Online Mechanism Design using Reinforcement Learning for Cloud Resource Allocation

Mateusz Ochal[1], Sebastian Stein[1], Fan Bi[1], Matthew Cook[1], Enrico Gerding[1], Ting He[2], and Thomas La Porta[2]

[1] Electronics and Computer Science, University of Southampton, United Kingdom
{mo4g15,ss2,mc18g14,fb1n15,eg}@ecs.soton.ac.uk
[2] School of Electrical Engineering and Computer Science, Penn State University, USA
{t.he,tlp}@cse.psu.edu

Abstract. We consider the allocation of dynamically arriving tasks with varying values and deadlines to resources within a cloud computing system. Reinforcement learning is a promising approach for this, but existing work has neglected that task owners may misreport their requirements strategically when this is to their benefit. To address this, we apply mechanism design and propose a novel mechanism based on reinforcement learning called Mono-RAwR that is incentive compatible and individually rational (i.e., truthful reporting and participation are incentivised). We evaluate our mechanism empirically on synthetic data, and we show that Mono-RAwR outperforms other state-of-the-art incentive compatible mechanisms by up to 70%, achieving about 80% of a greedy offline heuristic.

Keywords: Online mechanism design · Cloud resource allocation · Reinforcement learning

1 Introduction

Cloud computing allows organisations to pool their computational resources and share these with users who wish to run complex tasks [12]. Here, a key challenge is the dynamic allocation of heterogeneous tasks to computational resources, especially when resources are highly limited, and tasks have varying deadlines and values. While there has been significant work on designing allocation algorithms for these cases [6, 17], it typically neglects that in most realistic settings task owners are self-interested agents that may strategically misreport their requirements if this is to their benefit.

An exception to this is work that applies mechanism design to resource allocation [14]. This designs allocation and payment mechanisms to ensure desirable properties such as incentive compatibility (agents are best off being truthful) and individual rationality (agents do not make a loss). However, most work in this area focuses on *offline* settings, where full knowledge of future tasks is available [10, 9]. To deal with *online* settings, which are more realistic in dynamic cloud settings, work on online mechanism design [15] has been extended to resource allocation problems. Specifically, several model-free mechanisms have been proposed [18, 1, 13]. However, without a model of potential future task arrivals, they may over-allocate resources early on and are then unable to satisfy urgent, high-value tasks that arrive later [20].

To address this, model-based approaches are used in other domains to make better decisions using a stochastic model of future tasks [16, 20]. Similarly, work on automated mechanism design uses such models to automatically optimise mechanisms [19, 7]. However, these approaches can be computationally demanding and a model may not be available, e.g., due to limited initial domain knowledge or in non-stationary settings.

A promising resource allocation approach that overcomes these shortcomings is reinforcement learning [24, 21, 23, 11]. Here, adaptive algorithms like Q-learning can learn the long-term reward of taking particular allocation decisions without an explicit model [22]. Moreover, these algorithms are often robust when task demands change over time [8]. However, a key drawback is that these algorithms are not designed to be incentive compatible. Specifically, existing approaches typically learn arbitrary value functions for states or state-action pairs, and, depending on the limited training data an algorithm has been exposed to, these may present opportunities for strategic manipulation. For example, if the reinforcement learning mechanism has learnt that tasks with shorter deadlines should be given priority (a reasonable rule of thumb that is used in existing scheduling heuristics [17]), this can incentivise agents to misrepresent deadlines.

Now, some existing work has explicitly combined learning with mechanism design. Specifically, reinforcement learning has been used to optimise market mechanisms [5, 4], but that work also does not explicitly consider incentive compatibility. Other learning techniques have been used to design incentive compatible offline [2] and online auctions [3]. However, none of these is directly applicable to complex resource allocation problems, where allocation decisions have a significant impact on future states (e.g., allocation decisions affect the future availability of system resources, but can also be postponed if deadlines are sufficiently flexible).

Against this background, we consider, for the first time, how to design incentive compatible adaptive mechanisms based on reinforcement learning for a complex resource allocation problem. In doing so, we advance the state of the art in several ways. First, we propose a new mechanism called **Mono**tonic **R**esource **A**llocation **w**ith **R**einforcement learning (Mono-RAwR), which implements several modifications compared to standard reinforcement learning approaches, in order to limit the scope of strategic manipulation. The key novelty is that these modifications enforce specific monotonicity conditions on the underlying value function (and consequently on the allocation decisions) of our learning algorithm. Second, we prove that these modifications ensure that Mono-RAwR is dominant strategy incentive compatible and individually rational. Third, we conduct extensive empirical experiments, showing that Mono-RAwR outperforms state-of-the-art incentive compatible approaches by up to 70% and that it achieves around 80% of an offline benchmark.

2 System Model

We assume a finite time horizon and discrete time steps $T = \{1, 2, \ldots, T_{\max}\}$. There is a set of tasks $I = \{1, 2, \ldots\}$ that arrive dynamically over time, and each task i is characterised by an arrival time a_i, a deadline d_i, a required quantity of work q_i and a value v_i. The arrival time denotes the time at which the task becomes known to the system and the earliest time it can be allocated to a resource. The deadline indicates

the latest time by which it must be completed. The quantity of work corresponds to the computational effort that is needed to complete the task (e.g., measured in number of instructions). Finally, the value indicates how important the task is and could correspond to a financial opportunity cost that is incurred if this task is not completed in time.

Together, we describe the type of task i as $\theta_i = (a_i, d_i, q_i, v_i)$. We use θ^t to denote the set of tasks that have arrived by time t (i.e., with $a_i \leq t$), and we use $\theta = \theta^{T_{\max}}$ to denote all tasks. Furthermore, there is a set of m resources (physical or virtual machines): $R = \{1, 2, \ldots, m\}$. Each resource r produces $p_r \in \mathbb{R}^+$ units of work per time step. Tasks can be allocated to resources — once started, the resource will work on it until completion (i.e., tasks are non-preemptive), and a resource can work on at most one task per time step. Thus, when allocating task i to resource r, the resource is occupied for $\lceil q_i/p_r \rceil$ time steps.

Given this, at every time step t, we need to decide which waiting tasks to allocate to which unoccupied resources. We denote this decision as $\pi(i, t, \theta^t) \in R \cup \{0\}$, where $\pi(i, t, \theta^t) = r \in R$ means that task i will be allocated to resource r at time t and $\pi(i, t, \theta^t) = 0$ means that task i is not allocated then. Note the dependence on θ^t, i.e., allocation decisions have to be made *online* based only on known tasks that have arrived by t. We assume that tasks are only allocated once, i.e., $\forall i : |\{t \in T \mid \pi(i, t, \theta^t) \neq 0\}| \leq 1$. Finally, we use π to denote the overall policy (or allocation algorithm) that determines the allocations at every time step, and we use $x_i(\pi, \theta)$ to indicate whether task i was completed successfully by its deadline: $x_i(\pi, \theta) = 1$ if $\exists t \geq a_i, r > 0 : (\pi(i, t, \theta^t) = r) \wedge (t + \lceil q_i/p_r \rceil \leq d_i)$ and $x_i(\pi, \theta) = 0$ otherwise.

Our objective is to maximise the total value derived from tasks that complete within their deadline (the *social welfare*), i.e., $W(\pi, \theta) = \sum_{i \in I} x_i(\pi, \theta) \cdot v_i$. It is easy to show that this problem is NP-hard, even when all tasks are known in advance, as it is a generalisation of the NP-hard problem of minimising the total weighted number of tardy jobs on a single machine [17].

Next, we will describe a general solution to this problem, assuming that task owners are cooperative and report truthfully, followed by the strategic setting in Section 4.

3 Resource Allocation in Cooperative Settings

In this section, we outline a generic algorithm for our resource allocation problem (Section 3.1). This allows us to represent several existing benchmarks (Section 3.2) as well as a novel reinforcement learning approach (Section 3.3) using a single framework.

3.1 Generic Resource Allocation Algorithm

Algorithm 1 shows how we model a mechanism that myopically considers each unallocated task in isolation and decides whether to *admit* this task to the schedule or to postpone it, i.e., delay the admission decision. This framework broadly follows [20], but it allows us to use reinforcement learning methods, which is a key novelty of our work. The filtering decision is made based on the state representation of both the task under consideration and the current availability of the resources. Although we investigated representing several tasks in a single state, we found that single tasks reduced the

overall state space of our reinforcement learning solution, and became neccessary for monotonicity constraints as explained in Section 4. In more detail, we gradually build up the allocation decisions π, i.e., we start with no allocations (Line 3) and then add allocation decisions over time (up to the current time step) with increasing knowledge of the available tasks.[3] We also keep track of two sets of tasks: θ_{wait} are tasks that have arrived but have not been admitted yet, θ_{admitted} are those that the algorithm intends to execute (tasks in θ_{admitted} may be returned to θ_{wait}, as explained below). At each time step, the algorithm considers all waiting tasks in the order given by ρ. This function takes the set of waiting tasks yet to consider, and returns the next task (we order tasks by value density, but other functions could be used). For each task, we call a CHECK-FEASIBLE function, which checks whether the task can be scheduled along with all previously admitted tasks. This function uses the SCHEDULE heuristic function, which is explained below. If a feasible schedule exists, we pass the task through another filtering algorithm using the ADMIT function, which returns 1 to indicate admission and 0 to indicate postponement. On successful admission, we update the task sets accordingly.

When all tasks have been considered, we generate a temporary feasible schedule π' with all admitted tasks. This is done using the SCHEDULE(π, θ) function, which takes a partial schedule π and a set of tasks θ as input, and then schedules the tasks while keeping scheduling decisions in π fixed.[4] Lines 17–20 allocate tasks for current time t that appear in the feasible schedule, keeping these fixed from now on. All admitted tasks that are not scheduled now but in the future in π' (i.e., at $t+1$ or later) are returned to θ_{wait}. Finally, Line 23 removes any infeasible tasks from the waiting set.

In the following sections, we briefly explain a number of simple benchmarks and then a new reinforcement learning algorithms built on this framework.

3.2 Benchmark Algorithms

We use the following algorithms as benchmarks. With the exception of offline greedy, all are based on the general framework described in the preceding section.

- **OfflineGreedy** is a heuristic[5] approach for obtaining an upper bound on performance by assuming perfect knowledge of future tasks. Considering the tasks in descending order of value density, v_i/q_i, the algorithm tries to schedule each task on the slowest resource first, on its arrival time. If occupied at that time, it tries all subsequent time steps that still satisfy the deadline before trying the next fastest resource. For detailed pseudocode, see Algorithm 4 in Appendix A.
- **OnlineGreedy** uses ρ to sort the tasks by descending value density. The ADMIT function always returns 1, so it always tries to schedule when it can. An incentive compatible version of this algorithm (**OnlineGreedyPC**) can be obtained by adding

[3] Note, in Algorithm 1, we leave out the dependence of π on θ^t for clarity.

[4] We use offline greedy for this, as described in Section 3.2. Full pseudocode for SCHEDULE and CHECK-FEASIBLE can be found in Appendix A.

[5] We also implemented an optimal approach in ILOG CPLEX, but this was too slow for larger problem instances. However, on smaller instances, we obtained similar performance to offline greedy, so we are using the latter as indicative of what an offline solution could achieve.

Algorithm 1 Generic algorithm for resource allocation problem

1: $\theta_{\text{wait}} \leftarrow \emptyset$ ▷ Unscheduled tasks
2: $\theta_{\text{admitted}} \leftarrow \emptyset$ ▷ Admitted tasks
3: $\pi(i,t) \leftarrow 0, \forall t, i$ ▷ Allocation decisions
4: **for** $t \in \{1,\dots,T\}$ **do** ▷ For each time step
5: Update θ^t ▷ Receive new tasks
6: $\theta_{\text{wait}} \leftarrow \theta_{\text{wait}} \cup \{\theta_i \in \theta^t \mid a_i = t\}$ ▷ Update tasks
7: $\theta' \leftarrow \theta_{\text{wait}}$ ▷ Tasks to consider
8: **while** $\theta' \neq \emptyset$ **do**
9: $\theta_i \leftarrow \rho(\theta')$ ▷ Next to consider
10: $\theta' \leftarrow \theta' \setminus \{\theta_i\}$ ▷ Remove considered task
11: **if** CHECK-FEASIBLE$(\theta_i, \theta_{\text{admitted}}, \pi)$ **then**
12: $j \leftarrow$ ADMIT$(\theta_i, t, \theta_{\text{admitted}}, \pi)$ ▷ Admit?
13: **if** $j = 1$ **then** ▷ Admitted?
14: $\theta_{\text{wait}} \leftarrow \theta_{\text{wait}} \setminus \{\theta_i\}$ ▷ No longer available
15: $\theta_{\text{admitted}} \leftarrow \theta_{\text{admitted}} \cup \{\theta_i\}$ ▷ Admitted
16: $\pi' \leftarrow$ SCHEDULE$(\pi, \theta_{\text{admitted}})$ ▷ Feasible schedule
17: **for** $\theta_k \in \theta_{\text{admitted}}$ **do** ▷ Allocate tasks for time t
18: **if** $\pi'(k,t) \neq 0$ **then**
19: $\pi(k,t) \leftarrow \pi'(k,t)$
20: $\theta_{\text{admitted}} \leftarrow \theta_{\text{admitted}} \setminus \{\theta_k\}$ ▷ Scheduled
21: $\theta_{\text{wait}} \leftarrow \theta_{\text{wait}} \cup \theta_{\text{admitted}}$ ▷ Move admitted tasks
22: $\theta_{\text{admitted}} \leftarrow \emptyset$ ▷ Empty admitted tasks
23: $\theta_{\text{wait}} \leftarrow \{\theta_i \in \theta_{\text{wait}} \mid d_i > t\}$ ▷ Remove expired

the concept of pre-commitments [20]: once a task is added to θ_{admitted}, the algorithm commits to execution (Lines 21 and 22 are omitted).

- **Earliest Deadline First (EDF)** uses ρ to sort the tasks by ascending deadlines, and its ADMIT function also returns 1. This algorithm is not incentive compatible.
- **Consensus** sorts tasks by descending value density and implements the Consensus algorithm [20] in its ADMIT function. This votes for the inclusion of tasks by considering multiple sampled future scenarios. Two versions exist: **Consensus** and its incentive compatible counterpart, **ConsensusPC** with pre-commitments.

In the following section, we discuss how we solve the resource allocation problem using reinforcement learning.

3.3 Reinforcement Learning

In principle, the resource allocation problem can be formulated as an MDP and solved using standard reinforcement learning techniques (with appropriate assumptions on the arrival distribution). Here, the state space would include the available tasks to schedule, as well as the current availability of resources. The actions would be decisions to schedule tasks on specific resources, and the transition function would model the arrival distribution of tasks and their (deterministic) completion once they have been scheduled. Rewards depend directly on the task values that are completed by their deadline.

However, the state space for this MDP is exponentially large, primarily because it needs to represent the tasks that have not been allocated to resources yet. Furthermore, the action space is potentially very large too, as multiple tasks can be allocated to different resources at each time step. For this reason, we also adopt the myopic approach outlined in Algorithm 1 and now present a novel Resource Allocation with Reinforcement learning (**RAwR**) algorithm. In more detail, RAwR sorts the tasks by value-density in descending order. In the ADMIT function, we construct a state representation using θ_i and θ_{admitted}, and we use a linear combination of features to approximate a value function, $q_j(\mathbf{s})$, for each action $j \in \{0, 1\}$ (i.e., postpone or admit). By allowing only two actions, we restrict the complexity of the reinforcement learning algorithm and defer any scheduling decisions to the offline greedy scheduling heuristic. Given this, we then return the best action greedily with respect to $q_j(\mathbf{s})$, i.e., $\text{argmax}_{j\in\{0,1\}}\, q_j(\mathbf{s})$.

To represent a state $\mathbf{s}$ given a task, $\theta_i = (a_i, d_i, q_i, v_i)$, we use $\mathbf{s} = [1, d_i - t - \phi(q_i), q_i, v_i, v_i/q_i, o_1, \ldots, o_m, \gamma, \mu]$, where the initial 1 allows a bias to be added to each action, $d_i - t - \phi(q_i)$ expresses deadline slackness on the fastest resource, i.e., $\phi(q_i) = q_i / \max_r p_r$, v_i/q_i is task's value-density, o_r is the current occupation for resource r (i.e., in how many time steps it will be free, as given by π), $\gamma = |\theta_{\text{admitted}}|$ is the number of admitted tasks, and μ is the average quantity of work for those tasks.

Different reinforcement learning algorithms could be used to implement ADMIT, and Algorithm 2 shows a simple implementation based on Q-Learning [22]. This learns the values of actions in different states without an explicit model. We use the linear function approximator to estimate the value function and we learn the weights with mini-batch gradient descent [21]. Controlling the way we update the weights helps us establish a mechanism that is robust to strategic manipulation, as explained in Section 4.

More specifically, we use two weight vectors, $\mathbf{w}_0$ and $\mathbf{w}_1$ for the two actions. In Line 6 these are first updated using an appropriate UPDATE-WEIGHTS function based on the previous state, action and the derived reward.[6] We normalise the state into a feature vector by dividing by the maximum value for each dimension. Taking the dot product of the feature vector with the weight vectors, we derive an approximation for the expected future reward of each action (Line 8). An appropriate action is then returned via the CHOOSE-ACTION. This is greedy in our case, but we could include some element of exploration (e.g., via an ϵ-greedy strategy — for an example see Appendix A.4).

4 Resource Allocation in Strategic Settings

We now consider a setting with self-interested task owners.

4.1 Extended System Model with Strategic Agents

In practice, task owners should be seen as strategic agents that may misreport their types, if this is beneficial.[7] To consider such cases, we assume that task types are reported as $\hat{\theta}_i = (\hat{a}_i, \hat{d}_i, \hat{q}_i, \hat{v}_i)$. We make the standard assumption of limited misreports,

[6] While different implementations could be used for UPDATE-WEIGHTS and CHOOSE-ACTION, the specific functions we use are detailed in Appendix A.

[7] We use agent and task owner interchangeably. We also assume each task owner is associated with exactly one task.

Algorithm 2 Simple reinforcement learning algorithm

$\mathbf{w}_0, \mathbf{w}_1$ ▷ Initial weights
$\mathbf{s}_{\text{last}}, a_{\text{last}}, v_{\text{last}}, \mathbf{s}$ ▷ Last state transition (initially empty)

procedure ADMIT($\theta_i, t, \theta_{\text{admitted}}, \pi$)
  $\mathbf{s} \leftarrow [1, d_i - t - \phi(q_i), q_i, v_i, v_i/q_i, o_1, \ldots, o_m, \gamma, \mu]$
  $\mathbf{w}_0, \mathbf{w}_1 \leftarrow$ UPDATE-WEIGHTS($\mathbf{s}_{\text{last}}, a_{\text{last}}, v_{\text{last}}, \mathbf{s}$)
  $\mathbf{f} \leftarrow$ NORMALISE($\mathbf{s}$)
  $q_0, q_1 \leftarrow \mathbf{fw}_0, \mathbf{fw}_1$ ▷ Values of actions
  $\mathbf{s}_{\text{last}} \leftarrow \mathbf{s}$
  $a_{\text{last}} \leftarrow$ CHOOSE-ACTION(q_0, q_1) ▷ Select action
  $v_{\text{last}} \leftarrow v(a_{\text{last}})$ ▷ $v(1) = v_i$ and $v(0) = 0$
  return a_{last}

i.e., $\hat{a}_i \geq a_i$, $\hat{d}_i \leq d_i$ and $\hat{q}_i \geq q_i$. This is reasonable in our setting. Specifically, we assume a_i is the time a task owner becomes aware of the task, and so no reports can be made earlier than this. Furthermore, a resource allocation mechanism can be easily designed to withhold the results of tasks until the reported deadline $\hat{d}_i$. Finally, we also assume that no more than $\hat{q}_i$ work is done on a task. Thus, over-reporting $\hat{d}_i$ or under-reporting $\hat{q}_i$ would guarantee that the task is not completed successfully.

Given this, we denote by $\hat{\theta}$ the reported types of all agents, $\hat{\theta}^t$ the types of agents that reported their arrival by time t. We will also use $\hat{\theta}^t_{-i}$ to denote all reported types except that of agent i. Thus, $\hat{\theta}^t = (\hat{\theta}_i, \hat{\theta}^t_{-i})$ and $\hat{\theta} = (\hat{\theta}_i, \hat{\theta}_{-i})$. Since the resource allocation policy π will operate on the reported types $\hat{\theta}$, we need to modify x_i to explicitly consider the agent's true type: $x_i(\pi, \theta_i, (\hat{\theta}_i, \hat{\theta}_{-i})) = 1$ if $\exists t \geq \hat{a}_i, r > 0 : \pi(i, t, \hat{\theta}^t) = r \wedge\ t + \lceil \hat{q}_i / p_r \rceil \leq d_i \wedge\ \hat{q}_i \geq q_i \wedge\ \hat{d}_i \leq d_i$; and $x_i(\pi, \theta_i, (\hat{\theta}_i, \hat{\theta}_{-i})) = 0$ otherwise.

To address strategic settings, we now apply tools from mechanism design and assume that we can impose payments on task owners. Thus, we define a mechanism $M = (\pi, \tau)$ as an allocation policy π (as defined previously) and a payment policy τ. Here, $\tau_i(\hat{\theta})$ is the payment imposed on agent i. Given this, the utility of agent i is $u_i(\pi, \tau, \theta_i, \hat{\theta}) = x_i(\pi, \theta_i, \hat{\theta}) \cdot v_i - \tau_i(\hat{\theta})$. With this, we can define two desirable properties for a mechanism $M = (\pi, \tau)$.

Definition 1 (Dominant Strategy Incentive Compatibility (DSIC)). *A mechanism $M = (\pi, \tau)$ is dominant strategy incentive compatible if and only if $\forall i, \theta_i, \hat{\theta}_i, \hat{\theta}_{-i}$: $u_i(\pi, \tau, \theta_i, (\theta_i, \hat{\theta}_{-i})) \geq u_i(\pi, \tau, \theta_i, (\hat{\theta}_i, \hat{\theta}_{-i}))$.*

This means that an agent i maximises its utility by reporting its type θ_i truthfully.

Definition 2 (Individual Rationality (IR)). *A mechanism $M = (\pi, \tau)$ is individually rational if and only if $\forall i, \hat{\theta}_i, \hat{\theta}_{-i} : u_i(\pi, \tau, \hat{\theta}_i, (\hat{\theta}_i, \hat{\theta}_{-i})) \geq 0$.*

This means that an agent i never receives a negative utility, and this is important to ensure that agents have an incentive to participate in the mechanism.

To help us analyse the strategic setting, we recall some general results for domains with single-valued preferences (i.e., where agents have a single value for any successful allocation of resources, as in our domain) [15]. First, a necessary condition for a

mechanism $M = (\pi, \tau)$ to be DSIC is that its allocation policy π is monotonic. Here, monotonicity is defined with regard to a partial order $\preceq_\theta$ over the types, which in our case is $\theta_i \preceq_\theta \theta_j \equiv (a_i \geq a_j) \wedge (d_i \leq d_j) \wedge (q_i \geq q_j) \wedge (v_i \leq v_j)$. Given this, an allocation policy π is monotonic if $\forall \theta_i \preceq_\theta \theta'_i, \theta_{-i} : x_i(\pi, \theta_i, (\theta_i, \theta_{-i})) \leq x_i(\pi, \theta'_i, (\theta'_i, \theta_{-i}))$, i.e., if a lower type is allocated successfully, then so is a higher type. Second, a monotonic mechanism that is individually rational and DSIC must collect a payment equal to an agent's critical value, and this is both a necessary and sufficient condition. Here, the critical value is defined as the minimum value v_i agent i could report to be allocated, i.e., $v_i^c(a_i, d_i, q_i, \theta_{-i}, \pi) = \min v_i$, subject to $x_i(\pi, \theta_i, (\theta_i, \theta_{-i})) = 1$, where $\theta_i = (a_i, d_i, q_i, v_i)$ (and $v_i^c(a_i, d_i, q_i, \theta_{-i}, \pi) = \infty$ when no allocation is possible).

In general, it is easy to see that Algorithms 1 and 2 do not necessarily satisfy monotonicity. In particular, if tasks with a shorter deadline are selected first by the ρ function (in Line 9), as in EDF, then monotonicity may be violated. Similarly, the exact weight vectors $\mathbf{w}_0$ and $\mathbf{w}_1$ may break monotonicity, e.g., if $\mathbf{w}_1$ assigns a higher weight to the q_i component of the state than $\mathbf{w}_0$ (which may happen if there are particular correlations in the task characteristics or if the algorithm has observed insufficient data).

To deal with this, we present a variant of RAwR in the following section, which makes a number of modifications to Algorithms 1 and 2 to ensure monotonicity.

4.2 Monotonic Reinforcement Learning Mechanism (Mono-RAwR)

To keep the discussion in this section concise, we will assume that the UPDATE-WEIGHTS function in Line 6 of Algorithm 2 is inactive, i.e., that the weight vectors $\mathbf{w}_0$ and $\mathbf{w}_1$ are fixed and no longer change. This represents a situation where the algorithm has converged on a good set of weights and is no longer learning. For that reason, we also assume that CHOOSE-ACTION is deterministic and greedy, returning $\operatorname{argmax}_{i \in \{0,1\}} q_i$. These assumptions help us prove incentive compatibility, as they prevent agents from strategically influencing future weight vectors. Clearly, our main objective is to design mechanisms where these weights can be learnt over time, and in Section 4.3 we will discuss how this can be achieved in practice. Given this, we now propose several modifications to achieve monotonicity:

- **Pre-commitment**: As in [20], once a task is added to θ_{admitted}, the algorithm must commit to executing it. This is achieved by removing Lines 21 and 22 in Algorithm 1. Note that this means that when the ADMIT function returns 1 for a particular task θ_i, it is guaranteed to be executed (i.e., $x_i(\pi, \theta_i, (\theta_i, \theta_{-i})) = 1$).
- **Monotonicity of ρ**: The priority ordering determined by ρ must preserve monotonicity, i.e., it must hold that $\forall \theta : (\nexists \theta_i \in \theta : \rho(\theta) \prec_\theta \theta_i)$. This is easy to achieve, for example, by ordering the tasks by decreasing value (v_i) or value density (v_i/q_i) and breaking ties in a way that is consistent with monotonicity (we use v_i/q_i).
- **Monotonicity of SCHEDULE**: The scheduling heuristic SCHEDULE must be monotonic, i.e., $\forall \theta_i \prec \theta'_i, \theta_{-i} : y_i(\text{SCHEDULE}(\theta_i, \theta_{-i})) \leq y_i(\text{SCHEDULE}(\theta'_i, \theta_{-i}))$, where $y_i(\text{SCHEDULE}(\theta_i, \theta_{-i})) \in \{0, 1\}$ indicates whether θ_i is successfully scheduled. As CHECK-FEASIBLE uses the SCHEDULE function, this inherits its monotonicity. Note that the offline greedy heuristic we use already satisfies this monotonicity property.

– **Re-evaluation on successful admission**: Whenever a task is added to the set of admitted tasks, the state of the system changes and this may impact future scheduling decision. To avoid agents strategising about these changes, we re-evaluate all waiting tasks again when a new task is admitted (following the order given by ρ). This is done by adding a new line to Algorithm 1 after Line 14: $\theta' \leftarrow \theta_{\text{wait}}$.
– **Monotonicity of weight vectors**: The weight vectors $\mathbf{w}_0$ and $\mathbf{w}_1$ must preserve monotonicity in the sense that higher types are more likely to be allocated than lower types, i.e., $\forall t, o_{1,\ldots,r}, \gamma, \mu, \theta_i \preceq_\theta \theta_i' : s(\theta_i, t, o_{1,\ldots,r}, \gamma, \mu,)\mathbf{w}_1 - s(\theta_i, t, o_{1,\ldots,r}, \gamma, \mu,)\mathbf{w}_0 \leq s(\theta_i', t, o_{1,\ldots,r}, \gamma, \mu,)\mathbf{w}_1 - s(\theta_i', t, o_{1,\ldots,r}, \gamma, \mu,)\mathbf{w}_0$, where $s(\theta_i, t, o_{1,\ldots,r}, \gamma, \mu,)$ returns the appropriate normalised state representation $\mathbf{f}$, as given by Lines 5 and 7 of Algorithm 2. This can be achieved by ensuring that all of the following conditions hold:
  - $w_{0,2} \leq w_{1,2}$ (where $w_{j,2}$ weights $d_i - t - \phi(q_i)$).
  - $w_{0,3} \geq w_{1,3}$ (where $w_{j,3}$ weights q_i).
  - $w_{0,4} \leq w_{1,4}$ (where $w_{j,4}$ weights v_i).
  - $w_{0,5} \leq w_{1,5}$ (where $w_{j,5}$ weights v_i/q_i).

  Together, these ensure that a longer deadline, lower quantity of work and higher value can only increase $q_1 = \mathbf{f}\mathbf{w}_1$ relative to $q_0 = \mathbf{s}\mathbf{w}_0$ (keeping everything else constant), thus ensuring that monotonicity holds.
  Note that given two arbitrary weight vectors $\mathbf{w}_0$ and $\mathbf{w}_1$, we can derive a set that satisfies monotonicity by using $\mathbf{w}_0' = [w_{0,1}, \min(w_{0,2}, w_{1,2}), \max(w_{0,3}, w_{1,3}), \min(w_{0,4}, w_{1,4}), \min(w_{0,5}, w_{1,5}), w_{0,5}, \ldots, w_{0,|\mathbf{w}|}]^{\mathrm{T}}$ and $\mathbf{w}_1' = \mathbf{w}_1$ instead. Thus, to achieve monotonicity in practice, we apply this correction whenever new weights are chosen.

Overall, we refer to a mechanism that follows Algorithms 1 and 2, with the above modifications, as **Mono-RAwR**. We now show several properties for this algorithm.

Lemma 1. *Mono-RAwR with fixed weights and greedy action selection is monotonic.*

Proof. Assume for contradiction that there are two types $\theta_i \preceq_\theta \theta_i'$, with $x_i(\pi, \theta_i, (\theta_i, \theta_{-i})) = 1$ and $x_i(\pi, \theta_i', (\theta_i', \theta_{-i})) = 0$ (i.e., the only condition that violates monotonicity). First, by the use of pre-commitments, recall that $x_i(\pi, \theta_i, (\theta_i, \theta_{-i})) = 1$ is determined the first time ADMIT returns 1. Consider this point in the execution of the algorithm. Now, if θ_i' had been present instead of θ_i, it would still be in θ_{wait} at this point, given the assumption that it is unallocated and given that its deadline is not less than that of θ_i (by the definition of $\preceq_\theta$).

Due to the monotonicity of ρ and the re-evaluation on successful admission, θ_i' is guaranteed to be considered at the same point or earlier, and in the same system state as when θ_i is successfully admitted. Due to the monotonicity of SCHEDULE, CHECK-FEASIBLE$(\theta_i', \theta_{\text{admitted}}, \pi)$ will return true, so ADMIT is also called for θ_i'. We know that for θ_i, $q_0 = \mathbf{f}\mathbf{w}_0 \leq q_1 = \mathbf{f}\mathbf{w}_1$.[8] Due to the monotonicity of the weight vectors, $q_0 \leq q_1$ must also hold for θ_i'. Thus, ADMIT returns 1 and θ_i' is admitted, leading to a contradiction.

[8] W.l.o.g. we assume that ties are broken in favour of admission.

Given this result, we can show the following:

Theorem 1. *Mono-RAwR with fixed weights* $\mathbf{w}_0$ *and* $\mathbf{w}_1$*, greedy action selection and critical value payments is DSIC and IR.*

This can be shown directly using the results in [15]. Note that the critical value payment for a task θ_i can be computed in a similar way to that in [20] by running an instance of the setting without θ_i and, for each possible position in ρ and for each time step, determining the lowest value the agent could have reported and be allocated (which can be determined analytically for a given system state from the respective weight vectors). The critical value is then the lowest of these.

4.3 Adaptive Mono-RAwR

While we assumed fixed weight vectors and greedy action selection in the discussion above, we now briefly discuss two ways in which this can be relaxed.

Historical learning: As the only source of randomness in our setting is the arrival distribution of tasks, historical data can be used as the basis for simulating the setting and finding a good set of weight vectors. This can include exploration via ϵ-greedy or softmax. However, when the algorithm is executed in a real setting, the weights are fixed and greedy action selection is used. In practice, and to ensure that the historical data is based on truthful reports, the mechanism can run in an episodic fashion, i.e., re-train its weights regularly (e.g., at the start of a day) based on past data.

Continuous learning: Alternatively, learning and exploration can be interleaved with execution. However, it is important to enforce that agents cannot influence the weight vectors that are used during their presence in the system (either for their own allocations or those of other task owners), and that they cannot influence their likelihood of benefiting from exploratory actions. In practice, this can be achieved by delaying updates of the weight vectors (until a completely new set of agents is present) and by deciding once per task whether to apply an exploratory action (independently of the type reported). Since a formal discussion of this setting is beyond the scope of this paper, we leave this to future work.

5 Experimental Evaluation

In this section, we empirically evaluate Mono-RAwR against the benchmarks described in Section 3.2.

5.1 Experimental Setup

We consider a representative[9] setting with seven resources, $R = \{1, 2, \ldots, 7\}$. We assume three speed levels, $p_1 = p_2 = p_3 = 1$, $p_4 = p_5 = 0.75$ and $p_6 = p_7 = 0.5$ (we normalise these here for ease of exposition). This setting captures a heterogeneous set of resources while keeping the problem tractable for the Consensus benchmark algorithms,

[9] We observed similar trends in a range of other settings.

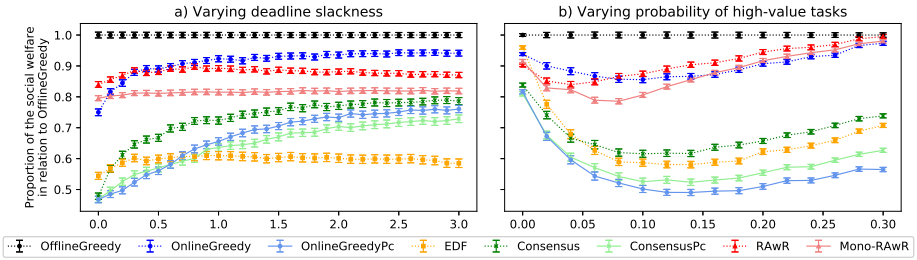

Fig. 1: The figures show the average social welfare of all algorithms as a fraction of the offline greedy solution, in various parameter settings. **a)** Shows how varying the upper bound of the deadline slackness of high-value tasks (on the x-axis) affects the performance of the algorithms. The deadline slackness is expressed as the proportion of a high-value task's size, q_i. **b)** Shows how varying the probability of a high-value task given task arrival (on the x-axis) affects the performance. 95% confidence intervals are shown on all figures.

which ran slowly for larger settings. For the same reason, we also consider episodes with a fixed time horizon $T_{\max} = 50$ in each. However, our RAwR algorithms scale to significantly larger settings and infinite time horizons.

We assume tasks arrive according to a Poisson distribution (with a constant parameter $\lambda = 1.8$ tasks per time step). Each arriving task is either a low-value or a high-value task (which follow different distributions). This is an interesting and challenging setting because a resource allocation strategy needs to balance running frequent low-value tasks with keeping resources free for the occasional high-value task.

Task values (v_i), deadlines (d_i) and required quantities (q_i) of work are drawn from uniform distributions. For low-value tasks, we choose v_i as an integer from 5 to 20, q_i from 1 to 7, and d_i is generated by multiplying q_i with a real number chosen uniformly at random from $[0, 3]$ and then using the earliest completion time on the fastest machine to determine d_i (rounded to the nearest integer)[10]. For high-value tasks, v_i is chosen between 180 to 200 and q_i between 8 and 10. We show how varying the deadline slackness of high-value tasks affects the performance of our algorithm in Figure1a, where the x-axis represents the upper bound of the scalar used for determining d_i (here, it is chosen from $[0, x]$). In that experiment, the probability of a task being a high-value task is 0.1 (and 0.9 for a low-value task). For the second experimental setup in Figure 1b we show how varying the probability of high-value tasks affects performance. Here, we keep the deadline slack very short (a random integer between 1 and 3 units of work on the fastest resource irrespective of the size).

For each different setting, we trained the RAwR algorithms with 50,000 episodes[11] with mini-batch gradient descent with the batch size of 1,000 and 100 iterations. We also gradually decreased the learning rate from 0.01 to 0.001, and exploration rate from 0.5 to 0.005 (using ϵ-greedy during training). And we set the discount-rate parameter

[10] For example, if we draw a random number of u, then $d_i = a_i + \phi(q_i * (u + 1))$, where $\phi(q_i) = q_i / \max_r p_r$, and $\max_r p_i$ is the speed of the fastest resource.

[11] Typically, much fewer episodes are needed (1,000s), but we use this to ensure convergence.

to 0.975. We also run all benchmarks described in Section 3.2 (using 20 scenarios for Consensus and ConsensusPc). Then we run 400 different episodes for each algorithm and record the average social welfare. Presented in the figures is the social welfare as a fraction of the full information heuristic, offline greedy.

5.2 Results

The results show that RAwR and its incentive compatible counterpart Mono-RAwR perform consistently well (achieving 80-90% of offline greedy[12]). The performance of RAwR is similar to online greedy, and Mono-RAwR performs only slightly worse. This is promising, showing that the cost of achieving incentive compatibility is relatively low. More importantly, Mono-RAwR performs significantly better than any other incentive compatible benchmark. Specifically, it beats the state-of-the-art ConsensusPc across all experiments not only in the total social welfare derived (by 25-50% on average and by up to 70% in some cases) but also in execution performance (running approximately 20 times faster when using 20 scenarios). This shows that on average the benchmarks tend to omit the high-value tasks due to their long deadlines and large sizes, whereas the RAwR algorithms appropriately filter low-value tasks according to resource availability.

Focusing on Figure1a, we can see that when high-value tasks have stricter deadlines, the performance of other benchmarks predictably decreases, whereas the RAwR algorithms retain just slightly worse or similar performance throughout as they learn to keep a fast resource free for high-value tasks. Consensus, due to its voting mechanism, completely omits the high-value tasks. Increasing the deadline slackness increases the likelihood and the possibility of scheduling a high-value task on more resources. Increasing the deadline slackness increases the likelihood and the possibility of scheduling a high-value task on more resources, increasing the performance of other benchmarks.

Looking at Figure1b, when there are no high-value tasks (i.e., the probability of such a task is 0), EDF performs best. However, as soon as high-value tasks appear, RAwR and Mono-RAwR take advantage of this and learn to schedule these tasks. OnlineGreedy and Mono-RAwR approach OfflineGreedy as the probability increases.

6 Conclusions and Future Work

We considered a resource allocation problem for cloud systems and showed how to apply reinforcement learning in this setting while incentivising truthful reporting. Our novel combination of incentive compatible mechanism design with reinforcement learning is promising for designing mechanisms in real-world settings. Unlike existing work, our approach requires no prior model of task arrivals but is able to learn a better allocation strategy adaptively based on historical data.

In future work, we will explore settings with multiple tasks per agent and more complex valuations beyond the single-value case studied here. We will also develop effective approaches for continuous learning (see Section 4.3).

[12] As explained earlier, the optimal solution was too slow to run.

Acknowledgments

This research was sponsored by the U.S. Army Research Laboratory and the U.K. Ministry of Defence under Agreement Number W911NF-16-3-0001. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Army Research Laboratory, the U.S. Government, the U.K. Ministry of Defence or the U.K. Government. The U.S. and U.K. Governments are authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation hereon.

References

1. Azar, Y., Kalp-Shaltiel, I., Lucier, B., Menache, I., Naor, J.S., Yaniv, J.: Truthful online scheduling with commitments. In: Proceedings of the Sixteenth ACM Conference on Economics and Computation. pp. 715–732. ACM (2015)
2. Balcan, M.F., Blum, A., Hartline, J.D., Mansour, Y.: Mechanism design via machine learning. In: 46th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2005). pp. 605–614. IEEE (2005)
3. Blum, A., Hartline, J.D.: Near-optimal online auctions. In: Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2005). pp. 1156–1163. Society for Industrial and Applied Mathematics (2005)
4. Cai, Q., Filos-Ratsikas, A., Tang, P., Zhang, Y.: Reinforcement mechanism design for e-commerce. In: Proceedings of the 2018 World Wide Web Conference (WWW'18). pp. 1339–1348. WWW '18, International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, Switzerland (2018)
5. Cai, Q., Filos-Ratsikas, A., Tang, P., Zhang, Y.: Reinforcement mechanism design for fraudulent behaviour in e-commerce. In: Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence (AAAI-18). pp. 957–964 (2018)
6. Chen, S., Tong, L., He, T.: Optimal deadline scheduling with commitment. In: 49th Annual Allerton Conference on Communication, Control, and Computing (Allerton), 2011. pp. 111–118. IEEE (2011)
7. Conitzer, V., Sandholm, T.: An algorithm for automatically designing deterministic mechanisms without payments. In: Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2004). pp. 128–135. IEEE Computer Society (2004)
8. Hernandez-Leal, P., Kaisers, M., Baarslag, T., de Cote, E.M.: A survey of learning in multiagent environments: Dealing with non-stationarity. CoRR (2017)
9. Huang, Z., Weinberg, S.M., Zheng, L., Joe-Wong, C., Chiang, M.: Discovering valuations and enforcing truthfulness in a deadline-aware scheduler. In: IEEE Conference on Computer Communications (INFOCOM 2017). pp. 1–9. IEEE (2017)
10. Jain, N., Menache, I., Naor, J.S., Yaniv, J.: Near-optimal scheduling mechanisms for deadline-sensitive jobs in large computing clusters. ACM Transactions on Parallel Computing **2**(1), 3 (2015)
11. Mao, H., Alizadeh, M., Menache, I., Kandula, S.: Resource management with deep reinforcement learning. In: Fifteenth ACM Workshop on Hot Topics in Networks (HotNets 2016). pp. 50–56 (2016)
12. Marinescu, D.C.: Cloud Computing: Theory and Practice. Morgan Kaufmann (2017)

13. Mashayekhy, L., Fisher, N., Grosu, D.: Truthful mechanisms for competitive reward-based scheduling. IEEE Transactions on Computers **65**(7), 2299–2312 (2016)
14. Nisan, N., Roughgarden, T., Tardos, E., Vazirani, V.V.: Algorithmic Game Theory. Cambridge University Press Cambridge (2007)
15. Parkes, D.C.: Online mechanisms. In: Nisan, N., Roughgarden, T., Tardos, E., Vazirani, V.V. (eds.) Algorithmic Game Theory, chap. 16, pp. 411–439. Cambridge University Press (2007)
16. Parkes, D.C., Singh, S.P.: An MDP-based approach to online mechanism design. In: Advances in Neural Information Processing Systems (NIPS 2004). pp. 791–798 (2004)
17. Pinedo, M.L.: Scheduling: Theory, Algorithms, and Systems. Springer (2016)
18. Porter, R.: Mechanism design for online real-time scheduling. In: Proceedings of the 5th ACM Conference on Electronic Commerce (EC 2004). pp. 61–70. ACM (2004)
19. Sandholm, T.: Automated mechanism design: A new application area for search algorithms. In: International Conference on Principles and Practice of Constraint Programming. pp. 19–36. Springer (2003)
20. Stein, S., Gerding, E., Robu, V., Jennings, N.R.: A model-based online mechanism with pre-commitment and its application to electric vehicle charging. In: Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2012). pp. 669–676 (2012)
21. Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction. MIT press Cambridge (1998)
22. Watkins, C.J., Dayan, P.: Q-learning. Machine Learning **8**(3-4), 279–292 (1992)
23. Wu, J., Xu, X., Zhang, P., Liu, C.: A novel multi-agent reinforcement learning approach for job scheduling in grid computing. Future Generation Computer Systems **27**(5), 430–439 (2011)
24. Zhang, W., Dietterich, T.G.: A reinforcement learning approach to job-shop scheduling. In: Proceedings of the 1995 International Joint Conference on AI (IJCAI 1995). vol. 95, pp. 1114–1120 (1995)

A Additional Pseudocode

In this appendix, we provide additional pseudocode for auxiliary functions in our resource allocation and reinforcement learning algorithms.

A.1 Checking for a Feasible Schedule

As shown in Algorithm 3, CHECK-FEASIBLE checks if a given task θ_i can be scheduled given the current permanent allocation decisions, π, and the set of admitted tasks, θ_{admitted}. The for-loop in Line 7 makes sure that all of the pre-committed tasks (with the additional θ_i) are included in the proposed schedule. If any of them are omitted, the function returns false. In algorithms without pre-commitments, θ_{admitted} is an empty set.

Algorithm 3 Checks if given θ_i can be scheduled

```
1:  θ_admitted                                          ▷ Admitted tasks
2:  π(i,t), ∀t,i                                        ▷ Current allocation decisions
3:
4:  procedure CHECK-FEASIBLE(θ_i, θ_admitted, π)
5:      θ_to_schedule ← θ_admitted ∪ {θ_i}
6:      π' ← SCHEDULE(π, θ_to_schedule)                 ▷ Feasible schedule
7:      for θ_j ∈ θ_to_schedule do
8:          if ∄t, π'(j,t) = 1 then
9:              return False
10:     return True
```

A.2 Schedule

The SCHEDULE action performs offline greedy on a given set of tasks, θ', and the current allocation decisions, π. Intuitively, the tasks are sorted and attempted to be scheduled on the slowest resources first one at a time. The order of tasks is given according to ρ (i.e., in descending order by value-density, v_i/q_i, in our case), and resources are sorted in ascending order according to their speed, given by ϱ. This is to maximise the allocation of resources whenever possible. In order to allocate a task, we must find an unoccupied gap in the current allocation for a particular resource that is large enough for the task to complete before its deadline or timeout without interruptions. We can restrict the search space for the gap by calculating the first and last possible time step when the task could be running, t_{first} and t_{last} respectively. We account for different resource speeds in line 10. Line 11 captures the current allocation decision for the resource. The function FIND-GAPS looks for gaps in the current allocation decisions which are between t_{first} and t_{last} time steps and at least task length l wide, and returns the time steps that indicate the beginning of these feasible gap. For better efficiency, we only calculate and take the earliest feasible start time (line 14). Lines 18-29 shows how FIND-GAPS(c)ould be implemented. Line 24 finds the time step of the next scheduled task, j_{next}.

Algorithm 4 Simple greedy approach to scheduling

1: t_{now} ▷ Current time step
2: T_{max} ▷ Time step horizon
3:
4: **procedure** SCHEDULE(π, θ')
5: $\pi' \leftarrow \pi$ ▷ Allocation decision copy
6: **for** $\theta_i \in \rho(\theta')$ **do**
7: **for** $r \in \varrho(R)$ **do**
8: $t_{\text{first}} \leftarrow max(a_i, t_{\text{now}})$ ▷ First possible time step
9: $t_{\text{last}} \leftarrow min(d_i, T_{\text{max}})$ ▷ Last possible time step
10: $l \leftarrow \lceil q_i / p_r \rceil$ ▷ Task length on resource r
11: $\pi'_r(j,t) \leftarrow \begin{cases} 1 & \pi'(j,t) = r, \forall t, j \\ 0 & otherwise \end{cases}$
12: $gaps \leftarrow$ FIND-GAPS$(t_{\text{first}}, t_{\text{last}}, l, \pi'_r)$
13: **if** $gaps \neq \emptyset$ **then**
14: $t_{\text{schedule}} \leftarrow min(gaps)$ ▷ Earliest feasible start
15: $\pi'(i, t_{\text{schedule}}) \leftarrow r$
16: **break**
return π'
17:
18: **procedure** FIND-GAPS($t_{\text{first}}, t_{\text{last}}, l, \pi'_r$)
19: **if** $\forall j, \sum_{t=t_{\text{first}}}^{t_{\text{last}}} \pi'_r(j,t) = 0$ **then** ▷ No allocations?
20: **if** $t_{\text{last}} - t_{\text{first}} \geq l$ **then** ▷ Wide enough?
21: **return** $\{t_{\text{first}}\}$
22: **else**
23: **return** $\emptyset$
24: $t_{\text{next}} \leftarrow min(\text{argmax}_{t \in \{t_{\text{first}}, \ldots, t_{\text{last}}\}} \pi'_r(j,t))$
25: $j_{\text{next}} \leftarrow \text{argmax}_j \, \pi'_r(j, t_{\text{next}})$
26: **if** $t_{\text{next}} - t_{\text{first}} \geq l$ **then** ▷ Wide enough?
27: **return** $\{t_{\text{first}}\}$
28: $t_{\text{first}} \leftarrow t_{\text{next}} + \lceil q_{j_{\text{next}}} / p_r \rceil$ ▷ Predicted end time of j_{next}
29: **return** FIND-GAPS$(t_{\text{first}}, t_{\text{last}}, l, \pi'_r)$

A.3 Updating Weights

Algorithm 5 shows how we update the weights with mini-batch stochastic gradient descent. Once we collect enough samples, we update the weights, $\mathbf{w}_0$ and $\mathbf{w}_1$ for admit and postpone actions respectively. We return unchanged weights the rest of the time (line 15). Once we have enough samples, we contract $\mathbf{X}$ which contains feature vectors of our batch samples. We also create an action vector, $\mathbf{a}$, and target value vector, $\mathbf{y}$ for all samples in the batch. Then, we repeat gradient descent step $iter$ number of times. Weights $\mathbf{w}_{0,0}$ and $\mathbf{w}_{1,0}$ refer to the bias terms and are updated without regularization, while $\mathbf{w}_{0,j}$ and $\mathbf{w}_{1,j}$ are weights for all the other features (where $j \geq 1$) and their updates include regularization. The novelty in our approach is in the weight constraints in lines 25-28. Here, we choose the weights corresponding to the deadline, size, value, and value density of a task, where idx_d, idx_q, idx_v and $idx_{v/q}$ correspond to indices of where these features appear in our state representation. The weights are subject to

those constraints at every gradient descent step. Note that this means that the algorithm may not necessarily converge to an optimal set of weights. This is a limitation of our approach, but it is required to ensure incentive compatibility.

A.4 Choosing an Action

Algorithm 6 shows a simple ϵ-greedy policy of picking actions. Procedure CHOOSE-ACTION gets the values of state-action value function and chooses the action that generates the greatest reward with ϵ-greedy policy.

Algorithm 5 Mini-batch stochastic gradient descent

m ▷ Number of samples in a batch
$batch$ ▷ batch placeholder
$counter$ ▷ sample counter
$iter$ ▷ Number of iterations
α ▷ Learning step size
λ ▷ Regularization factor
$\mathbf{w}_0, \mathbf{w}_1$ ▷ Initial weights
$idx_d, idx_q, idx_v, idx_{v/q}$ ▷ Indices of constrained weights

procedure UPDATE-WEIGHTS($\mathbf{s}_{\text{last}}, a, v, \mathbf{s}$)
  $\mathbf{s}_{\text{last}} \leftarrow$ NORMALISE($\mathbf{s}_{\text{last}}$)
  $batch \leftarrow batch \cup \{(\mathbf{s}_{\text{last}}, a, v)\}$
  $counter \leftarrow counter + 1$
  if $counter < m$ **then**
    return $\mathbf{w}_0, \mathbf{w}_1$ ▷ Return unchanged weights
  $\mathbf{X} \leftarrow [\mathbf{b}_0^1 ... \mathbf{b}_0^m], \mathbf{b}^k \in batch$
  $\mathbf{a} \leftarrow [\mathbf{b}_1^1 ... \mathbf{b}_1^m], \mathbf{b}^k \in batch$
  $\mathbf{y} \leftarrow [\mathbf{b}_2^1 ... \mathbf{b}_2^m], \mathbf{b}^k \in batch$
  $i \leftarrow 0$
  while $i < iter$ **do**
    $\mathbf{w}_{0,0} \leftarrow \mathbf{w}_{0,0} - \alpha \frac{1}{m} \sum_{k=1}^{m} (\mathbf{X}^k \mathbf{w}_0 - \mathbf{y}^k) \mathbf{X}_0^k * (1 - \mathbf{a}^k)$
    $\mathbf{w}_{1,0} \leftarrow \mathbf{w}_{1,0} - \alpha \frac{1}{m} \sum_{k=1}^{m} (\mathbf{X}^k \mathbf{w}_1 - \mathbf{y}^k) \mathbf{X}_0^k * \mathbf{a}^k$
    $\mathbf{w}_{0,j} \leftarrow \mathbf{w}_{0,j} - \alpha \frac{1}{m} \sum_{k=1}^{m} (\mathbf{X}^k \mathbf{w}_0 - \mathbf{y}^k) \mathbf{X}_j^k * (1 - \mathbf{a}^k) - \frac{\lambda}{m} \mathbf{w}_0, \forall j \geq 1$
    $\mathbf{w}_{1,j} \leftarrow \mathbf{w}_{1,j} - \alpha \frac{1}{m} \sum_{k=1}^{m} (\mathbf{X}^k \mathbf{w}_1 - \mathbf{y}^k) \mathbf{X}_j^k * \mathbf{a}^k - \frac{\lambda}{m} \mathbf{w}_1, \forall j \geq 1$
    $\mathbf{w}_{0,idx_d} \leftarrow min(\mathbf{w}_{0,idx_d}, \mathbf{w}_{1,idx_d})$
    $\mathbf{w}_{0,idx_q} \leftarrow max(\mathbf{w}_{0,idx_q}, \mathbf{w}_{1,idx_q})$
    $\mathbf{w}_{0,idx_v} \leftarrow min(\mathbf{w}_{0,idx_v}, \mathbf{w}_{1,idx_v})$
    $\mathbf{w}_{0,idx_{v/q}} \leftarrow min(\mathbf{w}_{0,idx_{v/q}}, \mathbf{w}_{1,idx_{v/q}})$
  $batch \leftarrow \emptyset$
  return $\mathbf{w}_0, \mathbf{w}_1$

Algorithm 6 Choose an action using ϵ-greedy approach

ϵ ▷ Exploration rate
$\Pi(q_0, q_1, \xi) \leftarrow \begin{cases} \text{pick action at random} & \text{if } \xi < \epsilon \\ \text{argmax}_{i \in \{0,1\}} \, q_i & \text{otherwise} \end{cases}$
procedure CHOOSE-ACTION(q_0, q_1)
  $\xi \sim U(0, 1)$ ▷ Generate a random number
  chosen $\leftarrow \Pi(q_0, q_1, \xi)$
  return chosen