Sequential Resource Allocation with Positional Costs

Bojun Huang

Microsoft Research, Beijing, China
bojhuang@microsoft.com

Abstract. We consider the problem of minimizing the total cost to run a sequence of n tasks in the given order by k agents under the *positional cost model*. The cost to run a task not only depends on the intrinsic cost of the task itself, but also monotonically related to the position this task is in the working list of the agent assigned. Such a positional effect can naturally arise from the classic sum-of-completion-time minimization problems, and is also well motivated by the varying *efficiency* when an agent works in reality (such as due to the learning effects or deteriorating effects). Also, it can be seen as a deterministic variant of the classic Baysian sequential decision making problems. This paper presents a simple and practical algorithm that runs in $O(k^2 n)$ time and minimizes the total cost of any problem instance consisting of two task types. The algorithm works by making greedy decision for each task *sequentially* based on some stopping thresholds in a "greedy-like" allocation simulation – a working style coinciding with Gittins' optimal-stopping based algorithm for the classic Baysian multi-armed bandit problem.

1 Introduction

Consider the problem of minimizing the *sum of completion time* to serve n customers with k identical service providers, where the customers are ordered by a *first-come-first-serve* queue so that any service provider must serve the customers assigned to it in the same order as the queue (but two customers may be served simultaneously if they are assigned to different providers). Suppose a customer i is the first customer of a provider that serves m customers in total, since all the other $m - 1$ customers of the same provider must wait for customer i to complete in time T_i, the completion time "caused by" customer i is thus $m \cdot T_i$. Summing up over all customers, we have

$$C_{total} = \sum_{i=1}^{n} f(p_i) \cdot T_i, \tag{1}$$

where p_i is the *position* of customer i in the working list of its service provider, and $f(p_i)$ equals to the *reversed position* of customer i, i.e., if a provider serves m customers in total, then $f(p_i) = m + 1 - p_i$, which equals to m for the first customer and 1 for the last customer, in particular.

The sum-of-completion-time problem presented above is a special case of the general problem of *Positional Allocation* studied in this paper. In general, we want to minimize the total cost to run a sequence of tasks by multiple agents under the *positional cost model* as Eq. 1 shows, where the cost to run a task not only depends on the independent cost of the task but also *monotonically* related to the number of tasks the agent has been assigned to before. The sum-of-completion-time problem already shows how such a positional effect can be naturally derived from the optimization objective of minimizing total completion time. Moreover, the positional cost may alternatively come from an abstraction of the varying *efficiency* when an agent works in reality. For example, the so-called *learning effect* [2] [5] usually helps a human agent to work more and more efficiently, while on the other hand various *deteriorating effects* [4] may just do the opposite (e.g. human/animals can get tired during working, devices can wear off during the usage, or the working situation is just getting worse over time in medical treatment and diseaster rescue). See Appendix A for a motivating application for the authors, where the goal is to optimize the overall battery efficiency in electrical systems powered by multiple batteries, in light of the phenomenon that the power efficiency of battery gets worse as the battery is discharging.

In cases where the tasks can be run in arbitrary order, we know from the *rearrangement inequalities* by Hardy et al. [13] that the problem can be solved by a simple *Shortest-Processing-Time* (SPT) rule that always matches the task with the shortest processing time to the position with the largest weight. However, when the *non-reorderable* cosntraint is imposed, which could be either caused by a priority of the tasks (such as in a queue) or by the online nature of the problem, the greedy SPT rule becomes suboptimal (see Figure 3 in Appendix B for a counter-example), and there seems to be no obvious way to solve the problem in polynomial time.

In this paper we propose a simple but nontrivial algorithm that runs in $O(k^2 n)$ time, and we show that this algorithm is optimal under any problem instance with two task-types. The algorithm works by making greedy decision for each task *sequentially* based on stopping thresholds in a "greedy-like" allocation simulation. We expect the combinatorial structures of the problem exhibited by our algorithm can inspire the design of practical and optimal algorithms in more general settings of this important problem.

1.1 Connections with Related Work

Both sum-of-completion-time (or, the *mean flow time*) and max-of-completion-time (i.e. the *makespan*) are extensively studied optimization objectives. It is widely known that the makespan optimization problem is strongly NP-hard even assuming a constant positional function $f(p) \equiv 1$, both for its uncapacitated version (i.e., multiprocessor scheduling/bin packing) and capacitated version (i.e., the 3-partition problem). Many studies were thus focusing on designing asymptotic-PTAS or constant-ratio approximation algorithms for makespan optimizations, especially under generalized cost functions [1] [8] [15] [7]. The

objective functions of most these generalized cost models are *symmetric* with respect to the tasks/items, and thus the order of items has no impact on the aggregate value. In contrast, in the positional allocation problem the "real cost" of an item further depends on *where* it is put in the bin.

Meanwhiles, another line of research tries to find polynomial-time exact algorithms for the makespan optimization problem assuming a constant number of task/item types. Specifically, Leung [14] presented an $O(n^{2c-2} \log k)$ dynamic programming algorithm, where c is the number of item types. However, since problems in such a setting can admit compact inputs (in fact, encoded by only $2c$ numbers), the running time of a polynomial algorithm for such *high-multiplicity* problems needs to be polynomial to $\log n$ (rather than to n). A polynomial algorithm for the case $c = 2$ was first given by McCormick, Smallwood and Spieksma in [16]. Later, Eisenbrand and Shmonin [6] showed that actually only 3 different "packing ways" are needed for the case of $c = 2$. Very recently, Goemans and Rothvoß [11] extended the techniques in [6] and gave the first polynomial-time bin packing algorithm for arbitrary (but constant) number of item types. Our work pursuits similar goals with these works, trying to find polynomial-time algorithms for problem instances with constant number of task/item types, albeit with a different optimization objective encompassing flow-time. Essentially, the metrics of makespan and flow-time correspond to the L_{inf} norm and L_1 norm, respectively (see Eq. 2 in Section 2). Besides, also note that the order of tasks in the input sequence plays a crucial role in the positional allocation problem considered in this paper, which means binary instances of this problem cannot be compressed into a sequence of multiplicities, but will have the same input format as the general form of the problem, thus having $O(n)$ input length.

On the other hand, scheduling under positional costs is also an active area in operations research. Biskup and others [2] [5] first considered the learning effects in single-machine scheduling, in which the positional weights decrease with the positions, typically modeled by explicit polynomial functions in most later works [17]. See [3] for a survey of them. Browne and Yechiali [4] first introduced the deteriorating effects in scheduling problems, in which the positional weights increase with the positions, typically modeled by explicit polynomial functions [18] or exponential functions [12]. See [20] for a survey of works in this line. Besides, some works also considered the parallelel machine scheduling problems with positional costs [19] [20]. In most of the positional scheduling works presented above, the key is to find good *permutations* of the task sequence so as to minimize the objectives considered, and the classic Short-Processing-Time-first (SPT) rule turns out to be optimal in various settings (e.g. see the summary table in [3]). Differently, the positional allocation problem considered here imposes a strict *nonorderable* constraint on the order of tasks. Note that the constraint is different from classic *precedence constriants* in that, the former only constrains the *order* of tasks in the same machine (but it is possible to run tasks in parallel in different machines) while the latter further rules out any parallelism between tasks with precedence relationship. It turns out that the nonorderable constraint invalidates the mostly-used SPT rule. Actually, as

demonstrated later, the positional allocation problem studied in this paper exhibits a quite different combinatorial structure, which leads to practical optimal algorithms quite different with the SPT rule.

Finally, there is an interesting connection between the positional allocation problem and the *Baysian Multi-Armed Bandit* (MAB) problem . In the Baysian MAB problem, we are given k "bandit-arms", and each arm r is in an observable "state" $Q_r^{(t)}$ at time-slot t. In any round t, we are asked to choose one arm x_r, leading to a stochastic payoff $\alpha^t \cdot R(Q_{x_t}^{(t)})$, and also causing the chosen arm x_t to stochastically change its state to $Q_{x_t}^{(t+1)} = P(Q_{x_t}^{(t)})$. Actually, the function R corresponds to a parameterized probability distribution of payoff, the state $Q_r^{(t)}$ corresponds to the parameter setting of R that the player "believes" the arm r "should be in" at time t, and the state transition function P follows the Baysian inference principle for probability distribution R. The goal is to maximize the cumulative reward at infinite horizon. It is not hard to see that the Baysian MAB problem is essentially a stochastic version of the positional allocation problem, with a special task sequence $\{T_t = \alpha^t\}$ on one hand, while with general stochastic transition functions on the other hand (in the positional allocation problem, $P(Q_r^{(t)}) = Q_r^{(t)} + 1$, and $R(Q_r^{(t)}) = 0$ when $Q_r^{(t)} > m$). In 1979, Gittins found an elegant *simulation-based* algorithm [10], which is proven to be optimal for the Baysian MAB problem [9]. Describing in our language, given an infinite task sequence $\alpha^1, \alpha^2, \alpha^3, \ldots$, where $0 < \alpha < 1$, the algorithm assigns a score V_r to each agent r according *solely* to the current capacity m_r of that agent (i.e., V_r is independent to any other $m_{r' \neq r}$), then the algorithm simply allocates the first task T_1 to the agent with the highest score. The score, later called the *Gittins Index*, happens to correspond to the (expected) normalized cumulative cost of the *optimal stopping strategy* of a simulation to keep allocating tasks in the single machine r. Note that the task sequence in Baysian MAB is by default sorted in a Shortest-Processing-Time-first manner [1], and due to the monotonicity of the positional function, the Gittins-Index-based algorithm degenerates to the naive SPT algorithm in the positional allocation problem, which is known to be suboptimal in general (but indeed optimal for that specific single task sequence!). Interestingly, as shown in this paper, it turns out that the truely optimal algorithm for the positional allocation problem may still exhibit a very similar working pattern with the Gittins' algorithm (at least for binary inputs), namely that the optimal decision for each task can be made *sequentially* based on the "stopping threshold" of a "greedy-like" allocation simulation.

2 Preliminaries

Definition 1. *A k-allocation scheme of n tasks is a partitioning of the sequence $1, 2, \ldots, n$ into k subsequences, denoted by $\mathbf{A}_{n,k} = (\mathbf{a}^1, \mathbf{a}^2, \ldots, \mathbf{a}^k)$, where (1)*

---

[1] More accurately, the Baysian MAB problem, as a reward maximization problem, always receives exponentially decreasing inputs, which translates to an increasing (SPT-first) task sequence in its dual problem concerning cost minimization.

$a_i^r \neq a_j^s$ if $(r, i) \neq (s, j)$ *(disjointness)*; (2) $\sum_{r=1}^{k} |\mathbf{a}^r| = n$ *(completeness)*; (3) $a_i^r < a_j^r$ *if $i < j$, for any $1 \leq r \leq k$ (monotonicity)*.

The *size* of an allocation scheme $\mathbf{A}_{n,k}$ is m if $|\mathbf{a}^r| = m$ for every $\mathbf{a}^r \in A_{n,k}$. Note that only an allocation scheme with uniform cardinality has a well-defined size. The *position* of task t under allocation scheme $\mathbf{A}_{n,k}$ is p_t if there exists $r \in [k]$ such that $a_{p_t}^r = t$. Given an allocation scheme $\mathbf{A}_{n,k}$, any integer sequence $T_1, T_2, \ldots, T_n$ can be accordingly partitioned into k subsequences, denoted by $\mathbf{T}_{n,k} = (\{T_{a_p^1}\}, \{T_{a_p^2}\}, \ldots, \{T_{a_p^k}\})$. For convenience we will write $T_{r,p}$ for $T_{a_p^r}$ when the context is clear.

Positional Allocation Problem. Given a problem instance $\mathbf{I} = (n, k, m, f, T_1, \ldots, T_n)$ where $n = km$, $T_i \in \mathbb{N}$, and $f : \mathbb{N} \mapsto \mathbb{N}$ is arbitrary monotonically decreasing function. We want to find an allocation scheme $A_{n,k}$ of size m in order to

$$
\min_{\mathbf{A}_{n,k}} \left\| \begin{pmatrix} T_{1,1} \ T_{1,2} \ldots T_{1,m} \\ \vdots \ \ \vdots \ \ \vdots \\ T_{k,1} \ T_{k,2} \ldots T_{k,m} \end{pmatrix} \cdot \begin{pmatrix} f(1) \\ f(2) \\ \vdots \\ f(m) \end{pmatrix} \right\|_1 \tag{2}
$$

In the above problem formulation, a positional allocation algorithm outputs an allocation scheme $\mathbf{A}_{n,k}$ for a given problem instance. Equivalently, a positional allocation algorithm may output a *decision sequence* $\mathbf{x} = (x_1, \ldots, x_n)$ where each $x_t \in [k]$ denotes the index of the agent assigned to the task t. Clearly, any decision sequence corresponds to a unique allocation scheme. The following lemma shows that the reverse is also true: any given allocation scheme also corresponds to a unique decision sequence. In other words, an allocation scheme is equivalent to a decision sequence. See the proof in Appendix C.1.

Lemma 1. *For any allocation scheme $\mathbf{A}_{n,k} = (\mathbf{a}^1, \mathbf{a}^2, \ldots, \mathbf{a}^k)$, there exists a unique decision sequence $\mathbf{x} = (x_1, \ldots, x_n)$ such that $a_{p_t}^{x_t} = t$ for each $1 \leq t \leq n$.*

In the rest of this paper we will mainly discuss algorithms assuming they output decision sequences, and as a general technique, we often prove the suboptimality of a given decision sequence $\mathbf{x}$ by re-arranging some tasks in the allocation scheme corresponding to $\mathbf{x}$ and showing that the re-arranged allocation scheme has lower cost than the original one. In such a proof, the monotonicity property of allocation scheme is the key to guarantee that the re-arranged allocation scheme is still valid (i.e. "achievable" by some decision sequence).

We remark that our problem formulation encompasses some other related models. For example, although in our formulation each agent must be assigned *exactly* n/k tasks, both the problem variants with and without cardinality constraints (in which an agent r can run *at most* m tasks and *arbitrary* number of tasks, respectively) can be reduced to the problem formulated here by appending enough number of "null tasks" with $T_i = 0$. Furthermore, although our formulation assumes the positional cost function f is decreasing, a problem instance with increasing function can be reduced to an instance in our model by reversing the

task sequence. Specifically, for any instance $\mathbf{I} = (n, k, m, g, T_1, T_2, \ldots, T_n)$ with increasing function g, we can construct an instance $\mathbf{I}' = (n, k, m, f, T_n, \ldots, T_2, T_1)$ with the decreasing function $f(p) = g(m + 1 - p)$, and from Lemma 1 we know that: if $\mathbf{x}' = (x_1', x_2', \ldots, x_n')$ is the solution of $\mathbf{I}'$ in our model, then $\mathbf{x} = (x_n', \ldots, x_2', x_1')$ is the solution of I in the model with increasing positional function (and vise versa). Similarly, the same reduction also works for the problem variant with decreasing positional function but reversely-growing positional index (from m back to 1). Also note that the tricks presented above can be further combined together to reduce more combinations of variants to our problem. For example, let $\mathbf{I} = (n, k, T_1, T_2, \ldots, T_n)$ be an instance of the sum-of-completion-time problem presented at the beginning of the paper. The "equivalent instance" of $\mathbf{I}$ in our model is $\mathbf{I}' = (n, k, n, f, T_n, \ldots, T_2, T_1, 0, 0, \ldots, 0)$ where there are $(kn - n)$ "null tasks" in $\mathbf{I}'$ and $f(p) = n + 1 - p$.

Finally, the current probem formulation is presented in a form for the sake of simplicity, and the algorithms presented in this paper may apply to some natural generalizations of the problem. For example, in our formulation every agent is assigned with the same number of tasks, while our algorithmic results also apply to problems with arbitrary capacity plan $(m_1, m_2, \ldots, m_k)$ in which agent r may be assigned with (exactly or at most) a different number of m_r tasks. Besides, in this paper we couple the position weights and task-specific costs with the multiplication operator (see Eq. 1 and Eq. 2), while the algorithmic discussions also apply to the more general cases where the cost function $f(p_i, T_i)$ is monotone and has positive mixed partial derivaties $\frac{\partial^2 f}{\partial p \partial T} = \frac{\partial^2 f}{\partial T \partial p} > 0$.

The Shortest-Processing-Time Rule. The monotonicity of the positional cost function implies a simple "principle" to allocate tasks: In general, we tend to run tasks with relatively smaller costs first (thus coupled with larger positional weights) while to allocate tasks with relatively larger costs later (thus coupled with smaller positional weights). Lemma 2 justifies this intuition formally.

Lemma 2. *(Rearrangement Inequality [13]) If a monotonically increasing function $f(Q, T) : \mathbb{R}_{\geq 0} \times \mathbb{R}_{\geq 0} \mapsto \mathbb{R}_{\geq 0}$ has positive mixed derivatives $\frac{\partial^2 f(Q,T)}{\partial Q \partial T} = \frac{\partial^2 f(Q,T)}{\partial T \partial Q} > 0$, then for any $Q_1 \geq Q_2$, and $T_1 \geq T_2$, we have*

$$f(Q_1, T_1) + f(Q_2, T_2) \geq f(Q_1, T_2) + f(Q_2, T_1) \tag{3}$$

The above principle suggests that, if we *could* arbitrarily change the order of the tasks, the optimal algorithm would be simply to sort the tasks in ascending order and to assign them among the agents in a round-robin way. However, the nonreorderable constraint of the problem brings additional difficulty that we are forced to allocate tasks sequentially. In this case, a naive greedy allocation algorithm may be to couple a large (small) task with the smallest (largest) positional weight *at that time*. This naive greedy algorithm turns out to be suboptimal. For example, Figure 3 in the appendix shows an instance with binary costs ($k = 4$ and $n = 80$), where the allocation scheme of the naive greedy algorithm (the left side) has larger cost than another allocation scheme (the

right side). Actually, the better allocation scheme at the right side is optimal under this instance, and it comes from a simple and efficient algorithm proposed in the next section. In Section 4 we prove that this algorithm minimizes the total cost for any binary-valued instance of the positional allocation problem.

3   The Algorithm

We assume $T_i \in \{H, L\}$, $0 \leq H < L$ in this section. A pragmatic motivation of the assumption is that, often in real world the task-specific costs follow a *bi-modal* distribution, in which case a two-value separation may well approximate the real values. For example, in the multi-battery application presented in Appendix A, the power consumption of a device can greatly depend on whether the device is active or on standby, with a huge gap of more than 100x (See Figure **??**). Moreover, detecting the active-standby mode of a device is usually much more efficient, robust, and easier than measuring the exact value of the power consumption of the device.

A straightforward k-dimensional dynamic programming procedure can solve the positional scheduling problem. Specifically, define $\mathbf{m} = (m_1, m_2, \ldots, m_k)$ as the state vector for the situation where there are m_r "slots" left in each agent r. Given a task sequence $\mathbf{T} = (T_1, T_2, \ldots, T_{\sum m_r})$, define $COST_{\mathbf{T}}(\mathbf{m})$ as the minimum total cost over all possible decision sequences that matches $\mathbf{T}$ to $\mathbf{m}$. By definition of the positional allocation problem we have

$$COST_{\mathbf{T}}(\mathbf{m}) = \min_{r \in [k]} \left\{ \, COST_{\mathbf{T} \setminus T_1}(m_1, \ldots, m_r - 1, \ldots, m_k) + f(m - m_r) \cdot T_1 \, \right\}.$$

Since the size of the state space is no more than n^k, the time complexity of the dynamic programming procedure is polynomial to n for constant k. For general k, however, the size of the state space is at least the *integer partition function* $p(\frac{n}{2}) = 2^{\Theta(\sqrt{n})}$ (e.g., when $k = \frac{n}{2}$), which is super-polynomial to the input size. Meanwhiles, another drawback of the dynamic programming solution is that it may not easily adapte to the cases where the information of the tasks is limited, such as in online allocation scenarios where we (at best) only know a stochastic generating process of the workload, rather than a deterministic task sequence.

In this section, we present a simple algorithm that turns out to generate the optimal decision sequence for any problem instance with binary-valued task sequence. The basic version of the algorithm is shown in Algorithm 1. Again we use $(m_1, m_2, \ldots, m_k)$ to denote the cardinality capacities of agents, and without loss of generality assume $m_1 \leq m_2 \leq \cdots \leq m_k$. To allocate the first task T_1, the algorithm iterates over each resource r in the descending order (priority) to decide whether to put T_1 in r. The decision for each r is made via doing a "simulated allocation" as follows (Line $12 \sim 21$ in Algorithm 1): the simulation first setups a *L-zone* and a *H-zone* among all the available positions, then sequentially allocates the task sequence, starting from T_1, by sending all "large" tasks (i.e. the ones with $T_i > T_1$) in the L-zone and all "small" tasks (the ones with $T_i \leq T_1$) in the H-zone. Whenever the "small" tasks overflow from the

---
Algorithm 1: The basic version of the simulation-based algorithm
---
Input: $m_1 \ldots m_k, T_1 \ldots T_n$, where $m_1 \leq m_2 \leq \cdots \leq m_k$
Output: $x_1 \ldots x_n$

1 set $Q_r^{(1)} = m_r$ for each $1 \leq r \leq k$
2 for $t = 1$ **to** n **do**
3 $\quad$ $x_t \leftarrow$ `ThresholdAllocation` $(Q_1^{(t)} \ldots Q_k^{(t)}, T_t \ldots T_n)$
4 $\quad$ $Q_r^{(t+1)} \leftarrow Q_r^{(t)} - \mathbb{1}(x_t = r)$ $\quad$ for each $1 \leq r \leq k$
5 end
6 return x

7 Function `ThresholdAllocation`
Input: $m_1, \ldots, m_k, T_1, \ldots, T_n$
Output: the agent to which T_1 is assigned
8 for $\gamma = k$ **to** 1 **do**
9 $\quad$ **if** $m_\gamma = m_{\gamma-1}$ **then**
10 $\quad$ $\quad$ **continue**
11 $\quad$ **end**
12 $\quad$ **for** $h = \gamma$ **to** k **do**
13 $\quad$ $\quad$ $S_L = \{1, 2, \ldots, h-1\}$
14 $\quad$ $\quad$ $S_H = \{\gamma, \gamma+1, \ldots, h\}$
15 $\quad$ $\quad$ $Z_L \leftarrow \sum_{i \in S_L} \min\{m_i, m_{\gamma-1}\}$
16 $\quad$ $\quad$ $Z_H \leftarrow \sum_{i \in S_H}(m_i - m_{\gamma-1})$
17 $\quad$ $\quad$ $\mathcal{L} = \{i : i \in \{1, \ldots, Z_L + Z_H\}$ and $T_i > T_1\}$
18 $\quad$ $\quad$ **if** $|\mathcal{L}| \geq Z_L$ **then**
19 $\quad$ $\quad$ $\quad$ **return** γ
20 $\quad$ $\quad$ **end**
21 $\quad$ **end**
22 end

H-zone, both the H-zone and the L-zone expand, and the simulated allocation continues. If at any time in the simulation the L-zone is filled up by "large" tasks (is full), the algorithm immediately stops the simulation and allocates T_1 to agent r; otherwise it will choose some other agent with smaller id in the later simulations. Note that the algorithm will guarantee to choose agent 1 (which has the smallest capacity) if getting chance to run simulation on it (i.e. with $r = 1$). Also note that the task with the largest cost $\max T_i$ (i.e. L in this context) will always be assigned to the smallest non-empty agent.

The initialization and expansions of H-zone and L-zone are based on the current capacities of agents $(Q_1^{(t)}, \ldots, Q_k^{(t)})$ and the index r of the (agent) candidate on which the simulation focuses. Let p^* be the "current position" of agent $r-1$ (so $p^* = m + 1 - Q_{r-1}^{(t)}$). At any time of the simulation the H-zone only contains positions smaller than p^* while the L-zone only contains positions equal or larger than p^*. Initially, the H-zone contains all such legal positions for the agent r (i.e., the single position of p^* in agent r), and the L-zone contains all
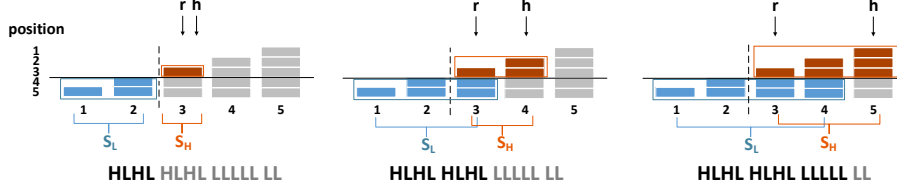
Fig. 1. Illustration of a simulation process performed by the ThresholdAllocation routine of Algorithm 1 for $r = 3$ under the task sequence $HLHL\ HLHL\ LLLLL\ LL$ ($n = 15$, $k = 5$). The three diagrams correspond to the initial setup and two later expasions of H-zone (positions in red) and L-zone (positions in blue) in the simulation, respectively. In particular, when $h = 5$ (the right side), the H-zone is of size $Z_H = 6$ and the L-zone is of size $Z_L = 7$. Since there are at least 7 "L" tasks in the first $7 + 6 = 13$ tasks, the algorithm will assign the first task to agent $r = 3$.

legal positions for agents smaller than r. In every expansion, both the H-zone and L-zone include legal positions of one more agent, from $r + 1$ to k. Figure 1 illustrates an example simulation process with five agents and $r = 3$.

Given a H-zone and L-zone setup, to check whether the L-zone can be filled up before the H-zone overflows, the algorithm only needs to count whether the number of "large" tasks within a look-ahead "window" exceeds a "threshold". Specifically, Let Z_H and Z_L be the sizes of H-zone and L-zone respectively, the L-zone can be "successfully" filled up if and only if there are at least Z_L "large" tasks in the first $Z_H + Z_L$ tasks. Suppose task t is the head of the current task sequence, the last task to check in the look-ahead will be $Z_L + Z_H + t - 1$, which will be called the *look-ahead horizon from task t under setup* (r, h) in the rest of the paper, or just *horizon* when the context is clear.

Note that the output of Algorithm 1 does not depend on the specific form of the positional function $f(p)$ at all, nor on the specific values of the task T_i, nor even on the detailed pattern of how the tasks are arranged within each look-ahead window. Moreover, the following lemmas establish some monotonicity properties of Algorithm 1. Specifically, Lemma 3 asserts that the algorithm never changes the order of the capacities of agents, i.e., the monotonicity of agent-capacities is conserved; Lemma 4 asserts that the look-ahead horizons under any given setup (r, h) always move forward in the direction from 1 to n, i.e., the monotonicity of horizon for given look-ahead setup is conserved; and Lemma 5 asserts that the allocation decisions for the same task type always move forward in the direction from agent 1 to k, i.e. the monotonicity of agent id's assigned to given task-type is conserved.

Lemma 3. *Under any problem instance $I = (m_1 \ldots m_k, T_1 \ldots T_k)$ with two types of task, $m_1 \leq m_2 \leq \cdots \leq m_k$, for the agent-capacity variables $\{Q_r^{(t)}\}$ computed by Algorithm 1 under I, we have $Q_r^{(t)} \leq Q_s^{(t)}$ if $r < s$, for any $1 \leq t \leq n$.*

Lemma 4. *Under any problem instance $I = (m_1 \ldots m_k, T_1 \ldots T_k)$ with two types of task, $m_1 \leq m_2 \leq \cdots \leq m_k$, for any given look-ahead setup (r, h),*

$1 \le r \le h \le k$, let $Z_H^{(r,h,i)}$ and $Z_L^{(r,h,i)}$ be the sizes of H-zone and L-zone (respectively) when Algorithm 1 is allocating task i under setup (r,h), and let $Z_H^{(r,h,j)}$ and $Z_L^{(r,h,j)}$ be the sizes of H-zone and L-zone (respectively) when Algorithm 1 is allocating task j under the same setup (r,h), we have

$$Z_H^{(r,h,i)} + Z_L^{(r,h,i)} + i - 1 \le Z_H^{(r,h,j)} + Z_L^{(r,h,j)} + j - 1 \quad if \quad i < j. \quad (4)$$

Lemma 5. *Under any problem instance* $I = (m_1 \ldots m_k, T_1 \ldots T_k)$ *with two types of task,* $m_1 \le m_2 \le \cdots \le m_k$*, let* i *and* j *be two tasks of the same type (i.e.* $T_i = T_j$*), and let* $(x_1, \ldots, x_n)$ *be the decision sequence output by Algorithm 1 under* I*, we have* $x_i \le x_j$ *if* $i < j$*.*

In particular, thanks to Lemma 4, all data variables (the horizons, thresholds, and counters) used by Algorithm 1 can be updated incrementally, yielding a more efficient implementation as shown by Algorithm 3 in Appendix D. It is easy to see that $O(k^2 n)$ time and $O(k^2)$ space is sufficient for Algorithm 3 to compute the same decision sequence with Algorithm 1 for any problem instance with two types of task.

Theorem 1. *Under any problem instance* $I = (m_1 \ldots m_k, T_1 \ldots T_k)$*,* $T_i \in \{H, L\}$ *for* $1 \le i \le n$*,* $0 \le H < L$*, Algorithm 3 returns the same decision sequence with Algorithm 1 in* $O(n \cdot k^2)$ *time and with* $O(k^2)$ *space.*

4 Optimality of Algorithm 1

In this section we prove that Algorithm 1 minimizes the total cost of any problem instance of positional allocation with two types of task. The basic idea is to show that if at any time we don't assign a task t to the agent decided by Algorithm 1, from the resulting decision sequence $\mathbf{x}$ we can always construct another sequence $\mathbf{x}^*$ such that $\mathbf{x}^*$ follows Algorithm 1 on t and has less total cost than $\mathbf{x}$. The construction is by re-arranging some tasks in the corresponding allocation scheme of $\mathbf{x}$ without violating the monotonicity property of Definition 1.

Since Algorithm 1 runs in an iterative manner, we only prove this for the first task T_1. In order to be consistent with the input of Function ThresholdAllocation of Algorithm 1, we slightly generalize the problem formulation to allow variable-sized agents in this section as follows:

Variable-Sized Position Allocation (VSPA) Problem. Given a problem instance $\mathbf{I} = (m_1, \ldots, m_k, T_1, \ldots, T_n)$, where T_i is the intrinsic-cost of task i and m_r is the cardinality capacity of agent r, without loss of generality assume all agents are "non-empty" at the beginning, and is sorted increasingly in capacity, that is, $0 < m_1 \le m_2 \cdots \le m_k$. A valid allocation scheme $\mathbf{A} = (\mathbf{a}^1, \ldots, \mathbf{a}^k)$ is required to be consistent with the cardinality constraints, that is, $|\mathbf{a}^r| = m_r$ for each $r \in [k]$. The allocation scheme is indexed by the capacity of the agent when the task is assigned to (i.e. the reversed position). Instead of directly applying the positional function $f(p)$, we define

$$g(q) = f(m + 1 - q). \quad (5)$$

Assigning task i to agent r of capacity m_r causes a cost of $g(m_r) \cdot T_i$. Again, every decision sequence $\mathbf{x}$ corresponds to a unique partitioning of the task sequence $\{T_{r,q}\}$, and the goal is to minimize the total cost defined by

$$COST_{\mathbf{I}}(\mathbf{x}) = \sum_{r=1}^{k} \sum_{q=1}^{m_r} g(q) \cdot T_{r,q}. \tag{6}$$

One can verify that the VSPA problem is exactly the original positional allocation problem when $m_1 = m_2 = \cdots = m_k = m$.

Now we will prove the optimality of Algorithm 1 for VSPA instances with two types of task. The main part of the proof is separated in cases of $T_1 = L$ (Lemma 6) and $T_1 = H$ (Lemma 7). In the former case, a straightforward re-arrangment can be done by observing the fact that Algorithm 1 always assigns a L task to the "smallest" agent (i.e., in agent 1). See the proof in Appendix C.4.

Lemma 6. *If $I = (m_1 \ldots m_k, T_1 \ldots T_n)$ is an instance of the VSPA problem with $T_i \in \{H, L\}$, $0 \le H < L$, $0 < m_1 \le m_2 \le \cdots \le m_k$, and $T_1 = L$, then for any decision sequence $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ with $x_1 > 1$, there exists another decision sequence $\mathbf{x}^* = (1, x_2^*, \ldots, x_n^*)$ such that $COST_I(\mathbf{x}^*) \le COST_I(\mathbf{x})$.*

The proof for cases with $T_1 = H$ (Lemma 7) is by induction. Specifically, assume by induction that Algorithm 1 is optimal for the subsequent tasks $T_2, \ldots, T_n$, which yields a specific sequence $\mathbf{x}$ that is guaranteed to be optimal among the set of decision sequences *not* following Algorithm 1 at T_1. We will show that the specific pattern of $\mathbf{x}$ always enables a re-arrangement to beat itself, and thus beat any sequence not following Algorithm 1 at T_1. More specifically, suppose Algorithm 1 assigns T_1 to agent γ^*, there can be two ways to *not follow* this decision: i) to allocate T_1 "lower", in some agent $r < \gamma^*$; or ii) to allocate T_1 "higher", in some agent $r > \gamma^*$. The constructions of better decision sequences are further separated into these two cases. See the proof in Appendix C.5.

Lemma 7. *Suppose $I = (m_1 \ldots m_k, T_1 \ldots T_n)$ is an problem instance with $T_i \in \{H, L\}$, $0 \le H < L$, $0 < m_1 \le m_2 \le \cdots \le m_k$, and $T_1 = H$, and suppose Algorithm 1 assigns task 1 to agent γ^* under I, then for any decision sequence $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ with $x_1 \ne \gamma^*$, there exists another decision sequence $\mathbf{x}^* = (\gamma^*, x_2^* \ldots, x_n^*)$ such that $COST_I(\mathbf{x}^*) \le COST_I(\mathbf{x})$.*

Finally, Theorem 2 combines results proved in Lemma 6 and Lemma 7 to complete the proof of the instance-optimality of Algorithm under binary instances consisting of H and L.

Theorem 2. *Algorithm 1 minimizes the total cost defined by Eq. 2 for any instance of the positional allocation problem with two types of task.*

References

1. Shoshana Anily, Julien Bramel, and David Simchi-Levi. Worst-case analysis of heuristics for the bin packing problem with general cost structures. *Operations research*, 42(2):287–298, 1994.

2. Dirk Biskup. Single-machine scheduling with learning considerations. *European Journal of Operational Research*, 115(1):173–178, 1999.

3. Dirk Biskup. A state-of-the-art review on scheduling with learning effects. *European Journal of Operational Research*, 188(2):315–329, 2008.

4. Sid Browne and Uri Yechiali. Scheduling deteriorating jobs on a single processor. *Operations Research*, 38(3):495–498, 1990.

5. TC Edwin Cheng and Guoqing Wang. Single machine scheduling with learning effect considerations. *Annals of Operations Research*, 98(1-4):273–290, 2000.

6. Friedrich Eisenbrand and Gennady Shmonin. Carathéodory bounds for integer cones. *Operations Research Letters*, 34(5):564–568, 2006.

7. Leah Epstein and Asaf Levin. An aptas for generalized cost variable-sized bin packing. *SIAM Journal on Computing*, 38(1):411–428, 2008.

8. Leah Epstein and Asaf Levin. Bin packing with general cost structures. *Mathematical programming*, 132(1-2):355–391, 2012.

9. Esther Frostig and Gideon Weiss. Four proofs of gittins multiarmed bandit theorem. *Annals of Operations Research*, pages 1–39, 1999.

10. John C Gittins. Bandit processes and dynamic allocation indices. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 148–177, 1979.

11. Michel X Goemans and Thomas Rothvoß. Polynomiality for bin packing with a constant number of item types. In *Proceedings of the twenty-fifth annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics, 2014.

12. Valery S Gordon, Chris N Potts, Vitaly A Strusevich, and J Douglass Whitehead. Single machine scheduling models with deterioration and learning: handling precedence constraints via priority generation. *Journal of Scheduling*, 11(5):357–370, 2008.

13. Godfrey Harold Hardy, John Edensor Littlewood, and George Pólya. *Inequalities*. Cambridge university press, 1952.

14. Joseph Y-T Leung. On scheduling independent tasks with restricted execution times. *Operations Research*, 30(1):163–171, 1982.

15. Chung-Lun Li and Zhi-Long Chen. Bin-packing problem with concave costs of bin utilization. *Naval Research Logistics (NRL)*, 53(4):298–308, 2006.

16. S Thomas McCormick, Scott R Smallwood, and Frits CR Spieksma. Polynomial algorithms for multiprocessor scheduling with a small number of job lengths. In *Proceedings of the eighth annual ACM-SIAM symposium on Discrete algorithms*, pages 509–517. Society for Industrial and Applied Mathematics, 1997.

17. Gur Mosheiov. Scheduling problems with a learning effect. *European Journal of Operational Research*, 132(3):687–693, 2001.

18. Gur Mosheiov. A note on scheduling deteriorating jobs. *Mathematical and Computer Modelling*, 41(8):883–886, 2005.

19. Gur Mosheiov et al. Parallel machine scheduling with a learning effect. *Journal of the Operational Research Society*, 52(10):1165–1169, 2001.

20. Kabir Rustogi and Vitaly A Strusevich. Simple matching vs linear assignment in scheduling models with positional effects: A critical review. *European Journal of Operational Research*, 222(3):393–407, 2012.

A The Multi-Battery Problem

Energy-efficiency is a key concern on mobile devices that depend on batteries to provide the power required to maintain operation. Whenever a device draws power, not all energy that is drawn from the battery is actually useful in the sense that it ends up powering the device. In fact, only a fraction of the energy drained from the battery ends up powering the device, the remainder is wasted: it heats up the device. We call these two components of the energy drained from a battery with each load the *useful energy* and the *wasted energy*, respectively. There are two main factors that determine the amount of wasted energy in the battery in a given time duration: the power consumption of the current user load and the *internal resistance* of the battery. In turn, one of the key factors determining internal resistance is the *State-of-Charge* (SoC) of the battery, i.e., how much remaining charge is in the battery. The quantatitive relationship between the wasted energy E_{waste}, the power of load p, and the SoC of battery s can be approximated by the following formula:

$$E_{waste} = T \cdot \left(1 - \sqrt{1 - \frac{2r}{V_{oc}} \cdot p} \right) / \frac{2r}{V_{oc}}$$

$$r = R_0 + \alpha \cdot (1 - s),$$

(7)

where V_{oc} as the open-circuit voltage, R_0 as the initial resistance of the battery, α as the DCIR-SoC coefficient, and T as the time length of the load – all can be considered to be constants. One can verify that E_{waste} is increasingly monotone to p and decreasingly monotone to s. Also, the function $E_{waste}(p, s)$ always has positive mixed partial derivatives. By Lemma 2 we know that, in general, we should try to power low-power loads by batteries in relatively low state-of-charge and power high-power loads by batteries in relatively high state-of-charge, so as to minimize the total wasted energy during a battery discharging cycle.

B Example Instance Showing the Naive Greedy Strategy is Suboptimal

The naive greedy strategy for the positional allocation problem is to allocate task with relatively larger (smaller) cost to a position with relatively smaller (larger) weight. Figure 3 shows a simple problem setting asking to assign a sequence of 80 tasks to 4 agents. There are only two task-types H and L in the task sequence, and $H < L$. The leftside shows the allocation scheme of the naive greedy strategy under this instance. The rightside shows the allocation scheme of the algorithm proposed in this paper, in which some L tasks are exchanged with some H tasks in lower positions, resulting a lower total cost due to the rearrangement inequality.

Fig. 2. Power consumption curve of smart phones. The average power under active mode is about 1800 mW, while the number under standby mode is less than 16 mW – a gap of more than 100x.

Fig. 3. Diagram illustration of the allocation scheme of the naive greedy algorithm and Algorithm 1 under the instance HHLHH LHLHH LLLHH HHLLH HHLLH HLLHH HHLLL LHHLH HHLHL LHHLL LHHHL HHHHH LHHLH HLLHL LHLHL HLLLH

14

C Proofs

C.1 Proof of Lemma 1

Proof. It is obvious that we can get unique allocation scheme from a decision sequence. We prove the opposite direction by giving an algorithm (Algorithm 2) that "decodes" how the tasks are assigned by the input allocation scheme.

Algorithm 2: Algorithm that decodes the decision sequence from the input allocation scheme

---

Input: $\mathbf{a}^1 \ldots \mathbf{a}^k$
Output: $\mathbf{x}$

1 set $p_r \leftarrow 1$ for $1 \leq r \leq k$;
2 $t \leftarrow 1$;
3 **while** $\exists r \in [k]$, $p_r \leq |\mathbf{a}^r|$ **do**
4   **if** $\exists r \in [k]$, $\mathbf{a}^r = t$ **then**
5          $x_t \leftarrow r$; $p_r \leftarrow p_r + 1$; $t \leftarrow t + 1$;
6   **else**
7   **return** "This is not a valid allocation scheme.";
8   **end**
9 **end**
10 **return x** ;

For contradiction, suppose Algorithm 2 fails to output a decision sequence (i.e. goes into Line 7) at time t when the input $\mathbf{A} = (\mathbf{a}^1, \ldots, \mathbf{a}^k)$ satisfies property (1)(2)(3) in Definition 1. Since every task before t has been decoded, we know $p_r > t$ for all $r \in [k]$. Due to the monotonicity of allocation schemes we know $t \notin \mathbf{a}^r$ for any $r \in [k]$, which violates the completeness of allocation schemes.

C.2 Proof of Lemma 3

Proof. The capacity of agent decreases 1 for each task assignment. Due to line $9 \sim 10$, Algorithm 1 never choose an agent i that has the same capacity with another agent j but has larger index of agent (i.e., $m_i = m_j$ and $i > j$), so the order of any agent pairs with respect to their capacities cannot be reversed.

C.3 Proof of Lemma 4

Proof. For a given look-ahead setup (r, h), we will show that $Z_H^{(r,h,t)} + Z_L^{(r,h,t)}$ decreases at most 1 for any single task assignment, which directly yields

$$(Z_H^{(r,h,i)} + Z_L^{(r,h,t)}) - (Z_H^{(r,h,j)} + Z_L^{(r,h,j)}) < j - i \text{ for any } i < j.$$

To prove $(Z_H^{(r,h,t)} + Z_L^{(r,h,t)}) - (Z_H^{(r,h,t+1)} + Z_L^{(r,h,t+1)}) \leq 1$ for any task t, let p^* be the current position of agent $r - 1$ at time t (i.e. $p^* = m + 1 - Q_{r-1}^{(t)}$), and let

15

r^* be any assignment decision of task t made by Algorithm 1 (i.e. $r^* = x_t$). It is obvious that $Z_H + Z_L$ will not change if $r^* > h$ or $r^* < r - 1$, since the task t is assigned to an agent not covered by either H-zone or L-zone (i.e. $r^* \notin S_H$ and $r^* \notin S_L$). In cases of $r \leq r^* leq h$, the size of L-zone Z_L doesn't change because the "threshold position" p^* doesn't change; and the size of H-zone Z_H decrease 1 due to the assignment of task t to agent $r^* \in S_H$.

In cases of $r^* = r - 1$, the "threshold position" p^* decreases 1, and the H-zone and L-zone will "get" and "lose", respectively, exactly one position for each agent $\gamma \in \{r \ldots h\}$. In addition, the L-zone will lose one position for agent $r - 1$ (i.e. the threshold position p^*). Due to Lemma 3, we have $m_{r-2} < m_{r-1}$ if Algorithm 1 chooses $r - 1$, which means none of agents $\gamma < r - 1$ has a position of p^* at this time, so the L-zone will not further lose positions for them despite of the decreasing of the threshold position p^*. In total, the sum of Z_H and Z_L decrease 1 in this case.

C.4 Proof of Lemma 6

Proof. For the decision sequence $\mathbf{x} = (x_1, \ldots, x_n)$, let $x_1 = \gamma$. By definition $\gamma > 1$. Let $A = (\mathbf{a}^1, \mathbf{a}^2, \ldots, \mathbf{a}^k)$ be the allocation scheme corresponding to $\mathbf{x}$.

Now consider $\mathbf{a}^1$ and $\mathbf{a}^\gamma$ in the allocation scheme. If $m_1 = m_\gamma$, we can simply construct $\mathbf{x}^*$ by switching the tasks in the two agents 1 and γ. Formally, this means we construct $A^* = (\mathbf{a}^\gamma, \mathbf{a}^2 \ldots \mathbf{a}^{\gamma-1}, \mathbf{a}^1, \mathbf{a}^{\gamma+1} \ldots \mathbf{a}^k)$, and by Lemma 1 we can in turn construct $\mathbf{x}^*$ from A^*. So, in the following we assume $m_1 < m_\gamma$. Because task 1 is assigned to agent γ, which is of capacity γ, we have

$$\mathbf{a}^1 = (a_1^1 \ldots a_{m_1-1}^1, a_{m_1}^1)^{\mathrm{T}}$$
$$\mathbf{a}^\gamma = (a_1^\gamma \ldots a_{m_1-1}^\gamma, a_{m_1}^\gamma, a_{m_1+1}^\gamma \ldots a_{m_\gamma-1}^\gamma, 1)^{\mathrm{T}}.$$

Since by definition $a_{m_1}^1 \neq a_{m_1}^\gamma$, there can be two cases:

Case 1: When $a_{m_1}^1 < a_{m_1}^\gamma$. Intuitively this means the agent 1 receives its first task $(a_{m_1}^1)$ before the capacity of agent γ goes down to below m_1. In that case, we exchange the allocation target for task 1 and $a_{m_1}^1$ (i.e. assign task 1 to agent 1 and assign task $a_{m_1}^1$ to agent γ). Due to Lemma 2, this is equivalent to a chain of exchanges, each either reduces the total cost or keeps it the same. Formally, assume $a_s^\gamma < a_{m_1}^1 < a_{s+1}^\gamma$ for some $m_1 < s \leq m_\gamma$ (such an s always exists), we construct

$$\mathbf{a}^{*1} = (a_1^1 \ldots a_{m_1-1}^1, 1)^{\mathrm{T}}$$
$$\mathbf{a}^{*\gamma} = (a_1^\gamma \ldots a_{m_1-1}^\gamma, a_{m_1}^\gamma, a_{m_1+1}^\gamma \ldots a_{s-1}^\gamma, a_{m_1}^1, a_s^\gamma \ldots a_{m_\gamma-1}^\gamma)^{\mathrm{T}},$$

and $\quad A^* = (\mathbf{a}^{*1}, \mathbf{a}^2 \ldots \mathbf{a}^{\gamma-1}, \mathbf{a}^{*\gamma}, \mathbf{a}^{\gamma+1} \ldots \mathbf{a}^k).$

16

Recall that $A = (\mathbf{a}^1, \mathbf{a}^2 \ldots \mathbf{a}^{\gamma-1}, \mathbf{a}^\gamma, \mathbf{a}^{\gamma+1} \ldots \mathbf{a}^k)$, then we have

$$COST_\mathbf{I}(\mathbf{x}^*) - COST_\mathbf{I}(\mathbf{x})$$

$$= \Big(g(m_1)L + g(s)T_{1,m_1} + \sum_{l=s+1}^{m_\gamma} g(l)T_{\gamma,l-1} \Big) - \Big(g(m_1)T_{1,m_1} + \sum_{l=s}^{m_\gamma-1} g(l)T_{\gamma,l} + g(m_\gamma)L \Big)$$

$$= L\Big(g(m_1) - g(m_\gamma) \Big) + T_{1,m_1}\Big(g(s) - g(m_1) \Big) + \sum_{l=s}^{m_\gamma-1} T_{\gamma,l}\Big(g(l+1) - g(l) \Big)$$

$$\leq L\Big(g(m_1) - g(m_\gamma) \Big) + L\Big(g(s) - g(m_1) \Big) + \sum_{l=s}^{m_\gamma-1} L\Big(g(l+1) - g(l) \Big)$$

$$= 0.$$

Case 2: When $a^\gamma_{m_1} < a^1_{m_1}$. This means the capacity of agent γ goes down to below m_1 before the agent 1 is ever assigned any task (thus still having a capacity of m_γ by then). In this case we simply exchange all tasks of agents 1 with the tasks assigned to agent γ when the capacity of γ is no more than m_1, yielding

$$\mathbf{a}'^1 = (a^\gamma_1 \ldots a^\gamma_{m_1})$$
$$\mathbf{a}'^\gamma = (a^1_1 \ldots a^1_{m_1}, a^\gamma_{m_1+1} \ldots a^\gamma_{m_\gamma-1}, 1).$$

Clearly the exchange will not change the total cost, thus reduce the problem to Case 1. □

C.5 Proof Sketch of Lemma 7

Proof Sketch. The complete proof is rather long, so in this paper we will omit some repeated details when it is safe to do that, especially for rigorous proofs of the superiority of a rearranged sequence like in Lemma 6.

It is easy to check that Lemma 7 holds if $m_r = 1$ for each agent $r \in [k]$ (any algorithm gives the same total cost in these cases). For general $(m_1 \ldots m_k) \in \mathbf{N}^k$, assume for induction that Lemma 7 holds for all the "smaller" instances, that is, that Algorithm 1 minimizes the total cost of any instance $(m_1 - \delta_1, m_2 - \delta_2, \ldots, m_k - \delta_k, T_{1+\sum \delta_r}, \ldots, T_n)$. We will prove that Algorithm 1 will also minimize the cost of the instance $(m_1, \ldots, m_k, T_1, \ldots, T_n)$. Specifically, for any decision sequence $\mathbf{x} = (x_1 \ldots x_n)$, let $x_1 = \gamma$, we only need to prove in two cases. For convenience we will denote H_i as the i-th H task in the sequence starting from task 1. Similarly we denote L_i as the i-th L task in the sequence.

When $\gamma < \gamma^*$: Without loss of generality we can assume that $x_2 \ldots x_n$ follows Algorithm 1, for otherwise we can simply turn to consider such a $\mathbf{x}'$, which guarantees to have lower costs than $\mathbf{x}$ due to the assumption of the induction. For any such instance $(m_1 \ldots m_k, H, T_2 \ldots T_n)$ and any such sequence $(\gamma, x_2 \ldots x_n)$, we can have the following observations, which collectively characterize a "overflowing" situation.

First, we know $m_\gamma < m_{\gamma^*}$ because Algorithm 1 always breaks ties by returning the agent with smaller id. Second, since Algorithm 1 returns γ^* for $T_1 = H$, by definition we know that there must exist h such that there are at least Z_L "L"s in $T_1 \ldots T_{Z_L + Z_H}$, where $Z_L = \sum_{i=1}^{h-1} \min\{m_i, m_{\gamma^*-1}\}$ and $Z_H = \sum_{i=\gamma^*}^{h} (m_i - m_{\gamma^*-1})$ are the sizes of H-zone and L-zone at this time, respectively. Third, we know $H_1 = 1$ is the only H task assigned by $\mathbf{x}$ to an agent with smaller id with γ^*, as shown by the following claim.

Claim. For any $i > 1$, we have $x_i \geq \gamma^*$ if $T_i = H$.

Proof. We know $x_{H_2} \geq \gamma^*$ because Algorithm 1 has chosen γ^* for H_1 and allocating H_1 to agent $\gamma < \gamma^*$ only decreases the stopping threshold of Algorithm 1. Furthermore, because we have assumed by induction that $x_2 \ldots x_n$ is optimal for $T_2 \ldots T_n$, by Lemma 5 we know that all the H tasks after H_2 will also be allocated to agents with id no smaller than γ^*.

In addition, the following claim shows that the sequence $\mathbf{x}$ (where $x_1 = \gamma$ and $x_2 \ldots x_n$ follows Algorithm 1) will allocate at least one "L" in the H-zone.

Claim. If allocating according to the sequence $x_1 \ldots x_n$, there always exists an i such that $1 \leq i \leq Z_L + Z_H$, $T_i = L$, $p_i > m_{\gamma^*-1}$, and $\gamma^* \leq x_i \leq h$.

Combining all the above observations together, we can derive the following claim, which asserts that there must be some L overflowing from the L-zone to the H-zone if we follow the sequence $\mathbf{x}$ when $\gamma < \gamma^*$. Let L^* denote the first such "L_i". Observe that the decision sequence $\mathbf{x}$ has put H_1 (which is also T_1) in the L-zone (i.e. "below" position m_r) while put L^* in the H-zone (i.e. above position m_r). Because L^* is the first "L" above m_r, we know all the tasks on top of L^* are "H".

Claim. If allocating according to the sequence $x_1 \ldots x_n$, there always exist $1 \leq H^* < L^* \leq n$ such that $p_{H^*} < p_{L^*}$, and that for any $H^* < i < L^*$, we have $T_i = L$ if $x_i = x_{H^*}$, and $T_i = H$ if $x_i = x_{L^*}$.

Based on this claim we can construct a better sequence $\mathbf{x}^*$ by allocating H_1 to agent x_{L^*} and allocating L^* to agent γ.

$$\mathbf{x}^* = (x_{L^*}, x_2 \ldots x_{L^*-1}, \gamma, x_{L^*+1} \ldots x_n)^{\mathrm{T}}$$

Compared with the one of $\mathbf{x}$, everything is the same except that a pair of H and L is exchanged in positions, which always lowers down the total cost due to Lemma 2). The rigorous proof of the benefit of this exchange is similar to the proof of Lemma 6.

When $\gamma > \gamma^*$: Informally these are the cases when $\mathbf{x}$ put the first task in somewhere "higher" (i.e., in a smaller position) than where it "should have been". In the following we will prove that, for any instance $(m_1 \ldots m_k, H, T_2 \ldots T_n)$ and any sequence $(\gamma, x_2 \ldots x_n)$ with $\gamma > \gamma^*$, there always exists a sequence $\mathbf{x}' = (x_1' \ldots x_n')$ with $x_1' = \gamma' < \gamma$ such that $COST_I(\mathbf{x}') < COST_I(\mathbf{x})$. In other words, any sequence $\mathbf{x}$ with $x_1 > \gamma^*$ cannot be the optimal sequence.

Again, by induction we can assume that $x_2 \ldots x_n$ follows Algorithm 1. Similar with the first case, we assume for contradiction that $x_1 \ldots x_n$ is optimal, which leads to a series of observations that collectively characterize a snapshot of the allocation. Then we will do some task exchanges in the allocation scheme to reduce the total cost without violating the monotonicity of the allocation scheme, thus forming a contradiction.

First, without loss of generality we know $m_{\gamma-1} < m_\gamma$, for otherwise if $m_{\gamma-1} = m_\gamma$ we will simply find the smallest agent id r with $m_r = m_\gamma$, exchange the tasks allocated in agent γ and r, and turn to consider the new allocation scheme.

Second, let H_2 denote the second H task in $T_1 \ldots T_n$ (i.e. $T_i = L$ for any $1 < i < H_2$), we know H_2 must not be put at the left side of γ if $\mathbf{x}$ wants to be optimal. That is, we have $x_{H_2} \geq x_{H_1} = \gamma$. This is because $x_{H_2} < x_{H_1}$ will lead to $p_{H_2} < p_{H_1}$ (note that we just showed $m_{\gamma-1} < m_\gamma$), in which case we can turn to consider the sequence $x_{H_2}, x_2 \ldots, x_{H_1}, \ldots x_n$, which guarantees to have no more cost than $x_1 \ldots x_n$. The complete proof is similar to the proof of Lemma 6. Note that by Lemma 5 all the subsequent H tasks will also be assign to agent γ or at its right side.

Third, we know that $\mathbf{x}$ allocates at least one H task below (including) the position $m_{\gamma-1}$. Formally, we have

Claim. There must exist an i such that $T_i = H$ and $p_i \leq n_{\gamma-1}$.

Let H^* be the *first* such H task (which basically "overflow"s from the H-zone). We know that there is no interleaved H and L in the H-zone at least until H^*. Formally, we have

Claim. For any $1 \leq i < j \leq H^*$, if $x_i = x_j$ and $T_j = H$, then we must have $T_i = H$.

Finally, let L^* be the first L task *after* H^* (such a L^* must exist, for otherwise H^* would be put somewhere lower). In the snapshot of the situation right after the L^* is allocated, we can construct a better decision sequence $\mathbf{x}'$ by allocating the first task in agent $\gamma-1$ (rather than in γ as the original sequence $\mathbf{x}$ does) and allocating L^* at the position of $L^* - 1$. It is easy to see that the re-arrangement virtually exchange a pair of H and L tasks, which will not increase the total cost due to Lemma 2. On the other hand, to show that such a re-arrangement will not violate the monotonicity of the allocation scheme, the key insight is to see that, by definition of Algorithm 1 there are no enough L tasks to "catch up with" the allocation pace of H as long as H is not overflowing from the H-zone. □

D   A Faster Version of Algorithm 1

Algorithm 3: A faster version of the simulation-based algorithm
---

Input: $m_1 \ldots m_k, T_1 \ldots T_n$, where $m_1 \leq m_2 \leq \cdots \leq m_k$

Output: $x_1 \ldots x_n$

1 let $\mathcal{H}_{r,h}$ be a $K \times K$ matrix denoting the look-ahead horizons

2 let $\mathcal{T}_{r,h}$ be a $K \times K$ matrix denoting the stopping thresholds

3 let $\mathcal{C}_{r,h}$ be a $K \times K$ matrix denoting the counters of L tasks

4 **for** $r = k$ **to** 2 **do**

5   **for** $h = \gamma$ **to** k **do**

6 $\mathcal{T}_{r,h} \leftarrow \sum_{i=1}^{h-1} \min\{m_i, m_{r-1}\}$

7 $\mathcal{H}_{r,h} \leftarrow \mathcal{T}_{r,h} + \sum_{i=r}^{h}(m_i - m_{r-1})$

8 $\mathcal{C}_{r,h} \leftarrow |\{i : i \in \{1, \ldots, \mathcal{H}_{r,h}\} \text{ and } T_i = L\}|$

9   **end**

10 **end**

11 set $Q_r^{(1)} = m_r$ for each $1 \leq r \leq k$

12 **for** $t = 1$ **to** n **do**

13 $x_t \leftarrow \texttt{StreamingAllocation} \ (Q_1^{(t)} \ldots Q_k^{(t)}, T_1 \ldots T_n, t)$

14 $Q_r^{(t+1)} \leftarrow Q_r^{(t)} - \mathbb{1}(x_t = r) \quad$ for each $1 \leq r \leq k$

15 **end**

16 **return x**

Algorithm 4: The routine in Algorithm 3, which updates the data variables incrementally.

---

1 Function StreamingAllocation

Input: $m_1 \ldots m_k, T_1 \ldots T_n, t$, where $m_1 \leq m_2 \leq \cdots \leq m_k$

Output: the agent to which T_t is assigned to

2 if $T_t = L$ **then**

3 $r^* \leftarrow$ the smallest r with $m_r > 0$

4 else

5   **for** $r = k$ **to** 2 **do**

6   **for** $h = \gamma$ **to** k **do**

7   **if** $\mathcal{C}_{r,h} \geq \mathcal{T}_{r,h}$ **then**

8 $r^* \leftarrow r$

9   **goto** line 14

10

11   **end**

12   **end**

13 $r^* \leftarrow 1$

14 end

15 for $r = k$ **to** 2 **do**

16   **for** $h = \gamma$ **to** k **do**

17   **if** $r^* > h$ **then**

18 $\mathcal{H}_{r,h} \leftarrow \mathcal{H}_{r,h} + 1$

19 $\mathcal{T}_{r,h} \leftarrow \mathcal{T}_{r,h}$

20 $\mathcal{C}_{r,h} \leftarrow \mathcal{C}_{r,h} - \mathbb{1}(T_t = L) + \mathbb{1}(T_{\mathcal{H}_{r,h}} = L)$

21   **else if** $r \leq r^* \leq h$ **then**

22 $\mathcal{H}_{r,h} \leftarrow \mathcal{H}_{r,h}$

23 $\mathcal{T}_{r,h} \leftarrow \mathcal{T}_{r,h}$

24 $\mathcal{C}_{r,h} \leftarrow \mathcal{C}_{r,h} - \mathbb{I}(T_t = L)$

25   **else if** $r^* = r - 1$ **then**

26 $\mathcal{H}_{r,h} \leftarrow \mathcal{H}_{r,h} + 1$

27 $\mathcal{T}_{r,h} \leftarrow \mathcal{T}_{r,h} - (1 + h - r)$

28 $\mathcal{C}_{r,h} \leftarrow \mathcal{C}_{r,h} - \mathbb{1}(T_t = L) + \mathbb{1}(T_{\mathcal{H}_{r,h}} = L)$

29   **else if** $r^* < r - 1$ **then**

30 $\mathcal{H}_{r,h} \leftarrow \mathcal{H}_{r,h}$

31 $\mathcal{T}_{r,h} \leftarrow \mathcal{T}_{r,h} - 1$

32 $\mathcal{C}_{r,h} \leftarrow \mathcal{C}_{r,h} - \mathbb{1}(T_t = L)$

33   **end**

34   **end**

35 end

36 return r^*