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 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 barely random $\frac{5}{3}$-competitive algorithm that uses only one random bit. We also show a lower bound of $\frac{3}{2}$ on the competitive ratio of barely random algorithms that randomly choose one of two deterministic algorithms. If the two algorithms are selected with equal probability, we can further improve the bound to $\frac{8}{5}$.

Second, 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, where each job j is specified by its release time r_j and deadline d_j. (All numbers are assumed to be positive integers.) 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*, 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 schedule.

*Department of Computer Science, University of California, Riverside, CA 92521, U.S.A. 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 IAA1019401 of GA AV ČR. {sgall,tichy}@math.cas.cz

Our results. It is known that a simple greedy algorithm is 2-competitive for this problem, and that this ratio is optimal for deterministic algorithms. We present two ways to improve the competitive ratio of 2.

First, addressing an open question in [12, 13], we give a $\frac{5}{3}$-competitive randomized algorithm. Interestingly, our algorithm is *barely random*; it chooses with probability $\frac{1}{2}$ one of two deterministic algorithms, i.e., it uses only one random bit. These two algorithms are two *identical* copies of the same deterministic algorithm, that are run concurrently and use a shared lock to break the symmetry and coordinate their behaviors. We are not aware of previous work in the design of randomized online algorithms that uses such mechanism to coordinate identical algorithms—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 choose one of two deterministic algorithms, with any probability. If these algorithms are chosen with probability $\frac{1}{2}$ each, we improve the lower bound to $\frac{8}{5}$.

Second, 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.

An extended abstract of this paper appeared as [9].

Previous work. The problem of scheduling equal-length jobs to maximize the number of completed jobs has been well studied in the literature. In the offline case, an $O(n \log n)$-time algorithm for the feasibility problem (checking if *all* jobs can be completed) was given by Garey *et al.* [11] (see also [20, 7]). The maximization version can also be solved in polynomial time [8, 2], although the known algorithms are rather slow. (Carlier [7] claimed an $O(n^3 \log n)$ algorithm but, as pointed out in [8], his algorithm is not correct.)

As the first positive result on the online version, Baruah *et al.* [4, 5] show that a deterministic greedy algorithm is 2-competitive; in fact, they show that 2-competitiveness holds for any non-preemptive deterministic algorithm that is never idle at times when jobs are available for execution.

Goldman *et al.* [12] gave a lower bound of $\frac{4}{3}$ on the competitive ratio of randomized algorithms and the tight bound of 2 for deterministic algorithms. We briefly sketch these lower bounds, as they illustrate well what situations an online algorithm needs to avoid in order to achieve a small competitive ratio. Let $p \geq 2$. The jobs, written in the form $j = (r_j, d_j)$, are $1 = (0, 2p+1)$, $2 = (1, p+1)$, $3 = (p, 2p)$. The instance consists of jobs 1,2 or jobs 1,3; in both cases the optimum is 2. Figure 1 illustrates the input instance and the adversary strategy. (In this figure, and later throughout the paper, the horizontal dimension corresponds to the time axis, each job j in the input instance is drawn as a line segment spanning the interval $[r_j, d_j]$, and jobs that appear in the schedules are represented by rectangles of length p positioned at the actual time of execution.) In the deterministic case, release job 1; if at time 0 the online algorithm starts job 1, then release

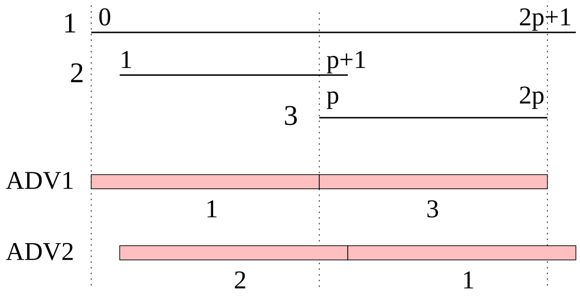

Figure 1: Jobs used in the lower bound proof.

job 2, otherwise release job 3. The online algorithm completes only one job and the competitive ratio is no better than 2. In the randomized case, using Yao's principle, we choose each of the two instances with probability $\frac{1}{2}$. The expected number of completed jobs of any deterministic online 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 $2/1.5 = \frac{4}{3}$.

Goldman *et al.* [12] show that the lower bound of 2 can be beaten if the jobs on input have sufficiently large "slack"; more 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 from [12]. Exploring further this direction, Goldwasser [13] obtained a parameterized extension 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 is $1 + 1/\lambda$.

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 preemptions may be allowed. For example, once arbitrarily processing times and/or weights are introduced, no constant-competitive non-preemptive 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.* [15]. They present a 2-competitive deterministic algorithm with restarts for jobs with arbitrary processing times and objective to maximize the number of completed jobs. They also give a matching lower bound. 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 algorithms have been found quite effective. Most randomized algorithms in the general scenarios use the classify-and-randomly-select technique by Lipton and Tomkins [17]. Typically, this method decreases the dependence of competitive ratio from linear to logarithmic in certain parameters (e.g., the maximum ratio between job weights), but it does not apply to the case of jobs with equal lengths and weights. Our randomized algorithm is based on entirely different ideas.

Barely random algorithms have been successfully applied in the past to a variety of online problems, including the list update problem [18], the k-server problem [3] and makespan scheduling [1, 10, 19]. In particular, the algorithm of Albers [1] involves two deterministic processes in which the second one keeps track of the first and corrects its potential "mistakes"—a coordination idea somewhat similar to ours, although in [1] the two processes are not symmetric.

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 a job j is $x_j = d_j - p$, i.e., the last time when it can be started. A job j is called *admissible* 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* $\mathcal{A}$ assigns to each completed job j an interval $[S_j^{\mathcal{A}}, C_j^{\mathcal{A}})$, with $r_j \leq S_j^{\mathcal{A}} \leq x_j$ and $C_j^{\mathcal{A}} = S_j^{\mathcal{A}} + p$, during which j is executed. These intervals are disjoint for distinct jobs. $S_j^{\mathcal{A}}$ and $C_j^{\mathcal{A}}$ are called the *start time* and *completion time* of job j. Without loss of generality, both are assumed to be integer. 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 $\mathcal{A}$, a job is *pending* at time t in $\mathcal{A}$ if it is admissible at t (that is, $r_j \leq t \leq x_j$) but not yet completed in $\mathcal{A}$. Note that according to this definition a job that is being executed at t may also be considered pending. When $\mathcal{A}$ is understood from context, we will typically use notation P_t to denote the set of jobs pending at time t.

For any set of jobs Q, we say that that Q is *feasible* at time t if there exists a schedule which completes all jobs in Q such that no job is started before t. Q is *flexible* at time t if it is feasible at time $t + p$.

Applying the Jackson rule [16], it is quite easy to determine whether a set P of pending jobs is feasible at t: Order the jobs in P in order of increasing deadlines, and schedule them at times t, $t+p$, $t+2p$, etc. Then P is feasible if and only if all jobs in P meet their deadlines. Furthermore, if we want to compute the maximum-size feasible subset $P' \subseteq P$, we can start with $P' = \emptyset$, and then add jobs $j \in P - P'$ to P', one by one and in arbitrary order, as long as P' remains feasible.

This means, in particular, that P' is a maximum-size feasible subset of P iff P' is a $\subseteq$-maximal feasible subset of P. All those properties can be proven by elementary exchange arguments, and the proofs are left to the reader.

We say that a job started by a schedule $\mathcal{A}$ at time t is *flexible in* $\mathcal{A}$ if the set of all jobs pending in $\mathcal{A}$ at t is flexible; otherwise the job is called *urgent*. Intuitively, a job is flexible if we could possibly postpone it and stay idle for time p, without losing any of the currently pending jobs; this could improve the schedule if a tight job arrives. On the other hand, postponing an urgent job can bring no advantage to the algorithm.

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, including its deadline, is revealed to the algorithm at its release time r_j. A *non-preemptive online algorithm* can start a job only when no job is running; thus, if a job is started at time t the algorithm has no choice but to let it run to completion at 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 final schedule, which is considered to be idle during this time interval. Thus the final schedule generated by an online algorithm with restarts is non-preemptive.

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

The definitions above assume the model—standard in the scheduling literature—with integer release times and deadlines, which implicitly makes the time discrete. Some papers on real-time scheduling work with continuous time. Both our algorithms can be modified to the continuous time model and unit processing time jobs without any changes in performance, at the cost of somewhat more technical presentation.

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 $\mathcal{A}$ is called *earliest-deadline-first* (or *EDF*) if, whenever it starts a job, it chooses the ED job of all the pending jobs that are later completed in $\mathcal{A}$. (Note that this may not be the overall ED pending job.)

A schedule $\mathcal{A}$ is *normal* if it satisfies the following two properties:

- (n1) when $\mathcal{A}$ starts a job, it chooses the ED job from the set of all pending jobs;
- (n2) if the set of all pending jobs in $\mathcal{A}$ at some time t is not flexible, then some job is running at t.

Obviously, any normal schedule is EDF, but the reverse is not true. All algorithms presented in this paper generate normal schedules. The properties (n1) and (n2) are reasonable, as the online algorithm cannot make a mistake by enforcing them. Formally, any online algorithm can be modified, using a standard exchange argument, to produce normal schedules without reducing the number of scheduled jobs. (We omit the proof, as we do not need this fact in the paper.)

The following property will be crucial in our proofs.

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

Proof: Denote by P the set of jobs pending at time $S_j^{\mathcal{A}}$ (including j). By the assumption about j, P is not flexible at $S_j^{\mathcal{A}}$. Towards contradiction, suppose that $\mathcal{A}$ is idle or starts a flexible job at time t, where $S_j^{\mathcal{A}} \leq t \leq x_j$. Then the set Q of jobs pending at time t is flexible at t. Since j is the ED job from P (by the normality of $\mathcal{A}$) and $t \leq x_j$, all other jobs in P have not expired until t, and thus Q contains all the jobs from P that are not completed in $\mathcal{A}$ until time t.

Using the above properties, we can rearrange the schedule as follows. Since Q is flexible at t, we can schedule all jobs of Q at time $t+p$ or later, start j at t and schedule all jobs in $P - Q - \{j\}$ as in $\mathcal{A}$. But this shows that P is flexible at time $S_j^{\mathcal{A}}$—a contradiction. □

Two schedules $\mathcal{D}$ and $\mathcal{D}'$ for an instance J are called *equivalent* if they satisfy the following conditions for each time t:

(eq1) $\mathcal{D}$ starts a job at t if and only if $\mathcal{D}'$ starts a job at t.

(eq2) Suppose that $\mathcal{D}$ starts a job j at time t and $\mathcal{D}'$ starts a job j' at time t. Then j is flexible in $\mathcal{D}$ iff j' is flexible in $\mathcal{D}'$. Furthermore, if they are both flexible then $j = j'$.

Obviously, if $\mathcal{D}$, $\mathcal{D}'$ are equivalent, then $|\mathcal{D}| = |\mathcal{D}'|$.

To facilitate competitive 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 we can include in the schedule, if there are more several choices. The modified schedules are no longer normal (only EDF, which is a weaker requirement); nevertheless, as they are equivalent to normal schedules, they inherit some of their important properties including Lemma 2.1.

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

(1) *All jobs $f(j)$ are completed in $\mathcal{A}$.*

(2) *Consider a time t when either $\mathcal{A}$ is idle or it starts a job and the set of all its pending jobs is feasible at t. Then all jobs pending at t are completed in $\mathcal{A}$. In particular, each job that is pending when $\mathcal{A}$ starts a flexible job is completed in $\mathcal{A}$.*

(3) *Let j be a job completed in $\mathcal{A}$, and let U be the set of all jobs j' with $r_{j'} < C_j^{\mathcal{A}}$ that are pending at time $C_j^{\mathcal{A}}$. (Note that the jobs released at time $C_j^{\mathcal{A}}$ are not included in U.) If U is feasible at time $C_j^{\mathcal{A}}$ then all the jobs in U are completed in $\mathcal{A}$.*

Furthermore, if $\mathcal{X}$ is constructed by an online algorithm and $f(j)$ can be determined online at time $C_j^{\mathcal{X}}$ for each flexible job j in $\mathcal{X}$, then $\mathcal{A}$ can be produced by an online algorithm.

Remarks: Property (1) is useful in our proofs, since it allows us to modify the schedule computed by the algorithm to resemble more the optimal schedule. Property (2) guarantees that any job planned to be scheduled is indeed scheduled in the future. Property (3) is a technical condition needed in the analysis of the algorithm with restarts.

Since $\mathcal{A}$ and $\mathcal{X}$ are equivalent, flexible jobs are the same and scheduled at the same times in $\mathcal{A}$ and $\mathcal{X}$. In particular, all the jobs j on which $f(j)$ is defined are scheduled at the same time both in $\mathcal{A}$ and $\mathcal{X}$—a property that will play an important role in our later arguments.

The basic idea of the construction of $\mathcal{A}$ is quite straightforward: Maintain a set Q_t of jobs that we plan to schedule. If the set of all pending jobs is feasible, we plan to schedule them all. In particular, if we start a flexible job j at time t, the flexibility of j allows us to add to Q_{t+p} an extra job released during the execution of j; so if $f(j)$ is defined, we add $f(j)$. Similarly, to achieve (3), we can add all jobs released during the execution of j, as long as the resulting set remains feasible.

Proof: We construct $\mathcal{A}$ iteratively. Throughout the proof, t ranges over times when $\mathcal{X}$ is idle or starts a job. For each such t, let P_t and P'_t denote the set of jobs pending in $\mathcal{X}$ and $\mathcal{A}$, respectively.

We will maintain an auxiliary set of jobs Q_t that are pending at t in $\mathcal{X}$ and $\mathcal{A}$, that is $Q_t \subseteq P_t \cap P'_t$. Simultaneously with the construction, we prove inductively that, for all t, the following invariant holds:

($*$) Q_t is an $\subseteq$-maximal feasible subset of each of P_t and P'_t.

Before describing the construction, we make two observations. First, recall that condition ($*$) implies that Q_t is also maximum with respect to size. Second, if any of sets P_t, P'_t, Q_t is flexible, then $Q_t = P_t = P'_t$, by the maximality of Q_t.

We now describe the construction. Initially, choose Q_0 as an arbitrary maximal feasible set of jobs released at time 0.

Assume we have already defined Q_t. If $\mathcal{X}$ is idle at t, we let $\mathcal{A}$ idle and choose an arbitrary $Q_{t+1} \supseteq Q_t$ by adding to Q_t the jobs released at $t+1$, as long as the set remains feasible. Since $\mathcal{X}$ is idle, P_t is flexible at t, and thus $Q_t = P_t = P'_t$ and Q_t is feasible at $t+1$. Thus $P_{t+1} = P'_{t+1}$ and ($*$) holds at time $t+1$.

Now suppose that $\mathcal{X}$ starts a job j at time t. We consider two subcases, depending on whether j is flexible or urgent.

Case 1: j is flexible in $\mathcal{X}$. Then $\mathcal{A}$ starts j, too. In this case, $Q_t = P_t = P'_t$, and thus $P'_{t+1} = P_{t+1}$ as well. Clearly, j is flexible in $\mathcal{A}$ as well. Since Q_t is flexible, it is feasible at $t+p$. To construct Q_{t+p}, we start with $Q_{t+p} = Q_t - \{j\}$ and expand it by adding newly released jobs, one by one, as long as Q_{t+p} remains feasible. The jobs are added in the following order:

(i) First, add $f(j)$, if it is defined, pending, and not yet in Q_{t+p}.

(ii) Next, add all jobs j' with $t < r_{j'} < t+p$, in an arbitrary order.

(iii) Finally, add all jobs j' with $r_{j'} = t+p$, in an arbitrary order.

Since Q_t is feasible at $t+p$, $Q_t - \{j\} \cup \{f(j)\}$ is feasible at $t+p$ as well, so $f(j)$ can be added to Q_{t+p} in (i) without violating feasibility. By the construction, ($*$) is satisfied at time $t+p$.

Case 2: j is urgent in $\mathcal{X}$. $\mathcal{A}$ starts the earliest-deadline (more precisely, $\prec$-minimal) job k from Q_t. Since Q_t is a maximal feasible set both for $\mathcal{X}$ and $\mathcal{A}$, it is non-empty whenever $\mathcal{X}$ starts a job. Furthermore, we know that Q_t is not flexible at t and thus k is urgent.

Let $T = Q_t - \{k\}$. We claim that:

(t1) T is a maximal subset of P_t (resp. P'_t) that is feasible at time $t+p$, and

(t2) $T \subseteq P_{t+p} \cap P'_{t+p}$.

That T is feasible at $t+p$ follows directly from the definition of T and the fact that k is the

ED job in Q_t. For the same reason, all jobs in T are pending in $\mathcal{A}$ at time $t+p$. Since k is pending in $\mathcal{X}$ at t, and $\mathcal{X}$ schedules the ED pending job (as $\mathcal{X}$ is normal), we have $j \prec k$. Therefore all jobs in T are pending at time $t+p$ in $\mathcal{X}$ as well. We conclude that (t2) holds.

No job in $P'_t - Q_t$ can be feasibly added to T at time $t+p$, as otherwise it could be feasibly added to Q_t at time t, contradicting the maximality of Q_t for $\mathcal{A}$. The same argument applies to $\mathcal{X}$. Thus, T satisfies condition (t1) for both P_t and P'_t.

We construct Q_{t+p} similarly as in the previous case. We start with $Q_{t+p} = T$ and add newly released jobs, first the jobs released before $C_j^{\mathcal{X}} = t+p$, and then the jobs released at $C_j^{\mathcal{X}}$, one by one, as long as Q_{t+p} remains feasible. Again, the maximality of T and the construction implies that Q_{t+p} satisfies $(*)$ at time $t+p$.

This completes the construction. Obviously, $\mathcal{X}$ and $\mathcal{A}$ are equivalent. Also, $\mathcal{A}$ is EDF since whenever it schedules a job, it chooses the ED job of Q_t, and jobs $j' \in P'_t - Q_t$ are never added to Q_s for $s > t$, so they will not be scheduled in $\mathcal{A}$.

By the construction, $\mathcal{A}$ schedules all the jobs that are in some plan Q_t. At any time t when $\mathcal{A}$ is idle or starts a flexible job, Q_t is flexible and thus $Q_t = P'_t$. This proves (2). Since $f(j) \in Q_{t+p}$ for $t = S_j^{\mathcal{X}}$, this also implies (1). Finally, to show (3), recall that when constructing Q_{t+p} for $t = S_j^{\mathcal{X}}$, we are first adding to Q_{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 $U \subseteq Q_{t+p}$. □

Lemma 2.2 gives an easy proof that any normal schedule $\mathcal{X}$ schedules at least half as many jobs as the optimum. Take the modified schedule $\mathcal{A}$ from Lemma 2.2 (with f undefined). Charge any job j completed in an optimal schedule $\mathcal{Z}$ to a job completed in $\mathcal{A}$ as follows: (a) If $\mathcal{A}$ is running a job k at time $S_j^{\mathcal{Z}}$, charge j to k. (b) Otherwise charge j to j. This is well defined since, if j is admissible and $\mathcal{A}$ is idle at time $S_j^{\mathcal{Z}}$, then $\mathcal{A}$ completes j by Lemma 2.2(2). Furthermore, only one job can be charged to k using rule (a), as all jobs have the same processing time and only one job can be started in $\mathcal{Z}$ during the interval when k is running in $\mathcal{A}$. Thus overall at most two jobs in $\mathcal{Z}$ are charged to each job in $\mathcal{A}$ and $|\mathcal{Z}| \le 2|\mathcal{A}| = 2|\mathcal{X}|$, 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.

Goldwasser and Kerbikov [14] introduced a concept of online algorithms that upon release of a job immediately commit whether it will be completed or not. We do not formulate our algorithms in this form, but Lemma 2.2 can be applied to normal schedules generated by our algorithms (with f undefined) to obtain equivalent online algorithms with immediate notification. (For the model with restarts, this implies that any preempted job is completed later.) Since the construction produces equivalent schedules, the performance is also the same.

3 Randomized Algorithms

In this section we present our $\frac{5}{3}$-competitive barely random algorithm. This algorithm uses only one random bit; namely, at the beginning of computation it chooses with probability $\frac{1}{2}$ between two deterministic algorithms. We also 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. We describe our algorithm in terms of two identical processes that are denoted by $\mathcal{X}$ and $\mathcal{Y}$. Each process is, in essence, a scheduling algorithm that receives its own copy of the input instance J and computes its own schedule for J. (This means that a given job can be executed by both processes, at the same or different times.) We chose to use the term "process" rather than "algorithm", since $\mathcal{X}$ and $\mathcal{Y}$ are not fully independent; they both have access to a shared lock mechanism used to coordinate their behavior.

Each process $\mathcal{X}$ and $\mathcal{Y}$ is defined as follows:

(RL1) If there are no pending jobs, wait until some job is released.

(RL2) If the set of pending jobs is not flexible, execute the ED pending job.

(RL3) If the set of pending jobs is flexible and the lock is available, acquire the lock (ties broken arbitrarily), execute the ED pending job, and release the lock upon its completion.

(RL4) Otherwise, wait until the lock becomes available or the set of pending jobs becomes non-flexible (due to progress of time or new jobs being released).

Algorithm RANDLOCK selects initially one of the two processes $\mathcal{X}$ or $\mathcal{Y}$, each with probability $\frac{1}{2}$. Then it simulates the two processes on a given instance, outputting the schedule generated by the selected process.

Before we analyze the algorithm, we illustrate its behavior on the instance in Figure 2. Both processes schedule only three jobs, while the optimal schedule has five jobs. Thus RANDLOCK is not better than $\frac{5}{3}$-competitive.

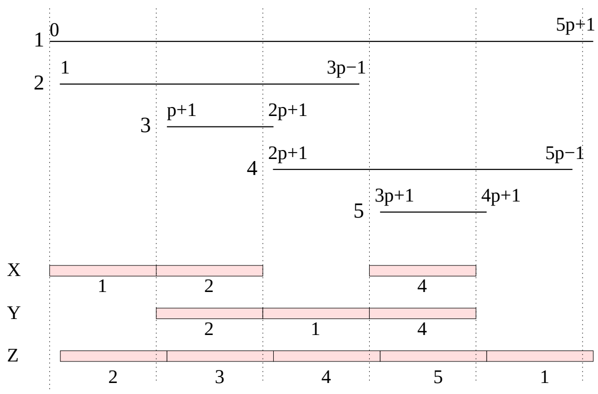

Figure 2: An instance on which RANDLOCK schedules three jobs out of five. Job 1 is executed as flexible by both processes; the other jobs are executed as urgent.

Lemma 3.1 *Let $\mathcal{D} \in \{\mathcal{X}, \mathcal{Y}\}$, and let $\bar{\mathcal{D}}$ be the other process of* RANDLOCK*. Suppose that $\mathcal{D}$ is idle at time t and the lock is available in this time step (i.e., $\bar{\mathcal{D}}$ is idle or is executing an urgent job). Then each job admissible at time t is completed in $\mathcal{D}$ as a flexible job by time t.*

Proof: This is a direct consequence of the way $\mathcal{X}$ and $\mathcal{Y}$ coordinate their actions. If a job j is admissible and pending at time t in $\mathcal{D}$, and if $\bar{\mathcal{D}}$ does not have the lock, then $\mathcal{D}$ would have acquired the lock and scheduled j or some other job at time t. Furthermore, if j is executed as urgent in $\mathcal{D}$ before time t then, since $S_j^{\mathcal{D}} \leq t \leq x_j$, Lemma 2.1 implies that $\mathcal{D}$ cannot be idle at time t. $\Box$

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

Proof: Overloading the notation, let $\mathcal{X}$ and $\mathcal{Y}$ denote the schedules generated by the corresponding processes on a given instance J. By rules (RL2), (RL3), both schedules are normal. Fix an arbitrary schedule $\mathcal{Z}$ for the given instance J.

We start by modifying the schedules $\mathcal{X}$ and $\mathcal{Y}$ according to Lemma 2.2. Define a partial function $f^{\mathcal{A}} : J \to J$ as follows. Let $f^{\mathcal{A}}(j) = k$ if j is a flexible job completed in $\mathcal{X}$ and k is a job started in $\mathcal{Z}$ during the execution of j in $\mathcal{X}$ and admissible at the completion of j in $\mathcal{X}$, i.e., $S_j^{\mathcal{X}} \leq S_k^{\mathcal{Z}} < C_j^{\mathcal{X}} \leq x_k$. Otherwise (if j is urgent or no such k exists), $f^{\mathcal{A}}(j)$ is undefined. Note that if k exists, it is unique for a given j. Then we define $\mathcal{A}$ to be the schedule constructed from $\mathcal{X}$ in Lemma 2.2 using function $f^{\mathcal{A}}(\cdot)$. Analogously we define function $f^{\mathcal{B}}(\cdot)$, and we modify schedule $\mathcal{Y}$ to obtain schedule $\mathcal{B}$. We stress that these new schedules $\mathcal{A}$ and $\mathcal{B}$ cannot be constructed online as their definition depends on $\mathcal{Z}$; they only serve as tools for the analysis of RANDLOCK.

Since $\mathcal{A}$ (resp. $\mathcal{B}$) is equivalent to a normal schedule $\mathcal{X}$ (resp. $\mathcal{Y}$), Lemma 2.1 still applies to $\mathcal{A}$ (resp. $\mathcal{B}$) and the number of completed jobs remains the same as well.

Throughout the proof we use the convention that whenever $\mathcal{D}$ denotes one of the schedules $\mathcal{A}$ and $\mathcal{B}$, then $\bar{\mathcal{D}}$ denotes the other one.

The charging scheme. Our proof is based on a charging scheme. The fundamental principle of this scheme is the same as in the proof for the greedy algorithm in Section 2. Each adversary job will generate a charge of 1. This charge will be distributed among the jobs in schedules $\mathcal{A}$ and $\mathcal{B}$, in such a way that each job in these schedules will receive a charge of at most $\frac{5}{6}$. This will imply the $\frac{5}{3}$ bound on the competitive ratio of RANDLOCK.

Let j be a job started in $\mathcal{Z}$ at time $t = S_j^{\mathcal{Z}}$. This job generates several charges of different weights to (the occurrences of) the jobs in schedules $\mathcal{A}$ and $\mathcal{B}$. There are two types of charges: *self-charges* from job j to the occurrences of j in $\mathcal{A}$ or $\mathcal{B}$, and *up-charges*, from j to the jobs running at time t in $\mathcal{A}$ and $\mathcal{B}$. If one of the processes runs j at time t, then the charge to this job may be either an up-charge or a self-charge; in case (III) below such a j can even receive both a self-charge and an up-charge from j. The total of charges generated by j is always 1. The charges depend on the status of $\mathcal{A}$ and $\mathcal{B}$ at time t. (See Figure 3.)

Case (I): Both schedules $\mathcal{A}$ and $\mathcal{B}$ are idle. By Lemma 3.1, in both $\mathcal{A}$ and $\mathcal{B}$, j is completed as flexible by time t. We generate two self-charges of $\frac{1}{2}$ to the two occurrences of j in $\mathcal{A}$ and $\mathcal{B}$.

Case (II): One schedule $\mathcal{D} \in \{\mathcal{A}, \mathcal{B}\}$ is running an urgent job k and the other schedule $\bar{\mathcal{D}}$ is idle. By Lemma 3.1, in $\bar{\mathcal{D}}$, j is completed as flexible by time t. We generate a self-charge of $\frac{1}{2}$ to the occurrence of j in $\bar{\mathcal{D}}$ and an up-charge of $\frac{1}{2}$ to k in $\mathcal{D}$.

Case (III): One schedule $\mathcal{D} \in \{\mathcal{A}, \mathcal{B}\}$ is running a flexible job k and the other schedule $\bar{\mathcal{D}}$ is idle. We claim that j is completed in both $\mathcal{A}$ and $\mathcal{B}$. For $\bar{\mathcal{D}}$, this follows directly from

Lemma 2.2(2). We now prove it for $\mathcal{D}$. If $r_j \leq S_k^{\mathcal{D}}$, then Lemma 2.2(2) applied to time $t' = S_k^{\mathcal{D}}$ implies that $\mathcal{D}$ completes j. If $x_j \geq C_k^{\mathcal{D}}$ then $f^{\mathcal{D}}(k) = j$, so $\mathcal{D}$ completes j by Lemma 2.2(1). In the remaining case, we have $S_k^{\mathcal{D}} < r_j \leq t \leq x_j \leq C_k^{\mathcal{D}}$. This means that j is an urgent job pending at time t and thus $\bar{\mathcal{D}}$ cannot be idle, contradicting the case condition.

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

Case (IIIa): If $\mathcal{E}$ schedules j as an urgent job and the other schedule $\bar{\mathcal{E}}$ is idle at some time t' satisfying $S_j^{\mathcal{E}} \leq t' \leq x_j$, then charge $\frac{1}{6}$ to the occurrence of j in $\mathcal{E}$ and $\frac{1}{2}$ to the occurrence of j in $\bar{\mathcal{E}}$.

We make a few observations that will be useful later in the proof. Since in this case j is urgent in $\mathcal{E}$, and $\mathcal{E}$ is either idle or executes a flexible job at time t, Lemma 2.1 implies that j is executed in $\mathcal{E}$ after time t. Further, since $\bar{\mathcal{E}}$ is idle at time t', j is executed in $\bar{\mathcal{E}}$ as flexible before time t'.

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

Case (IV): Both processes $\mathcal{A}$ and $\mathcal{B}$ are running jobs $k_{\mathcal{A}}$ and $k_{\mathcal{B}}$, respectively, at time t. We show below in Lemma 3.3 that one of $k_{\mathcal{A}}$, $k_{\mathcal{B}}$ receives a self-charge of at most $\frac{1}{6}$ from its occurrence in $\mathcal{Z}$. We generate an up-charge of $\frac{2}{3}$ from j to this job, and an up-charge of $\frac{1}{3}$ to the other one. No self-charge is generated in this case.

Lemma 3.3 *If case* (IV) *applies to j, then the self-charge to one of $k_{\mathcal{D}}$ in $\mathcal{D}$, for $\mathcal{D} \in \{\mathcal{A}, \mathcal{B}\}$, does not exceed $\frac{1}{6}$.*

Proof: Any self-charge has weight $\frac{1}{2}$ or $\frac{1}{6}$. Assume, towards contradiction, that both $k_{\mathcal{A}}$ and $k_{\mathcal{B}}$ receive a self-charge of $\frac{1}{2}$. At least one of $k_{\mathcal{A}}$ and $k_{\mathcal{B}}$ is scheduled as urgent in the corresponding schedule, due to the lock mechanism. Thus $k_{\mathcal{A}} \neq k_{\mathcal{B}}$, 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 $j = k_{\mathcal{D}}$, for some $\mathcal{D} \in \{\mathcal{A}, \mathcal{B}\}$, then $k_{\mathcal{D}}$ would not receive any self-charge. Thus $k_{\mathcal{A}}$, $k_{\mathcal{B}}$, and j are three distinct jobs.

Claim A: Let $\mathcal{F} \in \{\mathcal{A}, \mathcal{B}\}$, and suppose that $k_{\mathcal{F}}$ is scheduled before j in $\mathcal{Z}$ (that is, $S_{k_{\mathcal{F}}}^{\mathcal{Z}} \leq t - p$) and that it receives a self-charge in $\mathcal{F}$ of $\frac{1}{2}$ generated in Case (IIIb) applied to $k_{\mathcal{F}}$. Then $k_{\bar{\mathcal{F}}} \prec k_{\mathcal{F}}$.
Proof: Since $k_{\mathcal{F}}$ receives a charge of $\frac{1}{2}$ in (IIIb), the choice of $\mathcal{E}$ in Case (III) implies that $k_{\mathcal{F}}$ is executed in $\bar{\mathcal{F}}$ later than in $\mathcal{F}$, that is $S_{k_{\mathcal{F}}}^{\bar{\mathcal{F}}} \geq S_{k_{\mathcal{F}}}^{\mathcal{F}} > t - p$. On the other hand, $S_{k_{\bar{\mathcal{F}}}}^{\bar{\mathcal{F}}} \leq t$, so $k_{\mathcal{F}}$ must be executed in $\bar{\mathcal{F}}$ after $k_{\bar{\mathcal{F}}}$. Furthermore, $S_{k_{\bar{\mathcal{F}}}}^{\bar{\mathcal{F}}} > t - p \geq S_{k_{\mathcal{F}}}^{\mathcal{Z}} \geq r_{k_{\mathcal{F}}}$ and thus $k_{\mathcal{F}}$ is pending in $\bar{\mathcal{F}}$ when $k_{\bar{\mathcal{F}}}$ is started. Since $\bar{\mathcal{F}}$ is EDF, we have $k_{\bar{\mathcal{F}}} \prec k_{\mathcal{F}}$, completing the proof of Claim A. □

To finish the proof of the lemma, choose $\mathcal{D}$ such that $k_{\mathcal{D}}$ is urgent in $\mathcal{D}$ (as noted above, such $\mathcal{D}$ exists). The only case when an urgent job receives a self-charge of $\frac{1}{2}$ is (IIIb). By Lemma 2.1, $\mathcal{D}$ executes urgent jobs at all times t', $t \leq t' \leq x_{k_{\mathcal{D}}}$, which, together with the condition for Case

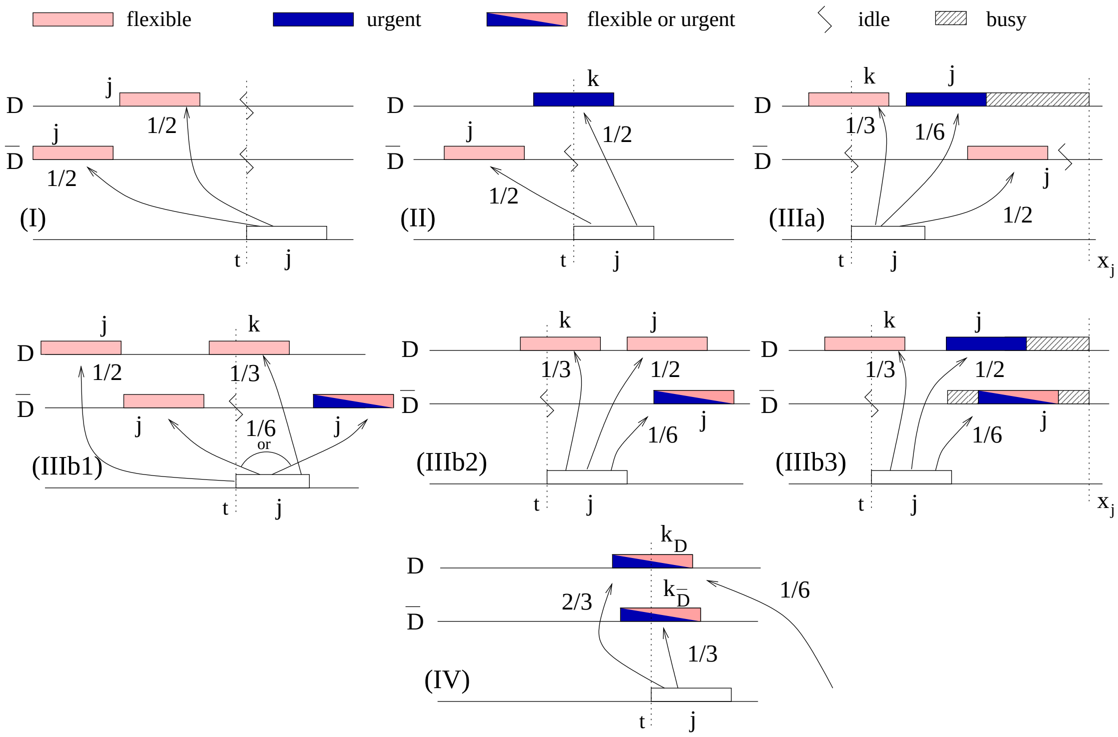

Figure 3: Illustration of the charging scheme in the analysis of Algorithm RandLock. The figure gives examples of different types of charges. In case (IIIb), there are several illustrations that cover possibilities playing a different role in the proof. (To reduce the number of cases, in the figures for Case (III) we assume that $j = k$ and j is executed in $\mathcal{D}$ before it is executed in $\bar{\mathcal{D}}$.)

(III) applied to $k_{\mathcal{D}}$, implies that $S^{\mathcal{Z}}_{k_{\mathcal{D}}} \le t$. As $j \neq k_{\mathcal{D}}$, it follows that $S^{\mathcal{Z}}_{k_{\mathcal{D}}} \le t - p$. By Claim A, $k_{\bar{\mathcal{D}}} \prec k_{\mathcal{D}}$ and $x_{k_{\bar{\mathcal{D}}}} \le x_{k_{\mathcal{D}}}$. Furthermore, since (IIIa) does not apply to $k_{\mathcal{D}}$, $\bar{\mathcal{D}}$ is also not idle at any time t', $t \le t' \le x_{k_{\mathcal{D}}}$.

We now consider the self-charge of $\frac{1}{2}$ to $k_{\bar{\mathcal{D}}}$ in $\bar{\mathcal{D}}$. If $k_{\bar{\mathcal{D}}}$ in $\mathcal{Z}$ generates a self-charge of $\frac{1}{2}$ in Case (I) or (II) to its occurrence in $\bar{\mathcal{D}}$, then we would have $S^{\bar{\mathcal{D}}}_{k_{\bar{\mathcal{D}}}} \le S^{\mathcal{Z}}_{k_{\bar{\mathcal{D}}}}$, and at least one of the processes would be idle at time $S^{\mathcal{Z}}_{k_{\bar{\mathcal{D}}}}$. But this is not possible, because the previous paragraph implies that both processes are busy between t and $x_{k_{\bar{\mathcal{D}}}}$. The same argument shows that this charge cannot be generated in Case (IIIa), and in the subcase of (IIIb) when $S^{\mathcal{Z}}_{k_{\bar{\mathcal{D}}}} > t$.

The last possibility is that $k_{\bar{\mathcal{D}}}$ is self-charged $\frac{1}{2}$ in the subcase of (IIIb) when $S^{\mathcal{Z}}_{k_{\bar{\mathcal{D}}}} \le t$. But then $S^{\mathcal{Z}}_{k_{\bar{\mathcal{D}}}} \le t - p$ as $j \neq k_{\bar{\mathcal{D}}}$, and Claim A above applies to $k_{\bar{\mathcal{D}}}$. However, the conclusion that $k_{\mathcal{D}} \prec k_{\bar{\mathcal{D}}}$ contradicts the linearity of $\prec$ as $k_{\mathcal{D}} \neq k_{\bar{\mathcal{D}}}$ and we have already shown that $k_{\bar{\mathcal{D}}} \prec k_{\mathcal{D}}$.

Summarizing, we get a contradiction in all the cases, completing the proof of the lemma. $\square$

Continuing the proof of the theorem, we now show that the total charge to each occurrence of a job in $\mathcal{A}$ or $\mathcal{B}$ is at most $\frac{5}{6}$. Suppose that k is executed in $\mathcal{D} \in \{\mathcal{A}, \mathcal{B}\}$. During the execution of k at most one job is started in $\mathcal{Z}$, thus k gets at most one up-charge in addition to a possible self-charge. If k does not receive any up-charge, it is self-charged $\frac{1}{2}$ or $\frac{1}{6}$, i.e., less than $\frac{5}{6}$.

If k receives an up-charge in (II), then k is an urgent job and, since $\bar{\mathcal{D}}$ is idle, it is already completed in $\bar{\mathcal{D}}$, so $S_k^{\bar{\mathcal{D}}} < S_k^{\mathcal{D}}$. The only case where the occurrence of k that is later in time is urgent and receives a self-charge is Case (IIIb), and in this case this self-charge is $\frac{1}{6}$. So the total charge would be at most $\frac{1}{6} + \frac{1}{2} < \frac{5}{6}$.

If a job 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 receives an up-charge in (IV), Lemma 3.3 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 expected number of jobs completed by RandLock is $\frac{1}{2}(|\mathcal{A}| + |\mathcal{B}|)$. Since each job in $\mathcal{A}$ and $\mathcal{B}$ receives a charge of at most $\frac{5}{6}$, and all jobs in $\mathcal{Z}$ generate a charge of 1, we have $\frac{5}{3} \cdot \frac{1}{2}(|\mathcal{A}| + |\mathcal{B}|) = \frac{5}{6}(|\mathcal{A}| + |\mathcal{B}|) \geq |\mathcal{Z}|$. This implies that RandLock is $\frac{5}{3}$-competitive. □

As discussed in the introduction, a lower bound of $\frac{4}{3}$ is known for randomized algorithms [12]. For barely random algorithms choosing one of two algorithms, we can improve this bound to $\frac{3}{2}$. Assuming further that the two algorithms are selected with equal probability, we can further improve the bound to $\frac{8}{5}$.

Theorem 3.4 *Suppose that $\mathcal{R}$ is a barely-random non-preemptive algorithm for scheduling equal-length jobs that chooses one of two deterministic algorithms. Then $\mathcal{R}$ is not better than $\frac{3}{2}$-competitive.*

Proof: Assume that $\mathcal{R}$ chooses randomly one of two deterministic algorithms, $\mathcal{A}$ and $\mathcal{B}$, with some arbitrary probabilities. Let $p \geq 3$ and write the jobs as $j = (r_j, d_j)$. We start with job $1 = (0, 4p)$. Let t be the first time when one of the algorithms, say $\mathcal{A}$, schedules job 1. If $\mathcal{B}$ schedules it at t as well, release a job $1' = (t+1, t+p+1)$; the optimum schedules both jobs while both $\mathcal{A}$ and $\mathcal{B}$ only one, so the competitive ratio is at least 2.

So we may assume that $\mathcal{B}$ is idle at t. Release job $2 = (t+1, t+2p+2)$. If $\mathcal{B}$ starts any job (1 or 2) at $t+1$, release job $3 = (t+2, t+p+2)$, otherwise release job $4 = (t+p+1, t+2p+1)$. $\mathcal{B}$ completes only one of the jobs 2, 3, 4. Since $\mathcal{A}$ is busy with job 1 until time $t+p$, it also completes only one of the jobs 2, 3, 4, as their deadlines are smaller than $t+3p$. So each of $\mathcal{A}$ and $\mathcal{B}$ completes at most two jobs.

The optimal schedule completes three jobs: If 3 is issued, schedule 3 and 2, back to back, starting at time $t+2$. If 4 is issued, schedule 2 and 4, back to back, starting at time $t+1$. In either case, two of jobs $2, 3, 4$ fit in the interval $[t+1, t+2+2p)$. If $t \geq p-1$, schedule job 1 at time 0, otherwise schedule job 1 at time $3p \geq t+2+2p$. Thus the competitive ratio of $\mathcal{R}$ is at least $\frac{3}{2}$. □

Theorem 3.5 *Suppose that $\mathcal{R}$ is a barely-random non-preemptive algorithm for scheduling equal-length jobs that chooses one of two deterministic algorithms, each with probability $\frac{1}{2}$. Then $\mathcal{R}$ is not better than $\frac{8}{5}$-competitive.*

Proof: Assume that $\mathcal{R}$ chooses one of two deterministic algorithms, $\mathcal{A}$ and $\mathcal{B}$, each with probability $\frac{1}{2}$. Let $p \geq 3$ and write the jobs in the format $j = (r_j, d_j)$. We start with job $1 = (0, 6p)$. Let t be the first time when one of the algorithms, say $\mathcal{A}$, schedules job 1. If $\mathcal{B}$ schedules it at t as well, release job $1' = (t+1, t+p+1)$; obviously the optimum schedules both jobs while both $\mathcal{A}$ and $\mathcal{B}$ only one, so the competitive ratio is at least 2.

In the remaining case, $\mathcal{B}$ is idle at t. Release job $2 = (t+1, t+p+1)$. If $\mathcal{B}$ does not execute both 1 and 2, then the ratio is at least 2. We may thus assume that $\mathcal{B}$ starts 2 at time $t+1$ and then starts 1 at some time $t' \geq t+p+1$. Release job $3 = (t'+1, t'+2p+2)$. If $\mathcal{A}$ starts job 3 at $t'+1$, release job $4 = (t'+2, t'+p+2)$, otherwise release $5 = (t'+p+1, t'+2p+1)$. By the choice of the last job, $\mathcal{A}$ can complete only one of the jobs $3, 4, 5$. Since $\mathcal{B}$ is busy with job 1 until time $t'+p \geq t'+3$, it also can complete only one of the jobs $3, 4, 5$, as their deadlines are strictly smaller than $t'+3p$. So $\mathcal{A}$ can complete 2 jobs only and $\mathcal{B}$ can complete 3 jobs.

The optimal schedule can complete all four released jobs. If 4 is issued, schedule $4, 3$, back to back, starting at time $t'+2$. If 5 is issued, schedule $3, 5$, back to back, starting at time $t'+1$. In either case, both jobs fit in the interval $[t'+1, t'+2p+2)$. This interval is disjoint with the interval $[t+1, t+p+1)$ when 2 is scheduled. Finally, these two intervals occupy length $3p+1$ of the interval $[0, 6p)$ and divide it into at most 3 contiguous pieces; thus one of the remaining pieces has length at least p and job 1 can be scheduled.

Summarizing, $\mathcal{R}$ completes at most $\frac{1}{2}(2+3) = 2.5$ jobs on average, while the optimal schedule completes 4 jobs. Therefore the competitive ratio is at least $4/2.5 = \frac{8}{5}$, as claimed. $\Box$

4 Scheduling with Restarts

Our algorithm with restarts is very natural. At any time, it greedily schedules the ED job. However, if a tight job arrives that 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 l a *preemption candidate* for j if $s < r_l \leq x_l < s+p$.

Algorithm TightRestart. At time t:

(TR1) If no job is running, start the ED pending job, providing there is at least one pending job, otherwise stay idle until some job is released.

(TR2) Otherwise, let j be the running job. If j was started as urgent or no preemption candidate is released at t, continue running j.

(TR3) Otherwise, let P_t^* be the set of all jobs pending at time t, including j but excluding any preemption candidates. If P_t^* is flexible at t, preempt j and start (at time t) a preemption candidate; choose the ED preemption candidate, if more are admissible at time t. Otherwise continue running j.

Let $\mathcal{X}$ be the final schedule generated by TightRestart, after removing the preempted parts of jobs. For any time t, as before, by P_t we denote the set of jobs that are pending in $\mathcal{X}$ at time t. We stress that we distinguish between $\mathcal{X}$ being idle and TightRestart being idle: at some time steps TightRestart can process a job that will be preempted later, in which case $\mathcal{X}$ is considered idle at these steps but TightRestart is not.

Lemma 4.1 *Schedule $\mathcal{X}$ is normal.*

Proof: By rules (TR1) and (TR3), TIGHTRESTART always starts the ED pending job; in (TR3) note that, by definition, any preemption candidate is tight and thus it has earlier deadline than any job in the flexible set P_t^* of the remaining pending jobs. The property (n1) of normal schedules follows.

If TIGHTRESTART is idle then there is no pending job. Thus, to show the property (n2), it remains to verify that $P_{t'}$ is flexible at any time t' when $\mathcal{X}$ is idle but TIGHTRESTART is not. This means that TIGHTRESTART is running a job which is later preempted.

Suppose TIGHTRESTART starts a flexible job j at time s and preempts it at time t. Let t' be any time such that $s < t' < t$. Since j is flexible at s and it is the ED job in P_s, no job in P_s expires before $s + p > t$. Thus we have $P_s \subseteq P_{t'}^* \subseteq P_t^*$, by the definition of $P_{t'}^*$ and P_t^* in (TR3). As TIGHTRESTART preempts at time t, P_t^* is flexible at t. Consequently, $P_{t'}^* \subseteq P_t^*$ is flexible at t and also at $t' < t$. Using this for all t', we conclude that the first preemption candidate for j is released at t, as otherwise j would be preempted earlier. Thus no preemption preemption candidate is admissible at any t', $s < t' < t$, and $P_{t'} = P_{t'}^*$ which we have shown is flexible at t'. Thus (n2) holds and $\mathcal{X}$ is normal. □

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

Proof: As usual, let $\mathcal{Z}$ denote an optimal schedule.

Let us start by giving some intuition behind the charging scheme. Suppose that a job j is started at time t in $\mathcal{Z}$. If a job k is running at t in $\mathcal{X}$, we want to charge j to k. If $\mathcal{X}$ is running a job k which is later preempted by l, we charge $\frac{1}{2}$ to l and $\frac{1}{2}$ to k (using Lemma 2.2 to guarantee that the modified schedule completes k). The main problem is to handle the case when $\mathcal{X}$ is idle when j starts in $\mathcal{Z}$; we call such a j a *free* job. In this case, TIGHTRESTART was "tricked" into scheduling j too early. We would like to charge j to itself. However, it may happen that then j would be charged twice, so we need to split this charge and find another job that we can charge $\frac{1}{2}$. The definition of $f(j)$ below chooses such a job and Lemma 2.2 again guarantees that the modified schedule completes $f(j)$.

The difficulty that arises in the above scheme is that, due to preemptions in TIGHTRESTART's schedule and to idle times in $\mathcal{Z}$, the jobs can become misaligned. To deal with this problem, we define a matching M between the jobs in $\mathcal{X}$ and $\mathcal{Z}$. Typically, a job k in $\mathcal{X}$ is matched to the first unmatched job in $\mathcal{Z}$ that starts later than k. In some situations we match a free job k to itself.

We now proceed with the formal proof. First, we define a partial function $f : J \to J$. For any job j scheduled as flexible in $\mathcal{X}$, we define $f(j)$ as follows.

(f1) If at some time t, $S_j^{\mathcal{X}} \le t < C_j^{\mathcal{X}}$, $\mathcal{Z}$ starts a job k which is not a preemption candidate then let $f(j) = k$.

(f2) Otherwise, if there exists a job k with $S_j^{\mathcal{X}} < r_k < C_j^{\mathcal{X}} \le x_k$ such that $\mathcal{Z}$ does not complete k, then let $f(j) = k$ (choose arbitrarily if there are more such k's).

(f3) Otherwise, $f(j)$ is undefined.

Notice that $f(\cdot)$ is one-to-one, for the first two cases are disjoint, and in each case j is uniquely determined by $k = f(j)$: If $k = f(j)$ and the first case applied to j, then j is the job that is being executed by $\mathcal{X}$ when $\mathcal{Z}$ starts k. If the second case applied to j, then j is the job being executed by $\mathcal{X}$ at the release time of k.

According to Lemma 4.1, $\mathcal{X}$ is a normal schedule. Let $\mathcal{A}$ be the schedule constructed in Lemma 2.2 from $\mathcal{X}$ and function $f(\cdot)$. Since $\mathcal{A}$ is equivalent to $\mathcal{X}$, it also satisfies Lemma 2.1.

Call a job j scheduled in $\mathcal{Z}$ a *free* job if TIGHTRESTART is idle at time $S_j^{\mathcal{Z}}$. This condition implies that at time $S_j^{\mathcal{Z}}$ no job is pending in $\mathcal{A}$; in particular, by Lemma 2.1, j is completed as a flexible job by time $S_j^{\mathcal{Z}}$ in $\mathcal{A}$.

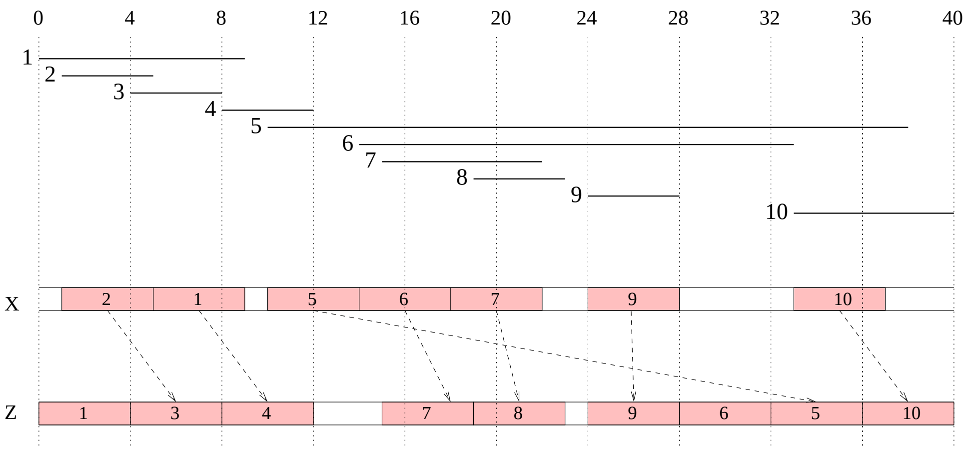

Figure 4: An example of an instance, the schedule $\mathcal{A}$ produced by TIGHTRESTART, and the construction of M (represented by directed edges). The processing time is $p = 4$. Jobs are identified by positive integers. Preempted pieces of jobs are not shown.

Now define a partial function $M : J \rightarrow J$ which is a matching of (some) occurrences of jobs in $\mathcal{A}$ to those in $\mathcal{Z}$. Process the jobs k scheduled in $\mathcal{A}$ in the order of increasing $S_k^{\mathcal{A}}$. For a given k, let j be the first unmatched job in $\mathcal{Z}$ started at or after $S_k^{\mathcal{A}}$, or, more specifically, a job with smallest $S_j^{\mathcal{Z}}$ among those with $S_j^{\mathcal{Z}} \geq S_k^{\mathcal{A}}$ and such that $j \neq M(k')$ for all k' in $\mathcal{A}$ with $S_{k'}^{\mathcal{A}} < S_k^{\mathcal{A}}$. If no such j exists, $M(k)$ is undefined. Else, if k is a free job, not in the current range of M, and $S_j^{\mathcal{Z}} \geq C_k^{\mathcal{A}}$, then let $M(k) = k$. Otherwise, let $M(k) = j$. The definition implies that M is one-to-one. (See Figure 4.)

Lemma 4.3 *Let j be a job executed in $\mathcal{Z}$.*

(1) *If $\mathcal{A}$ executes some job when j starts in $\mathcal{Z}$, that is $S_k^{\mathcal{A}} \leq S_j^{\mathcal{Z}} < C_k^{\mathcal{A}}$ for some k, then j is in the range of M.*

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

Proof: Part (1) is simple: Suppose that $\mathcal{A}$ is executing some job k at $S_j^{\mathcal{Z}}$, and consider the step in the construction of M when we are about to define $M(k)$. If j is not in the range of M at this time, then we would define $M(k)$ as j.

We now prove (2). Let $s = S_j^{\mathcal{A}}$ be the start time of j in $\mathcal{A}$ and $s' = C_j^{\mathcal{A}} = s + p$ its completion time. Since j is free, it is completed in $\mathcal{A}$ before it is started in $\mathcal{Z}$, that is $S_j^{\mathcal{Z}} \geq s'$.

Suppose for a contradiction that j is not in the range of M. By the definition of M, this implies that $M(j) = l$ for some job l with $s < S_l^{\mathcal{Z}} < s'$. Otherwise, during the construction of M when we are about to define $M(j)$, we would set $M(j) = j$.

Since $f(j)$ is undefined, by condition (f1), l must be a preemption candidate for j, that is $s < r_l \leq x_l < s'$. Furthermore, as TIGHTRESTART does not preempt j when l is released, the set $P^*_{r_l}$ is not flexible.

Figure 5 illustrates the argument that follows. The idea is this: Since j is not preempted even though a preemption candidate l arrives, $\mathcal{A}$ must be nearly full between s' and d_j. So, intuitively, one of the jobs scheduled in this interval should overlap in time with the occurrence of j in $\mathcal{Z}$, and this job would end up being matched to j. The rigorous argument gets a bit technical because of possible gaps in the schedules.

Let $K = \{j' \mid s < r_{j'} < s' \leq x_{j'}\}$ be the set of all jobs released during the execution of j in $\mathcal{A}$, excluding preemption candidates. Since $f(j)$ is undefined, by condition (f2), all these jobs are completed in $\mathcal{Z}$, and obviously they cannot be completed before $S^{\mathcal{Z}}_l$. Thus K is feasible at $C^{\mathcal{Z}}_l$ and also at $s' \leq C^{\mathcal{Z}}_l$.

Let $U = \{j' \in P_{s'} \mid r_{j'} < s'\}$ be the set of jobs pending at s' that are released strictly before s'. Since $\mathcal{A}$ is an EDF schedule and j is flexible (because j is free), all jobs $j' \in P_s - \{j\}$ have $x_{j'} \geq x_j \geq s'$, so they are still pending at s'. Therefore $U = P_s \cup K - \{j\} = P^*_{r_l} \cup K$.

We claim that U is feasible at s'. Suppose, towards contradiction, that it's not. Since K is feasible at s' and all jobs $j' \in U - K$ have $x_{j'} \geq x_j$, TIGHTRESTART would then execute urgent jobs from s' until at least the time x_j. Thus $\mathcal{A}$ would not be idle at time $S^{\mathcal{Z}}_j \leq x_j$, contradicting the assumption that j is free. We conclude that U is feasible at s', as claimed.

Now, by the feasibility of K at s' and Lemma 2.2(3) and by the flexibility of U at s and Lemma 2.2(2), $\mathcal{A}$ completes all jobs in U. Furthermore, all jobs in U are scheduled between s' and $S^{\mathcal{Z}}_j$, as TIGHTRESTART is idle at $S^{\mathcal{Z}}_j$.

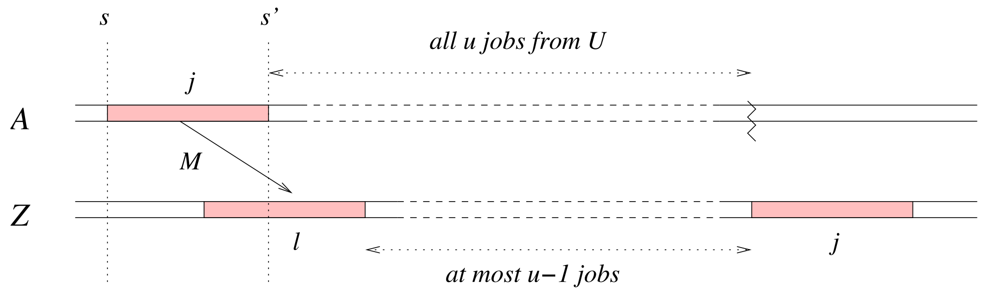

Figure 5: Illustration for the proof of Lemma 4.3(2).

Let $u = |U|$. Next we claim that

(i) $S^{\mathcal{Z}}_j - C^{\mathcal{Z}}_l < up$, and

(ii) $\mathcal{Z}$ does not schedule any of the jobs in U after j.

If either of these claims is violated, $U \cup \{j\}$ would be feasible at $C^{\mathcal{Z}}_l$, for we can first schedule K, which is feasible at $C^{\mathcal{Z}}_l$, and then the remaining jobs from U. Indeed, if (i) is violated, we can complete all jobs in U by the time $S^{\mathcal{Z}}_j$, which is smaller than the deadlines in $P^*_{r_l} - K$, and start j at $S^{\mathcal{Z}}_j$. If (ii) is violated, let j' be the job in U scheduled after j in $\mathcal{Z}$. We know that $S^{\mathcal{Z}}_j - C^{\mathcal{Z}}_l > S^{\mathcal{Z}}_j - s' - p \geq (u-1)p$, thus we can complete all jobs in U by the time $S^{\mathcal{Z}}_j$ and schedule j and j' as in $\mathcal{Z}$.

By the previous paragraph, if any of (i) or (ii) does not hold, then $P^*_{r_l} \cup \{j\} \subseteq U \cup \{j\}$ is feasible at $r_l + p \leq C^{\mathcal{Z}}_l$, contradicting the assumption that l (which is a preemption candidate) did not cause preemption. We thus obtain that (i) and (ii) are true, as claimed.

Summarizing, $\mathcal{A}$ completes the u jobs in U between s' and $S^{\mathcal{Z}}_j$, and, by (ii), these jobs are not executed after $S^{\mathcal{Z}}_j$ in $\mathcal{Z}$. Therefore, if j were not in the range of M, the jobs in U would have to be matched to the jobs in $\mathcal{Z}$ between $C^{\mathcal{Z}}_l$ and $S^{\mathcal{Z}}_j$, which is not possible, because there are at most $u - 1$ such jobs, by (i). We can thus conclude that j is indeed in the range of M. □

Charging scheme. Let j be a job started at time $t = S^{\mathcal{Z}}_j$ in $\mathcal{Z}$. We charge j to jobs in $\mathcal{A}$ according to the following cases.

Case (I): $j = M(k)$ for some k. Charge j to k. By Lemma 4.3(1), this case always applies when $\mathcal{A}$ is not idle at t, so in the remaining cases $\mathcal{A}$ is idle at t.

Case (II): Otherwise, if j is free, then charge $\frac{1}{2}$ of j to the occurrence of j in $\mathcal{A}$ and $\frac{1}{2}$ of j to the occurrence of $f(j)$ in $\mathcal{A}$. Note that, since (I) does not apply, Lemma 4.3(2) implies that $f(j)$ is defined, and then Lemma 2.2 implies that both j and $f(j)$ are completed in $\mathcal{A}$.

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

Analysis. We prove that each job scheduled in $\mathcal{A}$ 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 each job receives at most one charge of $\frac{1}{2}$. For the rest of the proof, we will distinguish two types of charges of $\frac{1}{2}$: *self-charges*, when j is charged to itself, and *non-self-charges*, when j is charged to a different job.

Suppose first that k receives a self-charge. (Obviously, it can receive only one.) By the charging scheme, $\mathcal{A}$ is idle at time $S^{\mathcal{Z}}_k$. This implies two things. First, k is not tight, so it cannot receive a non-self-charge in Case (III). Second, k cannot be in the range of $f(\cdot)$, since each job $f(j)$ is either not in $\mathcal{Z}$ or, if it is, $\mathcal{A}$ is executing some job at time $S^{\mathcal{Z}}_{f(j)}$. Therefore k cannot receive a non-self-charge in Case (II).

Next, suppose that k does not receive a self-charge. Since $f(\cdot)$ is one-to-one, k can receive at most one non-self-charge in Case (II). If k receives a non-self-charge in Case (III) from a job j, then k is started in $\mathcal{A}$ while $\mathcal{Z}$ is executing j, so k can receive only one such charge. Finally, if k receives a non-self-charge in Case (II) then, by the definition of $f(\cdot)$, k is not a preemption candidate, so it cannot receive a non-self-charge in Case (III).

We conclude that each job completed in $\mathcal{A}$ gets at most one charge of 1 and at most one charge of $\frac{1}{2}$, and thus is charged a total of at most $\frac{3}{2}$. Each job in $\mathcal{Z}$ generates a charge of 1. Thus, by summation over all jobs in $\mathcal{Z}$, we have $|\mathcal{Z}| \leq \frac{3}{2}|\mathcal{A}|$, completing the proof of the theorem. □

We now show that the competitive ratio of our algorithm is in fact optimal.

Theorem 4.4 *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 $j = (r_j, d_j)$: $1 = (0, 3p+1)$, $2 = (1, 3p)$, $3 = (p, 2p)$, $4 = (p+1, 2p+1)$. The instance consists of jobs $1, 2, 3$ or $1, 2, 4$.

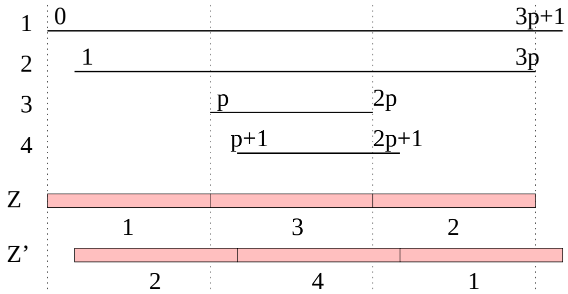

Figure 6: Jobs used in the lower bounds with restarts.

There exist schedules that schedule three jobs $1, 3, 2$ or three jobs $2, 4, 1$, in this order. (See Fig. 6.) Therefore the optimal solution consists of three jobs.

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 two jobs. As the optimal schedule has three jobs, the competitive ratio is no better than $\frac{3}{2}$.

Our proof for randomized algorithms is based on Yao's principle [21, 6]. We define a probability distribution on our two instances, as follows: Always release jobs 1 and 2, and then a randomly chosen one job from 3 and 4, each with probability $\frac{1}{2}$. If $\mathcal{A}$ is any deterministic online algorithm, then the expected number of jobs completed by $\mathcal{A}$ is at most 2.5, as on one of the instances it completes only 2 jobs. Using Yao's principle, we conclude that no randomized algorithm can have competitive ratio smaller than $3/2.5 = \frac{6}{5}$. $\square$

5 Final Comments

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 only 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. The best upper bound of $\frac{3}{2}$ is achieved by a deterministic algorithm. Can randomization help in the model with restarts?

References

[1] S. Albers. On randomized online scheduling. In *Proc. 34th Symp. Theory of Computing (STOC)*, pages 134–143. ACM, 2002.

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

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

[4] S. K. Baruah, J. Haritsa, and N. Sharma. On-line scheduling to maximize task completions. In *Proc. Real-Time Systems Symp.*, pages 228–236, 1994.

[5] S. K. Baruah, J. Haritsa, and N. Sharma. On-line scheduling to maximize task completions. *J. Comb. Math. Comb. Comput.*, 39:65–78, 2001.

[6] A. Borodin and R. El-Yaniv. *Online Computation and Competitive Analysis*. Cambridge University Press, 1998.

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

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

[9] M. Chrobak, W. Jawor, J. Sgall, and T. Tichý. Online scheduling of equal-length jobs: Randomization and restarts help. In *Proc. 31st International Colloquium on Automata, Languages, and Programming (ICALP)*, volume 3142 of *Lecture Notes in Comput. Sci.*, pages 358–370. Springer, 2004.

[10] L. Epstein, J. Noga, S. S. Seiden, J. Sgall, and G. J. Woeginger. Randomized on-line scheduling for two related machines. *J. Sched.*, 4:71–92, 2001.

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

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

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

[14] M. H. Goldwasser and B. Kerbikov. Admission control with immediate notification. *J. Sched.*, 6:269–285, 2003.

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

[16] J. Jackson. Scheduling a production line to minimize maximum tardiness. Technical Report 43, Management Science Research Project, Univ. of California, Los Angeles, U.S.A., 1955.

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

[18] N. Reingold, J. Westbrook, and D. D. Sleator. Randomized competitive algorithms for the list update problem. *Algorithmica*, 11:15–32, 1994.

[19] S. Seiden. Barely random algorithms for multiprocessor scheduling. *J. Sched.*, 6:309–334, 2003.

[20] B. Simons. A fast algorithm for single processor scheduling. In *Proc. 19th Symp. Foundations of Computer Science (FOCS)*, pages 246–252. IEEE, 1978.

[21] A. C. C. Yao. Probabilistic computations: Towards a unified measure of complexity. In *Proc. 18th Symp. Foundations of Computer Science (FOCS)*, pages 222–227. IEEE, 1977.