Online Scheduling of Equal-Length Jobs: Randomization and Restarts Help

Marek Chrobak* Wojciech Jawor* Jiří Sgall† Tomáš Tichý†

Abstract

We consider the following scheduling problem. The input is a set of jobs with equal processing times, where each job is specified by its release time and deadline. The goal is to determine a single-processor, non-preemptive schedule of these jobs that maximizes the number of completed jobs. In the online version, each job arrives at its release time. We give two online algorithms with competitive ratios below 2 and show several lower bounds on the competitive ratios. First, we give a $\frac{5}{3}$-competitive randomized algorithm. Our algorithm needs only one fair random bit, as it chooses one of two (nearly identical) deterministic algorithms, each with probability $\frac{1}{2}$. We also show a lower bound of $\frac{3}{2}$ for barely random algorithms, that (with arbitrary probability) choose one of two deterministic algorithms. Next, we give a deterministic $\frac{3}{2}$-competitive algorithm in the model that allows restarts, and we show that in this model the ratio $\frac{3}{2}$ is optimal. For randomized algorithms with restarts we show a lower bound of $\frac{6}{5}$.

1 Introduction

We consider the following fundamental problem in the area of real-time scheduling. The input is a collection of jobs with equal processing times p,

*Department of Computer Science, University of California, Riverside, CA 92521. Supported by NSF grants CCR-9988360 and CCR-0208856. {marek,wojtek}@cs.ucr.edu

†Mathematical Institute, AS CR, Žitná 25, CZ-11567 Praha 1, Czech Republic. Partially supported by Institute for Theoretical Computer Science, Prague (project LN00A056 of MŠMT ČR) and grant A1019401 of GA AV ČR. {sgall,tichy}@math.cas.cz

where each job j is specified by its release time r_j and deadline d_j. The desired output is a single-processor non-preemptive schedule. Naturally, each scheduled job must be executed between its release time and deadline, and different jobs cannot overlap. The term "non-preemptive" means that each job, must be executed without interruptions (in a contiguous interval of length p). The objective is to maximize the number of completed jobs.

In the *online version*, the jobs arrive over time. Each job j arrives at time r_j, and its deadline d_j is revealed at this time. The number of jobs and future release times are unknown. At each time step when no job is running, we have to decide whether to start a job, and if so, to choose which one, based only on the information about the jobs released so far. An online algorithm is called *c-competitive* if on every input instance it schedules at least $1/c$ as many jobs as the optimum.

It is known that a simple greedy algorithm is 2-competitive for this problem, and that this ratio is optimal for deterministic algorithms.

Our results. We present two ways to improve the competitive ratio of 2. First, addressing an open question in [6, 7], we give a $\frac{5}{3}$-competitive randomized algorithm, Interestingly, our algorithm needs only one fair random bit; it chooses with probability $\frac{1}{2}$ one of two deterministic algorithms. These two algorithms are, in fact, two independent copies of the same algorithm that use a lock mechanism to break the symmetry and coordinate their behaviors. We are not aware of previous work that uses such inter-process coordination mechanisms in the design of randomized online algorithms – and thus this technique may be of its own, independent interest. We then show a lower bound of $\frac{3}{2}$ on the competitive ratio of *barely random* algorithms that (with arbitrary probability) choose one of two deterministic algorithms.

Next, we give a deterministic $\frac{3}{2}$-competitive algorithm in the *preemption-restart* model. In this model, an online algorithm is allowed to abort a job during execution, in order to start another job. The algorithm gets credit only for jobs that are executed contiguously from beginning to end. Aborted jobs can be restarted (from scratch) and completed later. Note that the final schedule produced by such an algorithm is *not* preemptive. Thus the distinction between non-preemptive and preemption-restart models makes sense only in the online case. (The optimal solutions are always the same.) In addition to the algorithm, we give a matching lower bound, by showing that no deterministic online algorithm with restarts can be better than $\frac{3}{2}$-competitive. We also show a lower bound of $\frac{6}{5}$ for randomized algorithms with restarts.

We remark that both our algorithms are natural, easy to state and implement. The competitive analysis is, however, fairly involved, and it relies on some structural lemmas about schedules of equal-length jobs.

In a recent paper, Goldwasser [8], introduces the concept of algorithms that immediately, upon release of a job, commit if they will be completed or not. Although we do not formulate our algorithms explicitly in this form, they can be easily reformulated to satisfy this useful property.

In the paper we assume the model with integer release times and deadlines, which implicitly makes the time discrete. In the literature, some authors work with continuous time and assume that jobs have unit lengths. Both our algorithms can be easily modified to the continuous time model without any changes in performance, at the cost of somewhat more technical presentation of the analysis.

Previous work. The problem of scheduling equal-length jobs to maximize the number of completed jobs has been quite well studied in the literature. In the offline case, an $O(n \log n)$-time algorithm for the feasibility problem (checking whether *all* jobs can be completed) was given by Garey *et al.* [5] (see also [12, 3].) The maximization version can be also solved in polynomial time [4, 1], although the known algorithms[1] are slower than the one in [5].

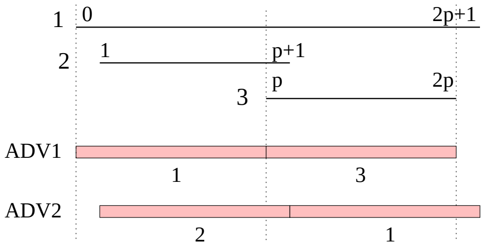

Figure 1: Lower bound proof

For the online version, Goldman *et al.* [6] gave a lower bound of $\frac{4}{3}$ on the competitive ratio for randomized algorithms and the tight bound of 2 for deterministic algorithms. To provide the reader with better intuition, we briefly outline these lower bound proofs. Let $p \geq 2$. The jobs used in the instance, written in the form $j = (r_j, d_j)$, are $1 = (0, 2p+1)$, $2 = (1, p+1)$, $3 = (p, 2p)$, see the figure. In the deterministic case, release job 1. If at time 0 the online algorithm starts job 1, then release job 2, otherwise release job 3. The online algorithm completes only one job. As the optimum is 2 in both cases, the competitive ratio is no better

[1] Carlier [3] claimed an $O(n^3 \log n)$-time algorithm for the maximization version but, as pointed out in [4], his algorithm is not correct.

than 2. For the randomized case, we use Yao's principle. We release job 1, and then randomly choose between jobs 2 and 3, each with probability $\frac{1}{2}$. The expected number of completed jobs of any deterministic algorithm is at most 1.5, as on one of the instances it completes only one job. Thus the competitive ratio is no better than $\frac{2}{1.5} = \frac{4}{3}$.

Goldman *et al.* [6] show that the lower bound of 2 can be beaten if the jobs on input have sufficiently large "slack". Specifically, they prove that a greedy algorithm is $\frac{3}{2}$-competitive for instances where $d_j - r_j \geq 2p$ for all jobs j. This is closely related to our algorithm with restarts: On such instances, our algorithm never uses restarts and becomes identical to the greedy algorithm. Thus in this special case our result constitutes an alternative proof of the result of [6]. Exploring this direction, Goldwasser [7] introduced a notion of patience and obtained a parameterized version of this result: if $d_j - r_j \geq \lambda p$ for all jobs j, where $\lambda \geq 1$ is an integer, then the competitive ratio decreases to $1 + 1/\lambda$.

In our brief overview of the literature given above we focused on the case when jobs are of equal length and the objective function is the number of completed jobs. We need to stress though that, in addition to the work cited above, there is vast literature on real-time scheduling problems, where a variety of other models is considered. Other or no restrictions can be placed on processing times, jobs may have different weights (benefits), we can have multiple processors, and preemption may be allowed. For example, once arbitrarily processing times and/or weights are introduced, no constant-competitive algorithms exist. Therefore it is common in the literature to allow preemption with resume, where a job can be preempted and later started from where it was stopped.

The model with restarts was studied before by Hoogeveen *et al.* [9]. They present a 2-competitive deterministic algorithm with restarts for jobs with arbitrary processing times and objective to maximize the number of completed jobs (i.e., no weights). They also give a matching lower bound. However, their algorithm does not use restarts on the instances with equal processing times, and thus it is no better than 2-competitive for our problem.

Real-time scheduling is an area where randomized algorithm have been found quite effective. Most randomized algorithms in the general scenarios use the "classify-and-randomly-select" technique introduced by Lipton and Tomkins [10]. In general, this method decreases the dependence of competitive ratio from linear to logarithmic in certain parameters (e.g., the ratios between the maximum and minimum weights.) This technique seems to be of no use for our case of equal length (and equal weight) jobs, and our randomized algorithm is based on entirely different ideas.

Barely random algorithms are an interesting concept recently introduced in the area of online algorithms. The first work on barely random scheduling algorithms we are aware of is that of Seiden [11], who obtained significant results in the area of makespan scheduling. Bartal *et al.* [2] show that barely random online algorithms are also useful for the k-server problem.

The area of real-time scheduling is of course well motivated by multitudes of applied scenarios. In particular, the model of equal-length jobs – without or with limited preemption – is related to applications in packet switched networks. When different weights are considered, the problem has further connections to the "quality of service" issues (recently a fashionable phrase.) Nevertheless, we shamelessly admit that this work has been partially driven by plain curiosity. It is quite intriguing, after all, that so little is known about the competitiveness of such a fundamental scheduling problem.

2 Preliminaries

Notation and terminology. The instance on input is a set of jobs $J = \{1, 2, \ldots\}$. Each job j is given by its release time r_j and deadline d_j. All jobs have processing time p. (We assume that all numbers are positive integers and that $d_j \geq r_j + p$ for all j.) The *expiration time* of job j is $x_j = d_j - p$, i.e., the last time when it can be started. A job j is called *available* at time t if $r_j \leq t \leq x_j$. A job j is called *tight* if $x_j - r_j < p$.

A *non-preemptive schedule* A assigns to each completed job j an interval $[S_j^A, C_j^A)$, with $r_j \leq S_j^A \leq x_j$ and $C_j^A = S_j^A + p$, during which it is executed. These intervals are disjoint for distinct jobs. S_j^A and C_j^A are called the *start time* and *completion time* of job j. Both are assumed to be integer, w.l.o.g. We adopt a convention that "job running (a schedule being idle, etc.) at time t" is an equivalent shortcut for "job running (a schedule being idle, etc.) in the interval $[t, t+1)$". Given a schedule A, a job is *pending* at time t in A if it was not completed yet and $r_j \leq t \leq x_j$. The *benefit* of the schedule, denoted $|A|$, is the number of completed jobs.

A set of jobs P is called *feasible* at time t if there exists a schedule which completes all jobs in P such that no job is started before t. A set of jobs P is *flexible* at time t if it is feasible at time $t + p$. We say that a job started by a schedule at time t is *flexible* if the set of all jobs pending at t is flexible; otherwise the job is called *urgent*.

An *online algorithm* constructs a schedule incrementally, at each step t making decisions based only on the jobs released at or before t. Each job j is revealed (including its deadline) to the algorithm at its release time r_j. A *non-preemptive online algorithm* can start a job only when no job is being running; thus, if a job is started at time t the algorithm has no choice but to let it complete by the time $t + p$. An *online algorithm with restarts* can start a job at any time. If we start a job j when another job, say k, is running, then k is aborted and started from scratch when (and if) it is started again later. The unfinished portion of k is removed from the overall schedule (so the schedule is considered to be idle during this time interval.) Thus the schedule generated by an online algorithm with restarts is also a non-preemptive schedule.

An online algorithm is called *c-competitive* if, for any instance (set of jobs) J and any schedule ADV for J, the schedule A generated by the algorithm on J satisfies $|\text{ADV}| \leq c|A|$. If the algorithm is randomized, the expression $|A|$ is replaced by the expected (average) number of jobs completed on the given instance.

Some properties of schedules. For every instance J, we fix a canonical linear ordering $\prec$ of J such that $j \prec j'$ implies $d_j \leq d_{j'}$. In other words, we order the jobs by their deadlines, breaking the ties arbitrarily but consistently for all applications of the deadline ordering. The term *earliest-deadline*, or briefly ED, now refers to the $\prec$-minimal job.

A schedule A is called EDF (earliest-deadline-first) if, whenever it starts a job, it chooses the earliest-deadline job of all the pending jobs that are later completed in A.

A schedule is *normal* if (i) whenever it starts a job, it chooses the ED job from the set of all pending jobs, and (ii) whenever the set of all pending jobs is not flexible, it starts a job. Both conditions (i) and (ii) are reasonable, in the sense that any algorithm can be modified, using a standard exchange argument, to satisfy them without reducing the number of scheduled jobs. (We omit the proof, as we do not need this fact in the paper). Furthermore, the rules can be guaranteed by an online algorithm; indeed, all our algorithms generate normal schedules. Obviously, any normal schedule is EDF, but the reverse is not true. The following property is useful.

Lemma 2.1 *Suppose that a job j is urgent in a normal schedule A. Then at any time t, $S_j^A \leq t \leq x_j$, an urgent job is running in A.*

Proof: Let P be the set of jobs pending at time S_j^A. Towards contradiction, suppose that there is a time t, $S_j^A \leq t \leq x_j$, when A is idle or starts a flexible job at t. Then the set Q of jobs pending at time t is flexible at t and, since j is the ED job from P (by normality of A), Q contains all the jobs from P that are not completed in A before t. Since Q is flexible at t, we can schedule all jobs of Q from $t+p$, start j at t and schedule all jobs in $P - Q - \{j\}$ as in A. This shows that P is flexible at time S_j^A – a contradiction. □

Two schedules D and D' for an instance J are called *equivalent* if D starts a job at t if and only if D' starts a job at t; furthermore the job started in D is flexible if and only if the job started in D' is flexible. Obviously, $|D| = |D'|$ for equivalent schedules.

For the purpose of our analysis, we modify normal schedules into equivalent EDF schedules with better structural properties. In particular, the next lemma gives us more control over the choice of jobs that are scheduled, if there are more equivalent choices. The idea of the construction is straightforward: keep a list of jobs we want to schedule to satisfy the requirements; verifying all the details is tedious.

Lemma 2.2 *Let A be a normal schedule for a set of jobs J. Let $f(j) : J \to J$ be a partial function such that if $f(j)$ is defined then j is scheduled as flexible in A and $r_{f(j)} < C_j^A \leq x_{f(j)}$. Then there exists an EDF schedule A' equivalent to A such that:*

(1) *All jobs $f(j)$ are completed in A'.*

(2) *If j is available at time t when A' is idle or starts a flexible job then j is completed in A'.*

(3) *Let j be a job completed in A, let $t = S_j^A$, and let R be the set of all jobs j' with $r_{j'} < t+p$ that are pending at $t+p$. If R is feasible at $t+p$ then all the jobs in R are completed in A'.*

Proof: We iteratively produce the schedule A', while maintaining a feasible set P_t of pending jobs at each time t when A is idle or starts a job. In parallel with the construction, we prove inductively that P_t is an $\subseteq$-maximal feasible subset of the jobs pending at time t, for **each** schedule A or A'. Note that if P_t is flexible, then its maximality implies that P_t is equal to the set of **all** jobs pending at t, both for A and A'. We now describe the construction.

Initially, choose P_0 as an arbitrary maximal feasible set of jobs released at time 0.

Suppose we have already defined P_t. If A is idle at t, we let A' idle and choose an arbitrary $P_{t+1} \supseteq P_t$ by adding the newly arrived jobs as long as the set is feasible. Since A is idle, P_t is flexible at t and thus (i) P_t contains all the pending jobs at t, in each schedule A or A', and (ii) P_t is feasible at $t+1$. This implies that the constructed set P_{t+1} is a maximal feasible set at $t+1$, both for A and A'.

If A starts a job j at time t, A' starts the earliest-deadline (more precisely, $\prec$-minimal) job k from P_t. Since P_t is a maximal feasible set both for A and A', it is non-empty whenever A starts a job. Furthermore, we have the following equivalences: the job started in A is flexible at t iff P_t is flexible at t iff k is flexible at t in A'.

Let $Q = P_t - \{k\}$. We claim that the following properties hold for each schedule A and A':

(Q1) Each job $i \in Q$ is pending at time $t+p$,

(Q2) Q is feasible at time $t+p$, and

(Q3) Q is $\subseteq$-maximal among all sets of jobs that satisfy conditions (Q1) and (Q2).

By the definition of Q, (Q2) holds (for both A and A') and (Q1) holds for A'. No job $k' \notin P_t$ can be feasibly added to Q at time $t+p$, as otherwise it could be feasibly added to P_t at time t. Thus, for A', Q satisfies (Q3), i.e., is maximal. If P_t is flexible, then P_t contains all jobs pending at t both for A and A', thus $j = k$ and Q satisfies (Q1) and (Q3) for A. If k is urgent, then $j \preceq k$ since k is available to A and A schedules the ED pending job (as A is normal); thus, for A, all jobs in Q are pending at $t+p$ and (Q1) holds. Furthermore, P_t is not feasible at $t+p$ (as otherwise P_t would be flexible contradicting the assumption that k is urgent). Thus (Q3) holds also for A, as adding both k and a job not in P_t leads to a non-feasible set.

Now we construct P_{t+p} from Q by adding the newly arrived jobs in a particular order. First, if defined, we add $f(j)$: in this case P_t is flexible at t and thus Q is flexible at $t+p$, so $f(j)$ can always be added to Q if it is pending and not yet in Q (which is the case if $r_{f(j)} \leq t$). Then we add all the jobs j' with $t < r_{j'} < t+p$, in an arbitrary order, adding each job only if the set is feasible. Finally, we add all the jobs j' with $r_{j'} = t+p$, in an arbitrary order, adding each job only if the set is feasible. The maximality of Q and the construction implies that P_{t+p} is a maximal feasible set of jobs pending at $t+p$, both for A and A'.

This completes the construction. Obviously, A and A' are equivalent. Also, A' is EDF since the ED job of P_t is scheduled and no jobs $j' \notin P_t$ pending at t are added to $P_{t'}$, $t' > t$.

By the construction, A' schedules all the jobs that are in some plan P_t. At any time t when A' is idle or starts a flexible job, plan P_t is flexible and thus contains all pending jobs. This proves (2). Since $f(j) \in P_t$ for $t = C_j^A$, it also implies (1). Finally, to show (3), recall that when constructing P_{t+p} for $t = S_j^A$, we are first adding to P_{t+p} all the jobs j' with $r_{j'} < t+p$ (by the assumption of the lemma, $j' = f(j)$ satisfies this); if they are all together feasible then they are all added and thus $R \subseteq P_{t+p}$. □

Lemma 2.2 gives an easy proof that any normal schedule A schedules at least half of the jobs of any schedule ADV for the same instance. Take the modified schedule A' from Lemma 2.2. Charge (i.e. map) any job j completed in ADV to a job completed in A' as follows: (i) If A' is running a job k at time S_j^{ADV}, charge j to k. (ii) Otherwise charge j to j. This is well defined, since if j is available at S_j^{ADV} and if A' is idle, A' completes j by Lemma 2.2(2). Furthermore, only one job can be charged to k using (i), as all jobs have the same processing time and only one job can be started in ADV during the interval when k is running in A'. Thus overall at most two jobs are charged to each job in A' and $|\mathrm{ADV}| \leq 2|A'| = 2|A|$, as claimed.

This shows that any online algorithm that generates a normal schedule is 2-competitive. In particular, this includes the known result that the greedy algorithm which always schedules the ED pending job when there are any pending jobs is 2-competitive. We use similar but more refined charging schemes to analyze our improved algorithms.

3 Randomized Algorithms

3.1 A $\frac{5}{3}$-Competitive Algorithm

In this section we present our $\frac{5}{3}$-competitive barely random algorithm. This algorithm needs only one random bit; at the beginning of computation it chooses with probability $\frac{1}{2}$ between two schedules. We also later show a lower bound for barely random algorithms: any randomized algorithm that randomly chooses between two schedules has ratio at least $\frac{3}{2}$.

Algorithm RANDLOCK**.** The algorithm runs two identical processes, A and B, sharing a common lock. Each process computes one schedule of its own copy of the given instance J. (This means that with the exception of the lock, the processes are independent; e.g., a given job can be executed by both processes at the same or different times.) The algorithm chooses with probability $\frac{1}{2}$ each process.

Each process works as follows:

(1) If there is no pending job, wait for the next arrival.

(2) If the pending jobs are not flexible, execute the earliest-deadline pending job.

(3) If the pending jobs are flexible and the lock is available, acquire the lock (ties broken arbitrarily), execute the earliest-deadline pending job, and release the lock upon its completion.

(4) Otherwise wait until the lock becomes available or the pending jobs become not flexible (due to progress of time and/or job arrivals).

Before we analyze the algorithm, we illustrate the behavior of the algorithm on the instance in Figure 2. Both processes schedule only 3 jobs out of optimal 5, thus the algorithm is not better than $\frac{5}{3}$-competitive.

Let A and B denote the schedule generated by the corresponding processes on a given instance J. It is easy to see that RANDLOCK is a non-preemptive online algorithm and both schedules are normal. Fix an arbitrary schedule ADV for the given instance J.

We start our proof by modifying the schedules A and B according to Lemma 2.2. We define partial functions f^D, $D \in \{A, B\}$. Define $f^D(j) = k$ if j is a flexible job completed in D and k is a job started in ADV during the execution of j in D and available at the completion of j in D, i.e., such that $S_j^D \le S_k^{\mathrm{ADV}} < C_j^D \le x_k$. Otherwise (if j is not flexible or no such k exists), $f^D(j)$ is undefined. Note that if k exists, it is unique for a given j.

Let D' be the schedule constructed in Lemma 2.2 using $f = f^D$. We stress that D' is not actually constructed by the algorithm (in fact, it cannot be constructed online as its definition depends on ADV); it is only a tool for the analysis of RANDLOCK. Since D' is equivalent to a normal schedule D, Lemma 2.1 still applies and the number of completed jobs remains the same as well.

To avoid clutter, we slightly abuse the notation and from now on we use A and B to denote the modified schedules A' and B'. Whenever D denotes one of the processes or schedules A and B, then $\bar{D}$ denotes the other one.

Observation: An important property guaranteed by the lock mechanism is that if D is idle at time t and the lock is available (i.e., $\bar{D}$ is idle or executing an urgent job), then all jobs j available at t are completed by time t in D, as otherwise D would schedule some job. Furthermore, any such j is executed as flexible: otherwise by Lemma 2.1 D cannot be idle at time t, $S_j^D \le t \le x_j$.

The charging scheme. Let j be a job started at time $t = S_j^{\mathrm{ADV}}$. This job generates several charges of different weights to (the occurrences of) the jobs in schedules A and B. There are two types of charges: *self-charges* from job j to the occurrences of j in A or B, and *up-charges*, from j to the jobs running at time t in A and B. (See Figure 3.) The total of charges generated by j is always 1.

Case (I): Both schedules A and B are idle. By the observation above, j is completed in both A and B. We generate two self-charges of $\frac{1}{2}$ to the two occurrences of j in A and B.

Case (II): One schedule $D \in \{A, B\}$ is running an urgent job k and the other schedule $\bar{D}$ is idle. By the observation, j is completed by time t in $\bar{D}$. We generate one self-charge of $\frac{1}{2}$ to the occurrence of j in $\bar{D}$ and one up-charge of $\frac{1}{2}$ to k in D.

Case (III): One schedule $D \in \{A, B\}$ is running a flexible job k and the other schedule $\bar{D}$ is idle. We claim that j is completed in both A and B. This follows from Lemma 2.2(2) for $\bar{D}$ and also for D, if $r_j \le S_k^D$. If $x_j \ge C_k^D$ then $f^D(k) = j$ and D completes k by Lemma 2.2(1). In the remaining case, we have $S_k^D < r_j \le t \le x_j \le C_k^D$; thus j is a tight job and $\bar{D}$ cannot be idle at t, contradicting the case condition.

In this case we generate one up-charge of $\frac{1}{3}$ to k in D and two self-charges of $\frac{1}{2}$ and $\frac{1}{6}$ to the occurrences of j according to the subcases as follows. Let $E \in \{A, B\}$ be the schedule which starts j first (breaking ties arbitrarily).

Case (IIIa): If E schedules j as an urgent job and the other schedule $\bar{E}$ is idle at some time t' satisfying $S_j^E \le t' \le x_j$, then charge $\frac{1}{6}$ to the occurrence of j in E and $\frac{1}{2}$ to the occurrence of j in $\bar{E}$. Note that by Lemma 2.1, E is running an urgent job at $t' \le x_j$, and by the observation above, j is flexible and completed in $\bar{E}$ before t'.

Case (IIIb): Otherwise charge $\frac{1}{2}$ to the occurrence of j in E and $\frac{1}{6}$ to the occurrence of j in $\bar{E}$.

Case (IV): Both processes A and B are running jobs k_A and k_B, respectively, at time t. We show in Lemma 3.1 that one of k_A and k_B receives a self-charge of at most $\frac{1}{6}$ from its occurrence in ADV. This job receives an up-charge of $\frac{2}{3}$ from j and the other one of k_A and k_B an up-charge $\frac{1}{3}$ from j. No self-charge is generated.

Lemma 3.1 *In case* (IV)*, one of k_A and k_B receives a self-charge of at most $\frac{1}{6}$.*

Proof: Assume, towards contradiction, that both k_A and k_B receive a self-charge of $\frac{1}{2}$. At least one of k_A and k_B is scheduled as urgent in the corresponding schedule, due to the lock mechanism. Thus $k_A \neq k_B$, as (I) is the only case when two self-charges $\frac{1}{2}$ to the same job are generated and then both occurrences are flexible. Furthermore, if $j = k_D$, $D \in \{A, B\}$, then k_D has no self-charge. Thus k_A, k_B and j are three distinct jobs.

Claim: If k_D, $D \in \{A, B\}$, receives a self-charge of $\frac{1}{2}$ in case (IIIb) (applied to k_D) and $S^{\mathrm{ADV}}_{k_D} \leq t - p$ (i.e., k_D is scheduled before j in ADV), then $k_{\bar{D}} \prec k_D$.

Proof of Claim: If (IIIb) applies, generating a self-charge of $\frac{1}{2}$ to k_D then $\bar{D}$ schedules k_D after $k_{\bar{D}}$. Furthermore, $S^{\bar{D}}_{k_{\bar{D}}} \geq t - p \geq S^{\mathrm{ADV}}_{k_D} \geq r_{k_D}$ and thus k_D is pending in $\bar{D}$ when $k_{\bar{D}}$ is started. Since $\bar{D}$ is EDF, we have $k_{\bar{D}} \prec k_D$, as claimed.

Choose D such that k_D is urgent in D (as noted above, such D exists). The only case when an urgent job receives a self-charge of $\frac{1}{2}$ is (IIIb). Since by Lemma 2.1, D executes urgent jobs at all times t', $t \leq t' \leq x_{k_D}$, and $j \neq k_D$, it follows that $S^{\mathrm{ADV}}_{k_D} \leq t - p$. By the claim, $k_{\bar{D}} \prec k_D$ and $x_{k_{\bar{D}}} \leq x_{k_D}$. Furthermore, since (IIIa) does not apply, $\bar{D}$ is also not idle at any time t', $t \leq t' \leq x_{k_D}$.

If $k_{\bar{D}}$ is self-charged $\frac{1}{2}$ in cases (I), (II), (IIIa) or the subcase of (IIIb) when $S^{\mathrm{ADV}}_{k_{\bar{D}}} > t$, then at least one process is idle at some time t', $t < t' \leq x_{k_{\bar{D}}} \leq x_{k_D}$, which is a contradiction with previous paragraph. If $k_{\bar{D}}$ is self-charged $\frac{1}{2}$ in the subcase of (IIIb) when $S^{\mathrm{ADV}}_{k_{\bar{D}}} \leq t$, then $S^{\mathrm{ADV}}_{k_{\bar{D}}} \leq t - p$ as $j \neq k_{\bar{D}}$, and the claim above applies to $k_{\bar{D}}$; however the conclusion that $k_D \prec k_{\bar{D}}$ contradicts the linearity of $\prec$ as $k_D \neq k_{\bar{D}}$ and we have already shown that $k_{\bar{D}} \prec k_D$. We get a contradiction in all the cases, completing the proof of the lemma. □

Finally, we show that the total charge to each occurrence of a job in A or B is at most $\frac{5}{6}$. During the time when a job is running in A or B, at most one job is started in ADV, thus each job gets at most one up-charge in addition to a possible self-charge (in certain degenerate cases these two may come from the same job in ADV). The weight of a self-charge j is always either $\frac{1}{2}$ or $\frac{1}{6}$. In particular, if a job does not receive any up-charge, it is charged less than $\frac{5}{6}$. If a job k in D receives an up-charge in (II), it is a tight job and, since the $\bar{D}$ is idle, it is already completed in $\bar{D}$; thus the self-charge is at most $\frac{1}{6}$ and the total is at most $\frac{1}{6} + \frac{1}{2} < \frac{5}{6}$. If a job k in D receives an up-charge in (III), the up-charge is only $\frac{1}{3}$ and thus the total is at most $\frac{1}{3} + \frac{1}{2} = \frac{5}{6}$. If a job k in D receives an up-charge in (IV), Lemma 3.1 implies that the up-charges can be defined as claimed in the case description. The total charge is then bounded by $\frac{1}{6} + \frac{2}{3} = \frac{5}{6}$ and $\frac{1}{2} + \frac{1}{3} = \frac{5}{6}$, respectively.

The competitive ratio of $\frac{5}{3}$ now follows by summing the charges over all jobs and observing that the expected number of jobs completed by RandLock is $\frac{1}{2}(|A| + |B|)$.

Theorem 3.2 RandLock *is a $\frac{5}{3}$-competitive non-preemptive randomized algorithm for scheduling equal-length jobs.*

3.2 A Lower Bound for Barely Random Algorithms

Theorem 3.3 *Suppose that $\mathcal{A}$ is a barely-random non-preemptive algorithm for scheduling equal-length jobs that chooses one of two deterministic algorithms. Then $\mathcal{A}$ is not better than $\frac{3}{2}$-competitive.*

Proof: Assume that we have two deterministic algorithms, A and B, of which one is chosen as the output schedule randomly, with arbitrary probability. Let $p \geq 3$ and write the jobs in the format $j = (r_j, d_j)$. We start with job $1 = (0, 4p)$. Let t be the first time when one of the algorithms, say A, schedules job 1. If B schedules it at t as well, release a job $1' = (t+1, t+p+1)$; obviously the optimum schedules both jobs while both A and B only one, so the competitive ratio is at least 2.

So we may assume that B is idle at t. Release job $2 = (t+1, t+2p+2)$. If B starts job 2 at $t+1$, release job $3 = (t+2, t+p+2)$, otherwise release $4 = (t+p+1, t+2p+1)$. By the choice of the last job, B can complete only one of the jobs 2, 3, 4. Since A is busy with job 1 until time $t+p$, it also can complete only one of the jobs 2, 3, 4, as their deadlines are strictly smaller than $t+3p$. So both A and B complete two jobs. The optimum can complete all three released jobs. If 3 is issued, schedule $3, 2$, back to back, starting at time $t+2$. If 4 is issued, schedule $2, 4$, back to back, starting at time $t+1$. In either case, both jobs fit in the interval $[t+1, t+2+2p)$. If $t \geq p-1$, then job 1 can be scheduled at time 0, otherwise, it can be scheduled at time $3p \geq t+2+2p$. Thus the competitive ratio is at least $\frac{3}{2}$. $\Box$

4 Scheduling with Restarts

4.1 The $\frac{3}{2}$-Competitive Algorithm

Our algorithm with restarts is very natural. At any time, it greedily schedules the ED job. However, if a new tight job arrives and it would expire before the running job is completed, we consider a preemption. If all pending jobs can be scheduled, the preemption occurs. If not, it means that some pending job is necessarily lost and the preemption would be useless—so we continue running the current job and let the tight job expire.

We need an auxiliary definition. Suppose that a job j is started at time s by the algorithm. We call a job k a *preemption candidate* if $s < r_k \leq x_k < s + p$.

Algorithm TIGHTRESTART**.** At time t:

(1) If no job is running, start the ED pending job, if there are any pending jobs, otherwise stay idle.
(2) Otherwise, let j be the running job. If no preemption candidate is released at t, continue running j.
(3) Otherwise, choose a preemption candidate k released at t (use the ED job to break ties.) Let P be the set of all jobs pending at time t, excluding any preemption candidates (but including j). If P is feasible at $t+p$, preempt j and start k at time t. Otherwise continue running j.

Preliminary considerations. Let A be the non-preemptive schedule generated by TIGHTRESTART (after removing the preempted parts of jobs.) We stress that we distinguish between A being idle and TIGHTRESTART being idle: at some time steps TIGHTRESTART can process a job that will be preempted later, in which case A is considered idle but TIGHTRESTART is not.

Obviously, TIGHTRESTART is an online algorithm with restarts, and any job it starts is the ED pending job. To prove that A is a normal schedule, we need a few more observations:

(1) A job j that was started as urgent is never preempted: Let R be the set of pending jobs at time t' when j is started, and suppose that at time t a preemption candidate arrives. If $x_j < t$ then j itself is not feasible at t. Otherwise all jobs in R are pending at t (as j is the ED job in P) and thus $P \supseteq R$ cannot be feasible at $t + p$ since already R is not feasible at $t' + p < t + p$.

(2) If j is preempted, then this happens on the first release of a preemption candidate: the condition in step (1) only gets stronger with further jobs released, using also the fact that j is flexible by (1) and thus no job available at its start expires.

(3) If A is idle at t' but a job j is running at t' and preempted at time $t > t'$, the set R of all jobs pending at time t' is flexible: Since j is flexible and R does not contain any preemption candidates by (2), we have $R \subseteq P$ where P is the set in step (3) of the algorithm at time t. If j is preempted at time t, P is flexible at t, thus R is flexible at $t' < t$.

Summarizing, A always starts the ED pending job; if a preemption occurs, we use (2) and the choice of the scheduled preemption candidate to see that it is ED. (1) implies that if an urgent job is started, it is also completed, and (3) implies that if A is idle then the set of pending jobs is flexible. Thus A is a normal schedule and we can proceed towards application of Lemma 2.2.

Define a partial function $f : J \to J$ as follows. Let j be a job scheduled as flexible in A.

- If at some time t, $S_j^A \leq t < C_j^A$, ADV starts a job k which is not a preemption candidate then let $f(j) = k$.
- Otherwise, if there exists a job k with $S_j^A < r_k < C_j^A \leq x_k$ such that ADV does not complete k, then let $f(j) = k$ (choose arbitrarily if there are more such k's).
- Otherwise, $f(j)$ is undefined.

Let A' be the schedule constructed in Lemma 2.2 from A and the function f. As before, we abuse A to denote the modified schedule A' as well.

Call a job j scheduled in ADV a *free* job if TightRestart is idle at time S_j^{ADV}. This implies that at time S_j^{ADV} no job is pending in A; in particular, j is completed by time S_j^{ADV} in A. (These jobs need special attention, as TightRestart was "tricked" into scheduling them too early.)

If a job j in ADV is started while a job k is running in A, we want to charge j to k. However, due to preemptions, the jobs can become misaligned, so we replace this simple matching by a slightly more technical definition. We match the jobs from the beginning of the schedule, a job k in A is matched to the next job in ADV, provided that it starts later than k; an exception is that if k is free then we prefer to match it to itself rather than to another job that starts much later.

Formally, define a partial function $M : J \to J$ which is a matching of (some) occurrences of jobs in A to those in ADV. Process the jobs k scheduled in A in the order of increasing S_k^A. Let j be the first unmatched job started in ADV after S_k^A, i.e., a job with smallest S_j^{ADV} among those with $S_j^{\mathrm{ADV}} \geq S_k^A$ and not in the current range of M (i.e., for no k' with $S_{k'}^A < S_k^A$, $j = M(k')$). If no such j exists, $M(k)$ is undefined. If k is a free job, not in the current range of M, and $S_j^{\mathrm{ADV}} \geq C_k^A$, then let $M(k) = k$. Otherwise let $M(k) = j$.

The definition implies that M is one-to-one. Furthermore, for any j scheduled in ADV, if A is executing a job k at S_j^{ADV}, then $j = M(k')$ for some k': if j is not in the range of M before k is processed then $M(k)$ is defined as j.

Lemma 4.1 *If j is free and $f(j)$ is undefined then j is in the range of M.*

Proof: Since j is free, it is completed in A before it is started in ADV. Let k be the job started in ADV at some time t, $S_j^A \leq t < C_j^A$. If no such k exists or $M(j) \neq k$ then j is in the range of M and the lemma holds: if j is not in the range of M before j is processed, then $M(j)$ is defined to be j.

Since $f(j)$ is undefined, k is a preemption candidate. Thus it remains to handle the case when k is a preemption candidate, yet TIGHTRESTART does not preempt, and $M(j) = k$.

The idea is this: We show that after j, A schedules many jobs that, in the definition of M, could be matched to jobs in ADV scheduled after k, so they cannot be all matched before we try to match to j in ADV. This gets a bit technical, e.g., we need to verify that these jobs are not free.

Let $K = \{j' \mid S_j^A < r_{j'} < C_j^A \leq x_{j'}\}$ be the set of all jobs released during the execution of j in A, excluding preemption candidates. Since $f(j)$ is undefined, all these jobs are completed in ADV, and obviously they cannot be completed before S_k^{ADV}. Thus K is feasible at C_k^{ADV} and also at $C_j^A \leq C_k^{\mathrm{ADV}}$.

Let P be the set of all jobs pending at r_k in A excluding all preemption candidates (but including j). Let $R = P \cup K - \{j\}$ and $u = |R|$. Since A is an EDF schedule and all jobs $j' \in J - K$ are available at S_j^A, they all have $d_{j'} \geq d_j \geq C_j^A$. Thus R is exactly the set of all jobs pending at C_j^A in A with $r_{j'} < C_j^A$.

Suppose, towards contradiction, that R not feasible at C_j^A. Since K is feasible and all jobs in $j' \in J - K$ have $d_{j'} \geq d_j$, TIGHTRESTART is then executing urgent jobs from C_j^A until at least the time x_j. Thus A is not idle at time $S_j^{\mathrm{ADV}} \leq x_j$ and j is not free. Thus R is feasible at C_j^A and by Lemma 2.2(3), A completes all jobs in R. Furthermore, all jobs in R

are scheduled between C_j^A and S_j^{ADV}, as TightRestart is idle at S_j^{ADV}. Consequently, $S_j^{\mathrm{ADV}} - C_j^A \geq up$.

Next we claim that (i) $S_j^{\mathrm{ADV}} - C_k^{\mathrm{ADV}} < up$ and (ii) ADV does not schedule any of the jobs in R after j. If either of the items is violated, $R \cup \{j\}$ is feasible at C_k^{ADV}: First we schedule K, which is feasible at C_k^{ADV}, and then the remaining jobs from R. If (i) is violated, we can complete all jobs in R by the time S_j^{ADV}, which is smaller than the deadlines in $P - K$, and start j at S_j^{ADV}. If (ii) is violated, let j' be the job in R scheduled after j in ADV. We know that $S_j^{\mathrm{ADV}} - C_k^{\mathrm{ADV}} > S_j^{\mathrm{ADV}} - C_j^A - p \geq (u-1)p$, thus we can complete all jobs in $R - \{j'\}$ by the time S_j^{ADV} and schedule j and j' as ADV. Consequently, $P \subseteq R \cup \{j\}$ is feasible at $r_k + p \leq C_k^{\mathrm{ADV}}$, contradicting the assumption that k did not cause preemption.

Now we show that no job in R is free. If any of $j' \in R$ is scheduled in ADV, using (ii) and (i) above we have $S_{j'}^{\mathrm{ADV}} \leq S_j^{\mathrm{ADV}} - p < C_k^{\mathrm{ADV}} + (u-1)p < C_j^A + up$, but at this time TightRestart is not idle as there is not enough time to complete all u jobs in R between C_j^A and $S_{j'}^{\mathrm{ADV}}$.

Summarizing, A completes at least u jobs that are not free between C_j^A and S_j^{ADV}, while ADV schedules at most $u-1$ jobs between C_k^{ADV} and S_j^{ADV} (using (i)). Since $M(j) = k$, the definition of M implies that j is in the range of M. $\square$

Charging scheme. Let j be a job started at time t in ADV. Note that case (I) below always applies when A is not idle at t, so the cases exhaust all possibilities.

Case (I): $j = M(k)$ for some k: Charge j to k.

Case (II): Otherwise, if A and TightRestart are idle at t, i.e., j is free: Since (I) does not apply, Lemma 4.1 implies that $f(j)$ is defined. Charge $\frac{1}{2}$ of j to the occurrence of j in A and $\frac{1}{2}$ of j to the occurrence of $f(j)$ in A.

Case (III): Otherwise, if A is idle at t, but TightRestart is running a job k' which is later preempted by a job k: By Lemma 2.2(2), j is completed in A. The job k is urgent and thus it is completed as well. Charge $\frac{1}{2}$ of j to k and $\frac{1}{2}$ of j to the occurrence of j in A.

Analysis. We prove that each job scheduled in A is charged at most $\frac{3}{2}$. Each job is charged at most 1 in case (I), as M defines a matching.

We claim that the total charge from cases (II) and (III) is $\frac{1}{2}$. The jobs j receiving charges in cases (II) and (III) are obviously distinct. The case analysis below shows that the other jobs receiving charges in (II) and (III) can uniquely determine the corresponding j and that if they are scheduled in ADV then (I) applies to them and thus they cannot play the role of j in (II) and (III).

In (II), $f(j)$ either is started in ADV during the execution of j in A, or it is not executed in ADV at all and arrives during the execution of j in A; this uniquely determines the corresponding j. Also, in the first case, at $S^{\mathrm{ADV}}_{f(j)}$, A is running j, and thus (I) applies to $f(j)$. By the definition of f, it is not a preemption candidate, so it cannot play the role of k in (III).

In (III), job k, as a preemption candidate, is tight, and since it preempts another job, $S^A_k = r_k$. Thus if ADV schedules k, at S^{ADV}_k, A is executing k, and (I) applies to k. The corresponding job j is uniquely determined as the job j running in ADV at time r_k.

We conclude that each job completed in A gets at most one charge of $\frac{1}{2}$ and thus is charged a total of at most $\frac{3}{2}$. The competitive ratio of $\frac{3}{2}$ now follows by summing the charges over all jobs.

Theorem 4.2 TIGHTRESTART *is a* $\frac{3}{2}$*-competitive algorithm with restarts for scheduling equal-length jobs.*

4.2 A Lower Bound

Theorem 4.3 *For scheduling equal-length jobs with restarts, no deterministic algorithm is less than* $\frac{3}{2}$*-competitive and no randomized algorithm is better than* $\frac{6}{5}$*-competitive.*

Proof: For $p \geq 2$, consider four jobs given in the form $j = (r_j, d_j)$: $1 = (0, 3p+1)$, $2 = (1, 3p)$, $3 = (p, 2p)$, $4 = (p+1, 2p+1)$. There exist schedules that schedule three jobs 1,3,2 or three jobs 2,4,1, in this order. (See Fig. 4.)

In the deterministic case, release jobs 1 and 2. If at time 1, job 2 is started in the online algorithm, release job 3, otherwise release job 4. The online algorithm completes only 2 jobs. As the optimum is 3, the competitive ratio is no better than $\frac{3}{2}$.

For the randomized case, we use Yao's principle. Always release jobs 1 and 2, and then a randomly chosen one job from 3 and 4, each with probability $\frac{1}{2}$. The expected number of completed jobs of any deterministic algorithm is at most 2.5, as on one of the instances it completes only 2 jobs. Thus the competitive ratio is no better than $\frac{3}{2.5} = \frac{6}{5}$. □

5 Conclusions

For equal processing times, closing the gap between our upper bound of $\frac{5}{3}$ and the lower bound of $\frac{4}{3}$ is a challenging open problem. It would also be interesting to close these gaps for barely random algorithms which – in our view – are of their own interest (even in the case when we use one fair random bit.)

Beyond our simple lower bound of $\frac{6}{5}$, nothing is known about the effect of allowing both randomness and restarts.

References

[1] P. Baptiste. Polynomial time algorithms for minimizing the weighted number of late jobs on a single machine with equal processing times. *Journal of Scheduling*, 2:245–252, 1999.

[2] Y. Bartal, M. Chrobak, and L. L. Larmore. A randomized algorithm for two servers on the line. *Information and Computation*, 158:53–69, 2000.

[3] J. Carlier. Problèmes d'ordonnancement à durées égales. *QUESTIO*, 5(4):219–228, 1981.

[4] M. Chrobak, C. Dürr, W. Jawor, Ł. Kowalik, and M. Kurowski. A note on scheduling equal-length jobs to maximize throughput. manuscript, 2004.

[5] M. Garey, D. Johnson, B. Simons, and R. Tarjan. Scheduling unit-time tasks with arbitrary release times and deadlines. *SIAM Journal on Computing*, 10(2):256–269, 1981.

[6] S. A. Goldman, J. Parwatikar, and S. Suri. Online scheduling with hard deadlines. *Journal of Algorithms*, 34:370–389, 2000.

[7] M. H. Goldwasser. Patience is a virtue: The effect of slack on the competitiveness for admission control. *Journal of Scheduling*, 6:183–211, 2003.

[8] M. H. Goldwasser and B. Kerbikov. Admission control with immediate notification. *Journal of Scheduling*, 6:269–285, 2003.

[9] H. Hoogeveen, C. N. Potts, and G. J. Woeginger. On-line scheduling on a single machine: Maximizing the number of early jobs. *Operations Research Letters*, 27:193–196, 2000.

[10] R. J. Lipton and A. Tomkins. Online interval scheduling. In *Proc. 5th Symp. on Discrete Algorithms (SODA)*, pages 302–311. ACM/SIAM, 1994.

[11] S. Seiden. Barely random algorithms for multiprocessor scheduling. *Journal of Scheduling*, 6:309–334, 2003.

[12] B. Simons. A fast algorithm for single processor scheduling. In *IEEE 19th Annual Symposium on Foundations of Computer Science*, pages 246–252, 1978.

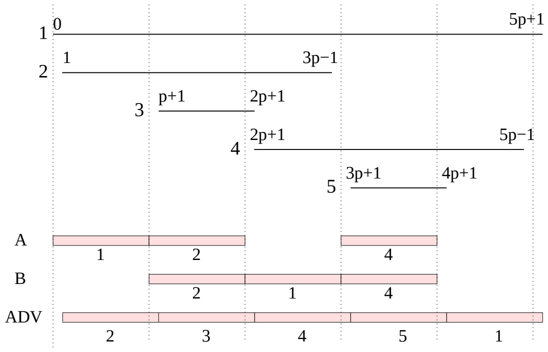

Figure 2: An instance on which RandLock schedules 3 jobs out of 5. Job 1 is executed as flexible by both processes, the other jobs are executed as urgent.

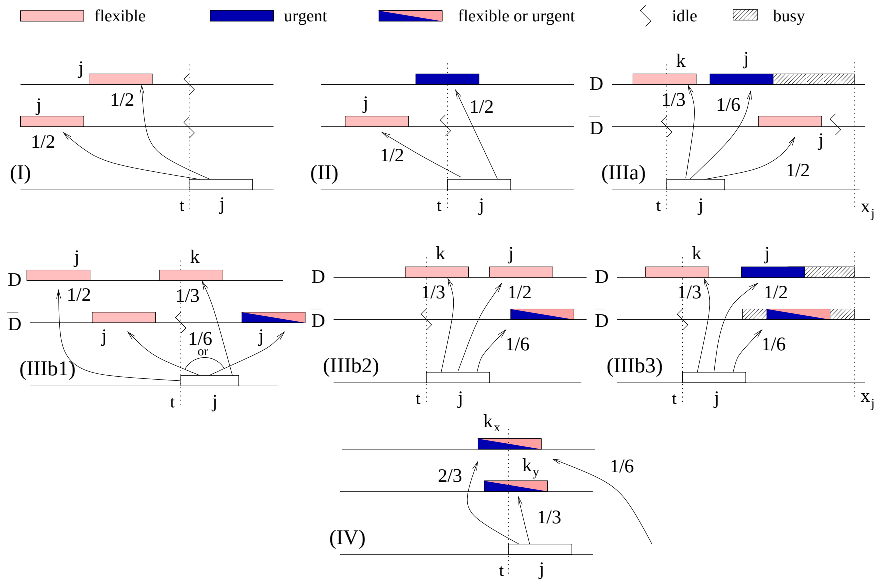

Figure 3: Illustration of the charging scheme in the analysis of Algorithm RandLock. The figure gives examples of different types of charges. In case (IIIb), there are several illustrations that cover possibilities playing a different role in the proof.

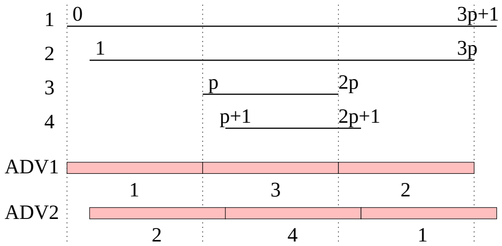

Figure 4: Instances used in the lower bounds with restarts.