a path connecting $s$ to the cycle (case (2.4c)). This finishes the proof of the lemma. □

The following claim will be useful in the proof of lemma 2.9.

Claim A.1. Let $f$ be an augmenting path from some $s \in V \setminus \{t\}$ in $G^s$ for some $x$. Let $v \in \{s, t\}$ be some vertex in the support graph of $f$. Then we can break $f$ into $f = f' + f''$ where $f'$ is a flow path in $G^s$ that sends flow from $s$ to $t$. The amount of flow leaving $s$ is 1 unit. However, the flow received by $v$ may not be 1. The flow $f''$ is a scaled augmenting path from $v$ in $G^s$.

Lemma 2.9. Let $t \in [T]$ and $v$ be a vertex in the graph $G$ at time $t-1$. Then $\text{height}(v) \geq \text{height}_{t-1}(v)$.

Proof. We will show a slightly stronger result than this. We will prove that after each iteration of the while loop, no nodes’ height decreases. To prove this, we proceed in three steps. First, we will show that the height of the source $s_t$ only increases. Second, we will show that the height of any node that received flow in the previous iteration cannot have decreased. Third, we will show that the height of any node that didn’t receive flow in the previous iteration could not have decreased.

Source height: Let $x$ be the flow before some iteration of the while loop and $\hat{x}$ be the flow after the iteration. Let $\delta = x - \hat{x}$. Suppose for contradiction that $\text{height}_x(s_t) < \text{height}_{\hat{x}}(s_t)$, i.e., the height of $s_t$ when the current flow is $x$ is smaller than the height of $s_t$ when the current flow is $\hat{x}$. Let $\delta_2$ be the flow of $e$ units from $s_t$ with cost $e\text{height}_{e}(s_t)$. Let $\theta$ be the net outflow from $s_t$ in $\delta$. Then $\delta' = \frac{\delta}{\theta} (\delta + \delta_2)$ must have cost $\frac{\delta}{\theta} (\text{height}_x(s_t) + \text{height}_{e}(s_t))$.

Let $s = s_t$ be the source arrived at time $t$. Notice that adding $s$ and its outgoing edges to $G$ does not decrease the cost of cheapest flow of $e$ units from $v$ in $G^s$, since $s$ does not have incoming edges. Let $\hat{x}$ be the flow at the beginning of some iteration of the while loop, at time $t$. Let $f$ be the cheapest augmenting path from $s$ in the residual graph $G^s$, as in Step 6. Let $\hat{x}$ be the flow obtained at the end of the iteration, i.e., after it is augmented using $f$. Let $f'$ and $f''$ be the cheapest augmenting paths from $v$ in the residual graph $G^s$ and $G^s$ respectively. Let $C_1 = \text{cost}(f')$ and $C_2 = \text{cost}(f'')$. It suffices for us to prove $C_2 \geq C_1$.

We first consider the case where $v$ is in the support graph $G'$ of the flow $f$, which falls in one of the three cases in Definition 2.4. Towards contradiction we assume $C_2 < C_1$. By Claim A.1, we can find an augmenting path $f''$ from $v$ in $G^s$, whose support is a subgraph of $G'$. As $f''$ is an augmenting path from $v$ in $G^s$, we have $\text{cost}(f'') > \text{cost}(f') = C_1$ by the choice of $f''$.

Let $\theta > 0$ be a sufficiently small real number. We define $f'' := f - \theta f' + \theta f''$, extending the domain of the three vectors by adding 0-coordinates if necessary. When $\theta$ is small enough, all entries in $f''$ are non-negative. So, the cost of $f''$ is strictly smaller than that of $f$ as $\text{cost}(f'') < C_1 = C_2 = \text{cost}(f')$. Clearly, $f''$ satisfies the balance constraints: the net flow sent by $s$ in $f''$ is 1, and the net flow sent by any vertex in $V \setminus \{s, r\}$ is 0.

However, some edges in the domain for $f''$ may be outside $E^s$. This issue can be handled as follows. Suppose some $e = uv$ with $e^2 > 0$ is not in $E^s$. Then it must be the case that $uv \in E^s$ and $f_{uv} > 0$. Then in $f''$, we update $f_{uv} := f_{uv} - \theta f_{uv} - \theta f_{uv}/\theta$ and discard the coordinate. When $\theta > 0$ is sufficiently small, we guarantee that all entries in $f''$ are non-negative. Moreover, this update operation can only decrease the cost of $f''$, and thus we still have $\text{cost}(f'') < \text{cost}(f)$. As the updated $f''$ is a fractional augmenting path from $v$ in $G^s$, this leads to a contradiction to the choice of $f$.

It remains to consider the case where $v$ is not in the support graph $G'$ of $f$. We consider first vertex $v'$ along the path $f''$ that is in $G'$. (In case $f''$ belongs to case (2.4b) or (2.4c), the path can be obtained by starting from $v$ and following out-going edges in the support of $f''$.) Using Claim A.1, we can break $f''$ into a flow path $\tilde{f}$ sending 1 units flow from $v$ to $v'$ and a scaled augmenting path $f''$ from $v'$ in $G^s$. We already proved that the cost of the cheapest augmenting path from $v'$ in $G^s$ is at most that in $G^s$. So, we can replace $f''$ with the cheapest augmenting path from $v$ in $G^s$, scaled by the same factor. Notice that $\tilde{f}$ is also a path in $G^s$ as $f''$ does not involve any vertex before $v'$. Therefore, we obtained an augmenting path from $v$ in $G^s$ with cost no larger than that of $f''$. Therefore, we have $C_2 \geq C_1$. □

Theorem 1.5. There is an efficient deterministic algorithm for ObMwRC which (i) maintains a matching of all the left vertices, (ii) ensures that each $v \in R$ is matched at most $\lfloor (1 + \epsilon) b_v \rfloor$ times, and (iii) has $O\left(\frac{1}{\epsilon} \right)$ amortized recource.

Proof. We reduce ObMwRC to the online generalized network flow problem (indeed, the standard network flow problem suffices as all gains will be unit). We maintain a digraph $G' = (L \cup R \cup \{t, E\})$. We have edges from $L$ to $R$ in $G'$ that are the same as those in $G$ (except that the edges in $G'$ are directed), and edges from each $v \in R$ to $t$. An edge $uv \in E'$ with $u \in L, v \in R$ has cost $c_{uv},$ capacity $c$ and gain 1. An edge $vr$ for $v \in R$ has cost 0, capacity $c(1 + \epsilon) b_v$ and gain 1. The sources in the network are $L$. The graph $G'$ is constructed online: when a new vertex in $L$ arrives, we add it and its outgoing edges to $G'$. Theorem 2.1 gives an online algorithm that maintains a network flow where every $e \in L$ sends 1 units of flow. As the edges have gain parameters $1$, $r$ receives $|L|$ units of flow. As the algorithm are based on augmenting paths, the flow is integral. Thus the algorithm
maintains a matching where every $v \in R$ is matched at most $\left\lceil \frac{1}{\alpha} \right\rceil$ times. The reassignment cost of the algorithm is equal to the cost incurred by the network flow algorithm, which is at most $\frac{1}{\alpha} \times$ times the optimum cost $C^*$. Hence, we get $C^* \leq \sum_{v \in L} c_v$.

**Lemma 3.3.** Let $H = (L \cup R, E_H)$ be a bipartite graph. Let $\alpha > 1$ be a real number such that $|N_H(A)| \geq \alpha |A|$ for every $A \subseteq L$. Let $F \subseteq E_H$ be a partial matching where not all vertices in $L$ are matched. Then there is an augmenting path of length at most $2D + 1$ w.r.t. $F$, where $D = \left\lfloor \log_2 |L| \right\rfloor + 1$.

**Proof.** Let $\overline{H}$ be the residual graph of $H$ w.r.t. $F$; $\overline{H}$ is a directed graph over $L \cup R$, for every edge $w \in E_H$, we have $w \in \overline{H}$, and for every $w \in F$, we have $w \in \overline{H}$. We say a vertex in $L$ is free if it is not unmatched in $F$. For every integer $d \in [0, D]$, define $L^d$ ($R^d$ resp.) to be the set of vertices in $L$ ($R$, resp.) to which there exists a path in $\overline{H}$ of length at most $2d$ ($2d + 1$, resp.) from a free vertex. So, we have $L^0 \subseteq L^1 \subseteq L^2 \subseteq \cdots \subseteq L^D$ and $R^0 \subseteq R^1 \subseteq R^2 \subseteq \cdots \subseteq R^D$. Also notice that $R^d = N_H(L^d)$ for every $d \in [0, D]$.

For every $d \in [0, D]$, we have $|L^d| \leq |R^d|$ by the condition of the lemma. All vertices in $R^D$ are saturated by our assumption that there are no augmenting paths of length at most $2D + 1$. So for every $d \in [0, L - 1]$, we have $|R^d| \leq |L^{d+1}|$ as all vertices in $R^d$ are matched by vertices in $L^{d+1}$.

Combining the two statements gives us $a|L^d| \leq |L^{d+1}|$ for every $d \in [0, L - 1]$. Thus $|L^D| \geq \alpha^2|L^0|$, which contradicts the definition of $D$ and that $|L^0| \geq 1$, $|L^D| \leq |L|$.

**B LOWER BOUND ON RECOURSE FOR LOAD BALANCING IN THE FULLY DYNAMIC MODEL**

In this section we consider OLBwR in the fully dynamic model. Again, we have a set $M$ of machines and a set $J$ of $n$ jobs. Jobs can arrive and depart, and at any time, we are guaranteed that there is a schedule of makespan at most $T^*$, for a given $T^*$. To achieve an $\alpha$-competitive ratio, our algorithm needs to maintain a solution of makespan at most $\alpha T^*$. An algorithm with an amortized recource of $\beta$ can make at most $\beta T^*$ reassignments for the first $t$ arrival/departure events, for any $t$. By making copies of jobs, we assume every job arrives and departs exactly once; so there are $2n$ events in the sequence. Our negative result holds even for the restricted assignment setting, and when we only need to maintain a fractional schedule. The lower bound holds in the offline setting; the best algorithm that knows everything upfront must incur a large recourse.

**Theorem B.1.** Let $a(n) = o\left(\log n\right)$ be a monotone non-decreasing function of $n$. There is an instance for the above problem such that the following holds. Any offline algorithm that maintains a fractional schedule of makespan at most $a(n)T^*$ needs to incur an amortized recource of $n \Omega(1/a(n))$.

To see why an algorithm needs to incur a large recourse in the fully dynamic model, consider the following simple instance. There are 2 machines, and $n'$ small jobs of size 1 each arrive at the beginning of the algorithm. Each of them can be assigned to the two machines. Big jobs have size $n'$ and they are 2 types of them: type-1 big jobs can only be assigned to machine 1, and type-2 big jobs can only be assigned to machine 2. Consider the following online updates: a type-1 big job arrives and departs, a type-2 big job arrives and departs, then a type-1 big job arrives and departs, and so on. Then at any time, there is a schedule of makespan $n'$ for active jobs. If the sequence is long enough, then any $(1 - c)$-competitive algorithm must incur an amortized recource of $\Omega(n')$.

The constant 1.5 can be made arbitrarily close to 2, if we introduce more machines. However, to go beyond 2, we need to use a recursive construction, using the basic instance as a building block.

Since our algorithm only needs to maintain fractional schedules, we can make the problem more general by allowing each job to have a reassignment cost $c_j \in \mathbb{Z}_{\geq 0}$: Reassigning $x$ fraction of a job $j$ incurs a recource of $x c_j$. To make the recource costs uniform without changing the instance, one can break a job of size $p_j$ with reassignment cost $c_j$ into $c_j$ jobs of size $p/c_j$, each with reassignment cost 1. This will change the number of jobs in the instance, and we take care of the issue at the end of this section. We compare the total recource of the algorithm against $\sum_{j \in J} c_j$. (Recall that every job arrives and departs only once).

Now describe the instance with recource costs. Let $L \geq 1, P \geq 3$ be two integers. We construct a perfect binary tree $T$ with $L$ levels of leaves; so there are $2^L$ leaves in the tree. The level of a vertex is $L$ minus its distance to the root. The root has level 0 and the leaves have level 0. Let $r, v, v^*$ be the root, vertex set and leaf set of the tree $T$ respectively. For every $v \in V \setminus V^*$, let left$(v)$ and right$(v)$ be the left and right child of $v$ respectively. For every $v \in V$, let $\ell(v)$ be the level of $v$ and $\Lambda(v)$ be the set of leaves that are descendants of $v$.

There is one machine at each leaf and so there are in total $m := 2^L$ machines. For every $v \in V$, we define a unit-size job $j_v$, which can be assigned to any machine at $\Lambda(v)$. We introduce multiple copies of $j_v$ in the instance. The recource cost $c_{j_v}$ of $j_v$ (and all its copies) is defined as $c_{j_v} := p_v/c_v$.

Algorithm 2 cstr-inst($u$) // $u \in V$

1. let $2^{\ell(v)}$ copies of the job $j_v$ arrive
2. if $\ell(v) > 0$ then
3. repeat $P$ times: cstr-inst(left$(v)$), cstr-inst(right$(v)$)
4. let the $2^{\ell(v)}$ copies of $j_v$ created at step 1 depart

The instance is constructed recursively by calling the procedure cstr-inst($u$), defined in Algorithm 2. After Step 1 of the procedure cstr-inst($u$) for some leaf $u$, we have $2^L + 2^{L-1} + 2^{L-2} + \cdots + 2^0 = 2^{L+1} - 1$ active jobs: For any ancestor $v$ of $u$, we have $2^{\ell(v)}$ active copies of $j_v$. It is easy to see that the active jobs can be scheduled on the machines so that each machine takes at most 2 jobs: for every strict ancestor $v$ of $u$ at level $\ell$, we schedule the $2^\ell$ copies of $j_v$ on the $2^{\ell-1}$ machines at $\Lambda(v')$, where $v'$ is the child of $v$ that is not an ancestor of $u$; the 1 copy of $j_v$ is scheduled on one machine at $u$. Therefore, the optimum makespan at any time is at most 2.

We show that any algorithm that maintains a fractional solution of makespan at most $\frac{2^{L+1}}{3}$ need to incur a large recourse. First, we
Finally, we only check the makespan of a schedule at the checkpoint; if recursion, the algorithm inserts these copies into the schedule, and fractional jobs will not be removed until they depart. There might the algorithm puts $(f(u) - \sum_{v \in \Lambda(u)} f(v))$ is the fractional number of copies of $j_0$ that we did not schedule in each iteration of the loop on line 3.

The sum of $c_j$ over all arrived jobs is

$$\sum_{u \in V} (f(u) - \sum_{v \in \Lambda(u)} f(v)) = PL \cdot \left( (L + 1) \cdot \alpha \right).$$

Above, $f(u)$ is the number of copies we create in each recursion of cstr-inst$(u)$, $PL - f(u)$ is the number of times we run the recursion, and $PL \cdot (L + 1)$ is the recourse for a copy of $j_0$.

Therefore, we proved that the total recourse is at least $\frac{P}{2} \cdot \sum c_j$. Splitting jobs to make all reassignment costs equaling 1, we obtain an instance with $n = (L + 1) \cdot 2P$ jobs, with amortized recourse $\Omega(P)$. Therefore, for any integer $\alpha$, we can set $L = 3\alpha - 1$, and $P = \left( \frac{n}{\alpha} \right)^{1/L} / 2 = \alpha \Omega(1/\alpha)$. This finishes the proof of Theorem B.1.

REFERENCES

[1] Ali Alakeel. 2009. A Guide to Dynamic Load Balancing in Distributed Computer Systems. International Journal of Computer Science and Network Security (IJCSNS) 10 (11) 2009.

[2] Susanne Albers. 1999. Better Bounds for Online Scheduling. SIAM J. Comput. 29, 2 (1999), 459–473.

[3] James Aspnes, Yossi Azar, Amos Fiat, Serge A. Plotkin, and Orli Waarts. 1997. Online routing of virtual circuits with applications to load balancing and machine scheduling. J. ACM 44, 3 (1997), 486–504.

[4] Baruch Awerbuch, Yossi Azar, Serge A. Plotkin, and Orli Waarts. 2001. Competitive Routing of Virtual Circuits with Unknown Duration. J. Comput. Syst. Sci. 62, 3 (2001), 385–397.

[5] Yossi Azar, Andrei Z. Broder, and Anna R. Karlin. 1994. On-Line Load Balancing. Theor. Comput. Sci. 130, 1 (1994), 73–84.

[6] Yossi Azar, Joseph Naor, and Raphael Rom. 1995. The Competitiveness of On-Line Assignments. J. Algorithms 18, 2 (1995), 221–237.

[7] Sebastian Berndt, Franziska Ebelke, and Nicole Megow. 2022. Online load balancing with general reassignment cost. Oper. Res. Lett. 50, 3 (2022), 322–328.

[8] Martin Gairing, Burkhard Monien, and Andreas Woclaw. 2007. A Faster combinatorial approximation algorithm for scheduling unrelated parallel machines. Theoretical Computer Science 380, 1 (2007), 87–99. https://doi.org/10.1016/j.tcs.2007.02.056. Automata, Languages and Programming.

[9] Anupam Gupta, Amit Kumar, and Cliff Stein. 2014. Maintaining assignments online: Matching, scheduling, and flows. In Proceedings of the twenty-fifth annual ACM-SIAM symposium on Discrete algorithms. SIAM, 448–479.

[10] John F. Rudin III and R. Chandrasekaran. 2003. Improved Bounds for the Online Scheduling Problem. SIAM J. Comput. 32, 3 (2003), 717–735.

[11] Jan Karel Lenaerts, David B Shmoys, and Eva Tardos. 1990. Approximation algorithms for scheduling unrelated parallel machines. Mathematical programming 46, 1 (1990), 259–271.

[12] Shi Li and Jiayi Xian. 2021. Online Unrelated Machine Load Balancing with Predictions Revisited. In Proceedings of the 38th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 139). Marina Meila and Tong Zhang (Eds.). PMLR, 6523–6532. https://proceedings.mlr.press/v139/li21w.html.

[13] Steven J. Phillips and Jeffery R. Westbrook. 1993. Online load balancing and network flow. In Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing, May 16-18, 1993, San Diego, CA, USA. S. Rao Kosaraju, David S. Johnson, and Alok Aggarwal (Eds.). ACM, 402–411.

[14] Peter Sanders, Naveen Sivadasan, and Martin Skutella. 2009. Online Scheduling with Bounded Migration. Math. Oper. Res. 34, 2 (2009), 481–498.

[15] David B. Shmoys and Eva Tardos. 1993. An Approximation Algorithm for the Generalized Assignment Problem. Math. Program. 62, 1–3 (feb 1993), 461–474.

[16] Jeffrey Westbrook. 2000. Load balancing for response time. Journal of Algorithms 35, 3 (2000), 1–16.

[17] David P Williamson. 2019. Network flow algorithms. Cambridge University Press.

Received 2022-11-07; accepted 2023-02-06.