Making Commitments in the Face of Uncertainty: How to Pick a Winner Almost Every Time

(Extended Abstract)

Baruch Awerbuch[*] *Yossi Azar*[†] *Amos Fiat*[‡] *Tom Leighton*[§]

Abstract

In this paper, we formulate and provide optimal solutions for a broad class of problems in which a decision-maker is required to select from among numerous competing options. The goal of the decision-maker is to select the option that will have the best future performance. This task is made difficult by the constraint that the decision-maker has no way to predict the future performance of any of the options. Somewhat surprisingly, we find that the decision-maker can still (at least in several important scenarios) pick a winner with high probability.

Our result has several applications. For example, consider the problem of scheduling background jobs on a network of workstations (NOW) when very little is known about the future speed or availability of each workstation. In this problem, the goal is to schedule each job on a workstation which will have enough idle capacity to complete the job within a reasonable or specified amount of time. This task is complicated by the fact that any particular workstation might become saturated by higher priority jobs shortly after one of our jobs is assigned to it, in which case progress will not be made on our job. Thus, in order to complete the jobs within a specified amount of time, we need to be able to accurately guess (or predict) which workstations will be idle and when. Somewhat surprisingly, it is possible to make such guesses with a very high degree of accuracy, even though very little is assumed about the future availability of the workstations. For example, if at least k of n workstations will be available for at least D units of time each (spread out over some interval of I units of time), then with probability at least $1-O(1/n)$, we will be able to complete $k \log n$ jobs with duration $\Omega(D/\log n)$ within the interval. The result holds for all k, d, n, and I, and only knowledge of n is needed in order to schedule the tasks. For small values of k, the result is far superior to the (seemingly optimal) "dart-throwing" approach in which each job is assigned to a random workstation in the hope that it will be idle.

Our results can also be used to provide the first competitive algorithm for the video-on-demand scheduling problem as well as the more general on-line set cover problem. The results may also be of interest in the context of investment planning, strategic planning, and other areas where it is important to be able to predict the future moves of an adversary or a market.

1 Introduction

In this paper, we consider a class of optimization problems in which there is a decision-maker who is required to choose from among numerous competing options. The goal of the decision-maker is to select the option that will have the best future performance. The task is complicated by the fact that the decision-maker only has information about the *past* performance of each option, and little or no information about future performance can be assumed or inferred based on the past performances. Nevertheless, we will show how to make optimal or near-optimal selections in a variety of settings.

Decision-making problems arise in numerous applications. For concreteness, we will focus most of our attention on one such application in this paper. In particular, we will focus on the problem of scheduling background or low-priority jobs on a network of workstations (NOW) when very little information is known about the future speed or availability of each of the workstations. We will then mention how the results can be generalized

[*]Johns Hopkins University and Lab. for Computer Science, MIT Supported by Air Force Contract TNDGAFOSR-86-0078, ARO contract DAAL03-86-K-0171, NSF contract 9114440-CCR, DARPA contract N00014-J-92-1799, and a special grant from IBM. E-Mail: baruch@theory.lcs.mit.edu.

[†]Department of Computer Science, Tel Aviv University. E-Mail azar@math.tau.ac.il. Research supported in part by Allon Fellowship and by the Israel Science Foundation administered by the Israel Academy of Sciences

[‡]Department of Computer Science, Tel-Aviv University, Israel E-Mail· fiat@math.tau ac il. Research supported in part by the Israel Science Foundation administered by the Israel Academy of Sciences.

[§]Mathematics Department and Lab for Computer Science, MIT E-Mail: ftl@math.mit.edu Research supported in part by ARPA Contract N00014-95-1-1246 and Army Contract DAAH04-95-1-0607.

and applied to other problems.

1.1 Scheduling Jobs on Networks of Workstations

Consider a company in which there is a network of n workstations and where there is a hierarchy of priorities for the various users of each workstation. For example, the owner of a workstation might have top priority for his/her workstation, the president of the company might have second priority for all workstations, and so forth. Sometimes, an employee (call him Bob) might have more work than can be handled by his own workstation and so he will schedule some of his jobs on other workstations in a background or low-priority mode. When those workstations are not busy working on higher-priority tasks (say those of their owners), they will work on Bob's background job.

Such job scheduling on NOWs has become very common. In fact, the "unused" capacity on NOWs has become increasingly viewed as a major computational resource within companies. Accordingly, it is often desired that this resource be used (and used efficiently) before funds are allocated to purchase additional equipment.

In this paper, we describe efficient algorithms for scheduling background jobs on NOWs. To begin, we focus on the problem faced by a single user who needs to schedule a set of jobs (perhaps with precedence constraints) on the workstations so that all of the jobs can be accomplished within some interval of time. In order to be efficient, the user is restricted to assign each job to a single workstation.[1] (E.g., the user is not allowed to make n copies of a job and then simultaneously assign one copy to each workstation.) If the user is unsatisfied with the progress of a workstation on a job, he may elect to kill the job and start over on another workstation, however. Of course, the user will be reluctant to make frequent switches because of the effort lost when the job is killed and the overhead incurred when the job is moved elsewhere.

The goal of the user is to assign the jobs to workstations that will be able to complete them quickly. The difficulty is that the user does not know ahead of time which workstations will be able to finish a job quickly. The user can monitor the NOW to see which workstations are currently available, and which were available at various times in the past, but we do not assume that there is any connection (probabilistic or otherwise) between the past or present and the future. (E.g., the fact that the workstation was available for the last T seconds does not imply anything about the prospect that it will be available in the next second.)[2]

For simplicity, we will partition time into units of equal length called *steps,* and we will assume that every workstation is either available or unavailable for each step. We will also assume that all workstations are equally fast and that there is no penalty incurred when a workstation swaps back and forth between a low-priority job and a higher-priority job. Hence, a workstation $\mathcal{W}$ will complete a d-step background job in the dth step that $\mathcal{W}$ was available after the job was assigned to $\mathcal{W}$.[3] We also assume that an adversary decides which workstations will be available at each step and that this adversary is aware of our scheduling algorithm, but not the results of our random coin tosses.

We first consider the scenario when Bob wants to schedule a single background job of duration d on the NOW. Bob may schedule the job on any machine that he chooses and he is assumed to have full knowledge of the past availability of every machine. After scheduling the job, Bob may later kill the job and restart elsewhere if he is not satisfied with the performance of his first choice. The number of times that Bob must restart his job is something that we would like to minimize (in addition to the total time needed to get the job done).

In order for Bob to get his job done in some interval of I steps, it is necessary that at least one workstation be available for d steps within the interval. Unfortunately, this condition is not sufficient to guarantee that Bob can get his job done since Bob does not know until it is too late which workstation(s) will be available for d steps. Even if many (say m) of the n workstations will be available for d steps, the probability that Bob will be able to pick one of the m good workstations can be limited to m/n by the adversary. (In this case, the adversary need only make $n-m$ workstations available for fewer than d steps and m workstations available for exactly d steps. The m good workstations can be chosen randomly. Bob's best strategy is to schedule the job on a random workstation at the beginning of the interval and then to hope for the best. It will not help Bob to make any changes in this case.)

The preceding example is very discouraging since it demonstrates that Bob will not be likely to get his job done even though there may be a great deal of unused capacity in the NOW. Even worse, the adversary need not be malicious since even a random adversary is sufficient to thwart Bob.

In the paper, we show that if the adversary is restricted slightly, however, then Bob can find a good workstation with very high probability. In particular, if at least one workstation will be available for $\alpha d \log n$ steps (where $\alpha = \Theta(1)$), then Bob will be able to get his d-step job done with probability $1-O(1/n)$. In fact, even if Bob has $\log n$ d-step jobs which must be done in sequence, he will still be able to get them all done with probability $1-O(1/n)$. Moreover, in the latter case, the number of times that Bob will need to kill a job and switch workstations will be very small compared to the number of times that Bob picks a good workstation for one of his jobs. In other words, Bob will almost always be able to pick a "good" workstation when he

[1] Such a constraint may also be needed to insure coherency in cases where the job makes use of external data or where the job could have side effects.

[2] In some scenarios, it might be reasonable to assume that such a dependency *does* exist (i.e., that workstations that were idle in the past are more likely to be idle in the future), but we will show that such assumptions are not necessary in order to obtain good performance. Hence, we will do without them, thereby obtaining algorithms with stronger guarantees of performance.

[3] Most of these assumptions can be relaxed without changing the results that we prove, although the analysis becomes more difficult. For example, our methods can also be applied to scenarios where machines run at different speeds and where there is a penalty for swapping between jobs. In fact, the methods even work for a model where job swapping is not allowed (e.g., if a job is swapped out, it is killed).

goes to schedule a job. Viewed in another way, our result shows that if Bob partitions his work into jobs of size $O(D/\log n)$, where D is the amount of time that the most idle workstation will be available, then he will almost certainly be able to get $\Theta(D)$ work done. Moreover, if Bob is allowed to schedule $O(1)$ jobs at once, then he will be able to get $(1-\epsilon)D$ work done for any constant $\epsilon > 0$ with high probability.[4]

The results can be extended to a scenario where there are multiple users with varying priorities. In this case, if the ith most available machine is available for D_i steps, then for all i, the user with the ith priority can be assured (with high probability) of getting at least $(1-\epsilon)D_i$ work done. Knowledge of the D_i's is not needed in order to obtain these results and no coordination is needed among the users.

Our scheduling algorithm is quite simple and tends to schedule jobs on machines that have been available in the past (thus confirming that what may be considered a standard practice is, in fact, good practice). Although the past is not related to the future, we will show that such a strategy makes it difficult for the adversary to prevent any of the users from making progress while also making sure that i workstations are available for at least D_i steps. (This is provided, of course, that the strategy is randomized and that the random choices are made in the right way.)

1.2 A General Problem Formulation

The problem just described can be formulated in a more general context. For example, consider a problem in which there are n commodities. A decision-maker or trader (call her Alice) is allowed to "hold" at most one commodity at any time. At each step, each commodity may or may not issue a "dividend" or a return of, say, \$1. Allice collects the dividend as profit if and only if she is holding the commodity at the time that the dividend is paid. Alice may or may not be allowed to change the commodity that she is holding and there may or may not be a steep penalty or cost every time that Alice makes a change. Alice's goal, of course, is to maximize her profit while minimizing the number of changes that she makes.[5]

Let D denote the number of dividends paid by the best commodity. At first glance, it seems as though Alice's best strategy is to select a random commodity, in which case her expected profit might be as small as $\Theta(D/n)$. In fact, Alice can do much better.

For example, even if Alice is not allowed to make any trades (i.e., she is allowed to hold only one commodity) and $D \geq \log n$, then our selection algorithm will obtain a profit of $\Omega(D/\log n)$ for Alice with probability at least 0.99. (This is optimal.) By making $O(\log n)$ trades, Alice's profit can be increased to $\Theta(D)$ with probability $1-O(1/n)$. (This is also optimal.) If Alice is allowed to hold $O(1)$ commodities at any step, then her profit can be increased to $.99D$ with high probability. The value of D need not be known in advance, although then the number of trades may increase to $O(\log n \log(D/\log n))$ if D is not known.[6] The results can also be extended to a scenario where there are multiple traders or to where Alice is allowed to own several commodities at the same time. In each case, we will show how Alice can almost always pick a winner every time.

1.3 On-line Set Cover and Related Applications

It turns out that our methods can also be used to give a $\Theta(\log n \log \frac{m}{k})$ competitive algorithm for the on-line set cover problem.[7][8] In the on-line set cover problem, we are given a family of n sets $F = \{S_1, S_2, \ldots, S_n\}$. Elements $v_1, v_2, \ldots, v_m$ arrive one at a time where each v_i belongs to at least one of the sets. As each element arrives, the sets to which it belongs are revealed to the player. The goal of the player is to pick k sets so as to maximize the number of elements that are covered. In this version of the problem, the player can make selections at any time, but cannot change a selection once it is made. In addition, the player only gets credit for elements that are contained in a set that was selected by the player *before* or *during* the step when the element arrived. Moreover, the player only gets credit for each element at most once, even if he/she has selected many sets that contain the element.

In the case when the sets are constrained to be disjoint, we have the special case of the commodity trading problem described in the previous section where Alice is allowed to own k commodities but cannot make any trades. Alice's profit is simply the number of elements covered.[9]

[4]Although the constraint that some workstation be available for $D = \alpha d \log n$ steps may seem strange at first, it is really a mild assumption. For example, such a condition would be likely to arise if workstations were available with some constant probability at each step. Such a condition may also be likely to arise in practice since some workstations may be idle for long periods of time. By focusing on the D-step availability constraint, we are merely identifying a salient feature that makes the problem tractable. The fact that we have identified a salient feature is demonstrated by the matching lower bounds that are proved in the paper Finally, the constraint is useful in guiding the choice of job sizes. In particular, we find that by partitioning maximal-length jobs into $\Theta(\log n)$ chunks, we will be assured of scheduling them efficiently, even without knowledge of which workstations will be available

[5]The connection between the trading problem and job scheduling on NOWs is quite close The workstations correspond to commodities and dividends correspond to workstation availability. The only difference is that a job is completed on a workstation only when a threshold is reached in terms of availability. Such a threshold can be modeled by a steep trading cost. Our methods can also be applied to a scenario in which a job is completed iff a workstation is available for d *consecutive* steps, in which case the dividends can be collected iff they are issued for d consecutive steps during which Alice holds that commodity

[6]If D is not known and Alice is not allowed to make trades, then her probability of obtaining a profit of $\Omega(D/\log n)$ is decreased

[7]This bound is optimal for many values of n, m, and k.

[8]We use the convention that the competitive ratio is always greater than one, even though we consider benefit problems rather than cost problems. Thus, an upper bound of c on the competitive ratio really means that the ratio between the on-line benefit and the adversary benefit is at least $1/c$ A lower bound of c on the competitive ratio means that the above ratio is at most $1/c$.

[9]Technically, Alice only gets credit for dividends paid on commodities that she owned in *previous* steps, in which case, we need to assume that the return of the off-line player is $\Omega(k \log n)$ in order to guarantee that Alice will be $O(\log n \log \frac{m}{k})$ competitive

1.3.1 Video-on-Demand Scheduling

The disjoint version of the set cover problem is also equivalent to the "video-on-demand" scheduling problem posed in [AGH94]. In this problem, customers issue requests for movies (which will start at a prearranged later time) to a video server with limited capacity k. Each customer demand is immediately accepted or rejected. If the demand is accepted, that movie must be shown. If the demand is rejected, the customer is lost. The goal is to be able to accept the largest number of demands subject to the capacity constraint k.

The on-line algorithms in [AGH94] achieve linear ($O(m)$) competitive ratio; the semi-offline algorithms with look-ahead (where the decision can be postponed to the time when future demand is known) achieve logarithmic ratio. In this case, the movies correspond to the sets, and the customers correspond to the base elements. The more general (non-disjoint) set cover problem corresponds to a setting in which every customer discloses a list of alternative movie titles that he/she wants to watch.

1.3.2 Investment Planning

The disjoint set cover problem also has applications in the context of investment planning. For example, previous work on the competitive analysis of financial problems [EFKT92, EK93, CEL93] focused on trading problems, where algorithms were allowed to make partial investments, to retract from previous decisions (at some cost), or were based on some statistical knowledge of the input. In contrast, on-line set cover captures situations where investment decisions are indivisible and irrevocable: once an investor has decided to invest in building a new factory, he/she must hope that there will be high demand for the product produced by the factory, in which case the dividends paid to investors will be high. In our model, investment decisions do not have to take place immediately; investors can postpone the decision until he/she get convinced that the company is doing well and indeed paying lofty dividends. Still there is risk involved since the future and the past are not necessarily related and once the investor decides to invest in a company, demand for its product could drop to zero, in which case the investor is stuck with the factory building and a warehouse full of merchandise that nobody wants. Hence, with limited financial resources, the investor has to decide what investments to make and at what stage along the curve of increasing demand.

The more general (non-disjoint) set cover problem captures scenarios where there may be complex relationships between between one company's growth and another's (e.g., they are producing a competitive product or mutually exclusive lines of products). For example, the marginal revenues of investing in a factory producing answering machines may be too low *after* purchase of a factory producing answering machines with a cordless phone. To analyze such problems, we view companies as sets of products, and arriving customers as base elements. Upon their arrival, customers disclose mutually exclusive lists of products they would like to purchase. If, by that time, the investor has already has purchased a company producing any one of these products, the investor makes a profit. Otherwise, the customer (and the money) are lost, since the client may not wait.

1.3.3 Strategic Planning

The methods developed in this paper may also be of interest in the domain of strategic planning and war gaming. For example, consider the following oversimplified battlefield scenario where a general needs to decide where and when to attack an enemy defender. In the scenario, the general wants to find a single soft spot from among n enemy defensive positions. The enemy does not have the resources to defend all n positions, but he can shift his forces on a daily basis. The attacking general has access to intelligence data that reveals which enemy positions were well defended in all previous days, but this data does not provide the general with any information about which positions will be defended during the coming day(s).

The general's problem is to pick a position $p \in [1, n]$ and a day $q \in [1, T]$ for the attack. The general wins if site p is not well defended on day q, and he loses otherwise. The only knowledge possessed by the general about the future is that over the course of the next T days, at least one of the n positions will not be defended for at least $\Theta(\log nT)$ consecutive days.[10] In this case, our results provide a strategy for the general that will result in victory with probability near one, no matter what strategy is used by the enemy. The high success probability holds even if the enemy knows the strategy being followed by the general (but not the result of the coin tosses being used by the general each morning when deciding whether or not to attack). In other words, it is not possible for the enemy to trick the general into attacking a well-defended position by, for example, leaving a position undefended for several consecutive days, only to switch and suddenly ramp up defenses in anticipation of an attack.

Our results can also be applied to scenarios where multiple attacks are being planned and/or where attacks can be broken off quickly if a position is attacked that is well defended. In such cases, the general can achieve success against even less restricted enemies. The general's decision about where and when to attack does rely on random coin tosses each morning (which might be disconcerting to some) but it is the randomness in the coin tosses that insures that the general will be successful with high probability. (For instance, without the randomness, the general will have no chance of success against an enemy who has acquired the general's strategy.)

1.4 Additional Previous Work

Various on-line optimization variants of many combinatorial problems have been subject to competitive anal-

[10]In some cases, the restriction that a position be left undefended for $\Theta(\log nT)$ consecutive days is unrealistic. Indeed, the enemy may decide to continually move his forces so that no site is left undefended for more than $o(\log nT)$ days at a time However, continual movement of forces might be costly, and eventually, the enemy may be forced to leave some sights less well defended for longer periods of time. If this ever happens, the general will win with high probability.

ysis [ST85a, KMRS88]. Such problems include, for example, *on-line matching* [KVV90], *partition* [FKT89], *on-line steiner tree* and generalizations [IW91, CV92, ABF93, WY93], and also *on-line graph coloring* [Vish90, Irani90, HS92].

The subject of job scheduling in the face of uncertainty has also been studied previously [BL94, KP94, BCLR95], though our results differ from those of prior researchers in several respects. The closest prior work is that of Kalyanasundaram and Pruhs [KP94], who devise competitive algorithms for task scheduling of unrelated jobs on a NOW where processors may become faulty. In the case when most processors are faulty and replication is not allowed, the results are mostly negative. In contrast, the positive nature of the results in this paper are possible because we consider a different adversary and objective function. Bhatt et al. [BCLR95], on the other hand, study a related cycle-stealing problem in which the goal is to maximize the amount of *uninterrupted* work that can be stolen from a *single* workstation.

1.5 Outline of the Paper

The remainder of the paper is divided into sections as follows. In Section 2, we describe an optimal algorithm for picking a winner with probability close to 1 on the first try. We then show how this algorithm can be used to provide an optimal solution to the on-line set cover problem. In Section 3, we describe a more efficient algorithm for picking winners repeatedly if we are allowed to kill and reschedule jobs. Several extensions of the results are presented in Section 4. We conclude with some acknowledgements and references.

2 Picking a Winner on the First Try

We begin by considering the scenario where Bob wants to run a single d-step job and he is only allowed to schedule the job once. (This corresponds to the scenario where Alice is allowed to own only one commodity and to the on-line set cover problem where we can select just one set.) We will assume that at least one of the n workstations will be available for at least $D \geq 3d\log n$ steps. In what follows, we will show how to select a workstation so that the job is completed with probability at least $1 - O(\frac{d\log n}{D})$.

The algorithm for selecting the workstation is quite simple. Label the workstations as $\mathcal{W}_1, \mathcal{W}_2, \ldots, \mathcal{W}_n$. At each step, Bob checks the status of each workstation to see which are available. For each i ($i = 1, 2, \ldots, n$ in sequence), if $\mathcal{W}_i$ was available at the end of the last step, Bob flips a coin to decide whether or not to assign the job to $\mathcal{W}_i$. In particular, Bob will assign the job to $\mathcal{W}_i$ with probability $n^{3x/D-2}/d$, where x is the number of steps for which $\mathcal{W}_i$ has been available thus far. If the job is ever assigned to some workstation, then Bob stops flipping coins and just waits for the job to be completed. In what follows, we will show that with probability at least $1 - O((d\log n)/D + 1/n)$, the job will be assigned to a workstation that will be available for d or more steps following the assignment, no matter what strategy is employed by the adversary in determining when machines are available.

Let $\mathcal{S}$ denote a sample space of coin tosses where there will be one coin toss for each pair (i, j) for which $\mathcal{W}_i$ is available at step j. If $\mathcal{W}_i$ is available for the xth time at step j, then the probability of Heads for the (i, j) flip is set to be $n^{3x/D-2}/d$.

There is a sample point in $\mathcal{S}$ for each possible combination of Heads and Tails for the flips. Each sample point also corresponds to an outcome of the coins flipped by Bob to determine when and where to schedule his job. In particular, Bob will schedule his job on $\mathcal{W}_i$ at step j if the (i, j) coin is Heads and if the (i', j') coin is Tails for all (i', j') such that $j' < j$ or $j' = j$ and $i' < i$.

Let $\mathcal{S}_{\text{win}}$ denote the subspace of $\mathcal{S}$ consisting of sample points for which Bob is successful in getting his job done. These are the sample points for which there is at least one Head among the flips and for which the first flip to result in Heads occurs for a workstation that is available for at least d subsequent steps (i.e., for which there are at least d subsequent flips). The probability that Bob is successful is then $Pr[\mathcal{S}_{\text{win}}]$.

In order to show that $Pr[\mathcal{S}_{\text{win}}]$ is close to 1, we will consider a second space $\mathcal{S}' \subseteq \mathcal{S}$ for which it is easy to show that $Pr[\mathcal{S}']$ is large and that $Pr[\mathcal{S}_{\text{win}}]$ is nearly as large as $Pr[\mathcal{S}']$. In particular, we define $\mathcal{S}'$ to consist of the sample points for which there is at least one Head and for which the first d flips for each workstation resulted in Tails.

Lemma 2.1 $Pr[\mathcal{S}'] \geq 1 - O(1/n)$.

Proof: The probability of getting a Head among the (at most) dn flips for which $x \leq d$ is at most

$$dn(n^{3d/D-2}/d) \leq 2/n.$$

The probability that there are no Heads among the last d flips for the workstation that is available for D steps is either 0 or at most

$$(1 - \frac{n^{3(D-d)/D-2}}{d})^d \leq (1 - \frac{n}{2d})^d \leq e^{-n/2}. \quad \Box$$

Lemma 2.2 $Pr[\mathcal{S}_{win}] \geq (1 - O(\frac{d\log n}{D}))Pr[\mathcal{S}']$.

Proof: We will construct an injection $f : \mathcal{S}' \to \mathcal{S}_{\text{win}}$ for which

$$\forall s' \in \mathcal{S}' \;\; Pr[f(s')] \geq (1 - O(\frac{d\log n}{D}))Pr[s'].$$

The lemma will then follow.

Consider any sample point $s' \in \mathcal{S}'$. Let $\mathcal{W}_i$ be the workstation for which a flip is first Heads in s' and let x denote the number of steps that $\mathcal{W}_i$ had been available up to and including the step when the first Heads occurred. By the definition of $\mathcal{S}'$, we know that $x > d$. Let

$$z' = n^{3x/D-2}/d$$

and

$$\begin{aligned} z &= n^{3(x-d)/D-2}/d \\ &= n^{-3d/D}z' \\ &= (1-\Theta(\frac{d\log n}{D}))z'. \end{aligned}$$

If $z' \geq 1/2$, then define $f(s')$ to be the sample point which is identical to s' in every way except that the outcome of the $(x-d)$th flip for $\mathcal{W}_i$ is changed from Tails to Heads. By definition, this sample point is in $\mathcal{S}_{\text{win}}$. Moreover,

$$\begin{aligned} \frac{Pr[f(s')]}{Pr[s']} &= \frac{z}{1-z} \\ &= \frac{(1-\Theta(\frac{d\log n}{D}))z'}{1-(1-\Theta(\frac{d\log n}{D}))z'} \\ &\geq \frac{\frac{1}{2}-\Theta(\frac{d\log n}{D})}{\frac{1}{2}+\Theta(\frac{d\log n}{D})} \\ &= 1-\Theta(\frac{d\log n}{D}), \end{aligned}$$

since $z' \geq 1/2$.

If $z' \leq 1/2$, then define $f(s')$ to be the sample point which is identical to s' except that the outcome of the $(x-d)$th flip for $\mathcal{W}_i$ is changed from Tails to Heads *and* the outcome of the xth flip for $\mathcal{W}_i$ is changed from Heads to Tails. Once again, it is easy to see that $f(s') \in \mathcal{S}_{\text{win}}$. It is also easy to check that f is an injection. Moreover,

$$\begin{aligned} \frac{Pr[f(s')]}{Pr[s']} &= \frac{z}{1-z}\cdot\frac{1-z'}{z'} \\ &\geq (1-\Theta(\frac{d\log n}{D})) \\ &\quad\times\frac{1-z'}{1-z'+\Theta(\frac{d\log n}{D}z')} \\ &\geq 1-\Theta(\frac{d\log n}{D}), \end{aligned}$$

since $z' \leq 1/2$. □

Theorem 2.3 $Pr[\mathcal{S}_{win}] \geq 1 - O(\frac{d\log n}{D} + \frac{1}{n})$.

Proof: Follows immediately from Lemmas 2.1 and 2.2. □

Remark 2.4 By being more careful with the asymptotic analysis and adjusting the probability of the flips slightly, it is possible to make $Pr[\mathcal{S}_{\text{win}}] \geq 1-\frac{3d\log n}{D}-\frac{2}{n}$.

Remark 2.5 The probability bound of Theorem 2.3 cannot be improved by more than a constant factor, no matter what algorithm is used to select a workstation. This is because the adversary can select $\rho^j n$ of the workstations at random to be available for the first dj steps for $1 \leq j \leq D/d$ where $\rho = 1-\Theta(\frac{d\log n}{D})$. (In other words, $(\rho^j - \rho^{j+1})n$ machines will cease to be available immediately after step dj.) No matter what selection algorithm is used, it can have at most a ρ chance of picking a winner. More generally, a similar adversary can be used to show that if $D \in [td, (t+1)d-1]$, then Bob can select a good workstation with probability at most $n^{-1/t}$.

2.1 Application to On-line Set Cover

Next, we show how to adapt the preceding algorithm to provide an optimal $O(\log n \log \frac{m}{k})$-competitive algorithm for the on-line set cover problem.

In the set cover problem, we are given n sets $F = \{S_1, S_2, \ldots, S_n\}$, of which we are allowed to choose k. The elements $v_1, v_2, \ldots, v_m$ arrive one per step (without loss of generality), and as each element arrives, we learn what sets it belongs to. In what follows, we will assume that credit is given for an element if we have chosen a set containing the element in the past or if we choose a set containing the element during the step when the element arrives.[11] (E.g., in the video-on-demand problem, we can get credit for a request for a movie *after* seeing the request, but only if we immediately accept the request.)

For clarity of exposition, we will think of the k choices as being made by k different people $\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_k$, each of whom will make at most one choice. When a person $\mathcal{P}_l$ chooses some set S, he will get *credit* for all elements $v \in S$ that arrived during or after the step when he selected S *and* that are not credited to $\mathcal{P}_{l'}$ for $l' < l$. (In other words, if v is contained in a set that was selected by $\mathcal{P}_{l'}$ for some $l' < l$ before or during the step when v arrived, then v is not credited to $\mathcal{P}_l$.) By prioritizing the allocation of credit in this way, we can ensure that we only get credit for each element once, even if the same set is selected more than once. The prioritization also allows the analysis to proceed as if $\mathcal{P}_1$ selects first (seeing all the elements before $\mathcal{P}_2$ selects), $\mathcal{P}_2$ selects from what remains, and so forth, even though all the players make their decisions as each element arrives. (So, in fact, $\mathcal{P}_1$ might select the same set as $\mathcal{P}_2$ after it is selected by $\mathcal{P}_2$. In this case, $\mathcal{P}_2$ stops getting credit for the set as soon as $\mathcal{P}_1$ selects the set.)

$\mathcal{P}_l$ makes his selection using the protocol followed by Bob in the previous section, but with the following modifications. First the value of D is replaced by $D_l = 2^{j-1}$ where j is selected at random from $[1, \log \frac{m}{k}]$. (Each D_l is chosen independently from the others.) The value of d is replaced by $d_l = \lfloor \frac{D_l}{\alpha \log n} \rfloor$, where α is a suitably large constant. Next, we identify set S_i with workstation $\mathcal{W}_i$ for each i. $\mathcal{W}_i$ will be considered to be "available" by $\mathcal{P}_l$ at the jth step iff $v_j \in S_i$ and $v_j \notin S_{i'}$ for all i' such that $S_{i'}$ was previously selected by $\mathcal{P}_{l'}$ for some $l' < l$. (In other words, S_i is considered to be available at step j by $\mathcal{P}_l$ if $\mathcal{P}_l$ would have gotten credit for v_j had $\mathcal{P}_l$ chosen S_i in the past or if $\mathcal{P}_l$ chooses S_i now.) If $d_l = 0$ and this is the first time that $\mathcal{P}_l$ has a chance to get credit for an element, then $\mathcal{P}_l$ selects S_i. Otherwise, $\mathcal{P}_l$ selects S_i at step j with the same probability that Bob would select $\mathcal{W}_i$ at step j.

In what follows, we use R_l to denote the random variable that counts the number of covered elements that are credited to $\mathcal{P}_l$. We also define $R = R_1 + R_2 + \cdots + R_k$ to be the size of the cover produced by the algorithm. Our goal is to show that $Ex[R] \geq$

[11]The results can be modified to handle a scenario where credit is only given for elements that are contained in *previously-held* sets, but we will then need to assume that the optimal k-cover contains $L \geq \alpha k \log n$ elements where α is a sufficiently large constant, although we will not assume that the value of L is known in advance.

$\Omega(\frac{L}{\log n \log(m/k)})$, where L is the size of the optimal off-line k-cover.

Let L_l be the random variable that denotes the maximum amount that any workstation (or set) appeared to be available to $\mathcal{P}_l$. If $L_l \geq L/2k$, then D_l will be chosen so that $\frac{L}{4k} \leq D_l \leq L_l$ with probability at least $1/\log\frac{m}{k}$. If this happens (i.e., if $\frac{L}{4k} \leq D_l \leq L_l$) and $d_l \geq 1$, then we can use Theorem 2.3 to show that $R_l \geq d_l \geq \Omega(\frac{D_l}{\log n}) \geq \Omega(\frac{L}{k \log n})$ with probability close to 1. If $\frac{L}{4k} \leq D_l \leq L_l$ and $d_l = 0$, then $R_l = 1 \geq \Omega(\frac{D_l}{\log n}) \geq \Omega(\frac{L}{k\log n})$ with certainty. Combining the preceding facts yields the conclusion that $R_l \geq \Omega(\frac{L}{k\log n})$ with probability at least $\Omega(1/\log\frac{m}{k})$ (all provided $L_l \geq L/2k$).

We next show that if $L_l < L/2k$ for some l, then $R \geq L/2$. The proof is by contradiction. Suppose that $R < L/2$. Then $R_1 + R_2 + \cdots + R_{l-1} < L/2$. Since there are k sets that cover L items, this means that some set must cover at least $\frac{L-L/2}{k} \geq \frac{L}{2k}$ items that were not credited to $\mathcal{P}_1, \mathcal{P}_2, \ldots,$ or $\mathcal{P}_{l-1}$. By the definition of availability to $\mathcal{P}_l$, this means that $L_l \geq L/2k$. Hence, if $L_l < L/2k$ for some l, then $R \geq L/2$, as claimed.

Let $T_l = R_l + R/k$. Then $Ex[T_l] \geq \Omega(\frac{L}{k\log n \log\frac{m}{k}})$ since we get that contribution from $Ex[R_l]$ if $L_l \geq L/2k$ and we get more than that from $Ex[R/k]$ otherwise.

Let T denote the sum of the T_l. Then $Ex[T] = 2Ex[R]$ and $Ex[T] \geq \Omega(\frac{L}{\log n \log\frac{m}{k}})$. Hence, $Ex[R] \geq \Omega(\frac{L}{\log n \log \frac{m}{k}})$, as claimed.

The preceding algorithm allowed the same set to be chosen more than once, even though each element was credited only once. In fact, it never helps to select a set a second time and so we can easily restrict our algorithm to select each set only once. In this case, we may select fewer than k sets overall.

2.2 Lower Bounds

The lower bounds we give hold even in the special case where the sets are disjoint, *i.e.*, every element belongs to precisely one set.

We first show that $m/(2k)$ is a lower bound on the competitiveness of any deterministic algorithm. This bound is obviously tight up to a factor of 2.

The adversary first presents elements taken from set number 1. Once the on-line algorithm commits to this set, no further elements from set 1 arrive, but elements from set number 2 are presented. This process is repeated until the on-line algorithm commits to k sets or at total of m elements have been presented overall.

For every set chosen by the on-line algorithm, the on-line benefit is 1 giving a total benefit of no more than k for all sets. If after k such sets at least $m/2$ elements were presented then the off-line algorithm accepts all these sets in advance and obtains a benefit of at least $m/2$. Otherwise at least $m/2$ elements belonging to set $k+1$ arrive. The off-line algorithm accepts this set obtaining a benefit of at least $m/2$ whereas the on-line algorithm remains at benefit k as it cannot accept any more sets. In any case the off-line benefit is at least $m/2$ whereas the on-line benefit is at most k. This implies the lower bound.

The randomized lower bound is more complicated. We prove it for the case when $k = 1$. A similar result can be proved for some larger values.

The competitive ratio of a randomized algorithm is defined as the supremum over all sequences of the ratio $b_{off}/E(b_{on})$. To prove the lower bound we apply a variation of Yao's theorem to the competitive ratios under consideration. (See ABM-93 for this variation). This allows us to replace randomness in the algorithm with randomness in the input. We will choose a distribution on the input sequence such that the expected competitive ratio of any deterministic algorithm is at least the desired lower bound when averaging over the possible sequence inputs.

Consider the following probability distribution over sequences of elements from the sets:

1. Choose integers y and z such that $n \geq 2^y z$ and $m \geq 2^z 2^y$.
2. Choose an integer $1 \leq i^* \leq z$ uniformly at random.
3. Construct sequences that consists of i^* phases while each phase consists of y steps. Associated with phase i step j, $0 \leq i < i^*$, $1 \leq j \leq y$, is a set of 2^{y-j} sets $S(i,j)$ $(S(i,j) \subset \{S_1, \ldots, S_n\})$. During phase i step j, 2^i different elements are presented from every set $S \in S(i,j)$.

   Throughout the sequence, no element is ever presented twice. (In fact, elements names are not significant, only the sets to which they belong, and every element belongs to exactly one set).

   The set of sets $S(i,1)$ consists of 2^{y-1} sets and is pairwise disjoint with sets of sets $S(r,1)$ for all $r \neq i$, $0 \leq i < i^*$.

   The set of sets $S(i,j)$, $j > 1$, (elements from whose elements are presented at phase i step j), is a random set of size 2^{y-j} out of the 2^{y-j+1} sets associated with the previous step $(S(i,j-1))$. In other words $S(i,j)$ is a random half of $S(i,j-1)$.

Note that the number of new sets in each phase is precisely 2^{y-1} and there are at most z phases. Thus the number of sets is at most $2^{y-1}z \leq n$. Moreover, the number of elements requested in phase i is less than $2^y 2^i$. Hence the total number of elements is bounded by $2^y 2^z \leq m$. This justifies the choice of y and z as a function of m and n.

Since the off-line algorithm knows the value of i^* and the set that will be used in the last step of phase i^*, it is easy for the off-line algorithm to get a benefit of $2^{i^*-1}y$.

Matters are more difficult for the on-line algorithm, however, since it knows neither piece of information. In fact, we will next show that the on-line algorithm can not do any better than picking a predetermined point at which to select the set, where the selected set is the set containing the item just presented.

Consider "simple" deterministic on-line algorithms of the following form: "wait until the element ℓ is presented, choose the set to which it belongs." We can now argue that given any deterministic on-line algorithm A for this problem with inputs drawn from the above distribution, there exists a simple deterministic algorithm A' such that the expected benefit of A', over the above distribution on input sequences, is at least the expected benefit of A on the same distribution. This is because nothing is significant in one sequence over another. (Up to reordering the input and the set lables, every input of a fixed length in the same.) Hence, we can successively modify A, without decreasing the expected benefit, until a simple algorithm is obtained.

It follows that to prove a lower bound on the competitive ratio it suffices to prove an upper bound on the expected ratio between the on-line benefit of simple deterministic algorithms and the adversary benefit.

A simple deterministic on-line algorithm has a single parameter ℓ as discussed above, this translates to choosing a set which has an element presented in phase i', step j' for some $0 \leq i' < z$, $1 \leq j' \leq y$.

Then the probability that $i^* = i' + s$ for $1 - i' \leq s \leq z - i'$ is $\frac{1}{z}$. Now, if $s \leq 0$ then the on-line benefit is zero because the sequence ends before the on-line algorithm chooses any set. If $s \geq 1$ then the on-line algorithm chooses some set $S \in S(i', j')$. The on-line benefit depends on the maximal value $j' \leq k \leq y$ for which $S \in S(i', k)$, in which case it is no more than $(k - j + 1)2^{i'}$. The conditional probability that $S \in S(i', k + r)$, $r \geq 0$, given that $S \in S(i', k)$, is 2^{-r}.

It now follows that the expected ratio between the on-line benefit and the adversary benefit is no more than

$$\left(\frac{1}{z}\right) \cdot \left(\frac{2^{i'}}{2^{i'}} + \frac{2^{i'}}{2^{i'+1}} + \frac{2^{i'}}{2^{i'+2}} + \cdots\right) \times \left(\frac{2^{-0} + 2^{-1} + 2^{-2} + \ldots}{y}\right) \leq \frac{4}{yz}.$$

This implies the $yz/4$ lower bound on the competitive ratio. Note that in terms of n and m the lower bound is $\log n \log m$ for wide range of values.

3 Picking Winners Repeatedly

We next consider the scenario where Bob has $\log n$ d-step jobs to run and where he may elect to kill a job and restart it on another workstation if he is not satisfied with the progress thus far. To make matters more difficult, we will assume that the jobs, call them $\mathcal{J}_1, \ldots, \mathcal{J}_{\log n}$, must be run in sequence (i.e., that $\mathcal{J}_{i+1}$ cannot be scheduled until $\mathcal{J}_i$ is completed for all $i < \log n$).

In what follows, we will show how to schedule the jobs so that all of them are completed with probability $1 - O(1/n)$. We will assume only that at least one workstation will eventually be available for $D \geq \alpha d \log n$ steps where α is a sufficiently large constant.

As before, the scheduling algorithm is quite simple. In this case, we flip $(1 + \epsilon)\log n$ coins (call them $\mathcal{C}_1, \mathcal{C}_1, \ldots, \mathcal{C}_{(1+\epsilon)\log n}$) for every pair (i, j) where $\mathcal{W}_i$ is available at step j. ($\epsilon = \Theta(1/\log\alpha)$ is a small constant that will be specified later.) The probability of a Heads for $\mathcal{C}_h$ will be

$$2^{-c_1 h} n^{c_2 x/D - 2}/d$$

where $c_1 = \Theta(\log\alpha)$ and $c_2 = \Theta(\log\alpha)$ are large constants that will be specified later and x is the number of steps that $\mathcal{W}_i$ has been available thus far. Whenever one of the coins is Heads for the first time (i.e., if the hth coin is Heads for the first time at step j of $\mathcal{W}_i$), then Bob schedules a job on $\mathcal{W}_i$. If a job was still running on another workstation, then it is killed and restarted on $\mathcal{W}_i$. Otherwise, the next job in the queue is scheduled on $\mathcal{W}_i$.

The proof that this algorithm works is similar to but more complicated than the proof for the case when Bob schedules a job only once. The complication arises because we need to overcome the $\log n$ barrier described in Remark 2.5 as well as dependence problems caused by occasionally having to kill jobs in order to be more efficient. We begin by getting good bounds on when each coin is likely to first come up Heads.

Each time that some workstation is available at a step, there is a chance (specifically, the probability is $2^{-c_1 h} n^{c_2 x/D-2}/d$) that $\mathcal{C}_h$ will come up Heads. Let m_h denote the sum of these probabilities up to and including the current flip. (In the case that $2^{-c_1 h} n^{c_2 x/D-2}/d > 1$, we still add the full amount into m_h. Then, it will be the case that $m_{h+1} = 2^{-c_1} m_h$ for all h.) We say that a coin is *early* if $\mathcal{C}_h$ comes up Heads for the first time when $m_h < 2^{-c_1/2}$ and *late* if $\mathcal{C}_h$ does not come up Heads while $m_h < 2^{c_1/2}$. In what follows, we show that the probability that a coin is either early or late is small.

Lemma 3.1 *For any h, the probability that $\mathcal{C}_h$ is early is at most $2^{-c_1/2}$ and the probability that $\mathcal{C}_h$ is late is at most $e^{-2^{c_1/2}}$.*

Proof: The probability that there is a Heads among the first r tosses of $\mathcal{C}_h$ is at most $p_1 + p_2 + \cdots + p_r$ where p_l is the probability of a Heads on the lth flip of $\mathcal{C}_h$ (over all workstations and steps). For $\mathcal{C}_h$ to be early, one of these tosses must result in Heads where $m_h = p_1 + p_2 + \cdots + p_r < 2^{-c_1/2}$. Thus, the probability that $\mathcal{C}_h$ is early is at most $2^{-c_1/2}$.

By similar reasoning, the probability that $\mathcal{C}_h$ is late is at most

$$\prod_{l=1}^{s}(1 - p_l) \leq e^{-(p_1 + \cdots + p_s)}$$

where $m_h = p_1 + \cdots + p_s \geq 2^{c_1/2}$. Thus, the probability that $\mathcal{C}_h$ is late is at most $e^{-2^{c_1/2}}$. □

Lemma 3.2 *With probability $1 - O(1/n)$, at most $\frac{\epsilon}{6}\log n$ coins will be either early or late.*

Proof: By Lemma 3.1 and the independence of the coins, the probability that $\frac{\epsilon}{6}\log n$ or more coins are either early or late is at most

$$\binom{(1+\epsilon)\log n}{\frac{\epsilon}{6}\log n}\left(2^{-c_1/2} + e^{-2^{c_1/2}}\right)^{\frac{\epsilon}{6}\log n}.$$

This probability is $O(1/n)$ provided that $\epsilon \leq 1$ and that $c_1 \geq 36/\epsilon$. Henceforth, we will assume that α is large enough and that c_1 and ϵ are selected so that both conditions are satisfied. □

Lemma 3.3 *For each $h < (1+\epsilon)\log n$, the probability that $\mathcal{C}_h$ first becomes Heads after $\mathcal{C}_{h+1}$ first becomes Heads is at most $2^{-c_1/2} + e^{-2^{c_1/2}}$.*

Proof: The step at which m_h first reaches $2^{c_1/2}$ is the same as the step at which m_{h+1} first reaches $2^{-c_1/2}$ (since $m_{h+1} = 2^{-c_1} m_h$ by definition). Thus in order for $\mathcal{C}_h$ to first become Heads *after* $\mathcal{C}_{h+1}$ first becomes heads, it must be the case that either $\mathcal{C}_h$ is late or $\mathcal{C}_{h+1}$ is early. The result then immediately follows from Lemma 3.1. □

Lemma 3.4 *For any $h \leq (1+\epsilon)\log n$, with probability $1 - O(c_2/\alpha + 2^{-c_1/2} + 1/n)$, $\mathcal{C}_h$ will eventually become Heads and it will first become Heads when flipped for a workstation which will be available for at least d steps before $\mathcal{C}_{h+1}$ first becomes Heads.*

Proof: The proof is similar to the proof in Section 2. In particular, let $\mathcal{S}$ denote the sample space of all flips for $\mathcal{C}_h$ and $\mathcal{C}_{h+1}$. Let $\mathcal{S}'$ denote the subspace of sample points for which $\mathcal{C}_h$ first becomes Heads at or before the step where $\mathcal{C}_{h+1}$ first becomes Heads and for which the first d flips of $\mathcal{C}_h$ for each workstation results in Tails. Let $\mathcal{S}_{\text{win}}$ denote the subspace of points for which the condition of the Lemma holds; namely, that $\mathcal{C}_h$ will first become Heads when flipped for a workstation that will be available for at least d steps before $\mathcal{C}_{h+1}$ first becomes Heads.

We first show that $Pr[\mathcal{S}']$ is close to 1. The probability of getting a Heads among the (at most) dn flips of $\mathcal{C}_h$ for which $x \leq d$ is at most

$$\begin{aligned} dn(2^{-c_1 h} n^{c_2 d/D-2}/d) &\leq n^{c_2 d/D-1} \\ &= \frac{2^{c_2/\alpha}}{n} \\ &= O(1/n) \end{aligned}$$

since $c_2 = O(\log \alpha)$.

The probability that there are no Heads among the last d flips of $\mathcal{C}_h$ for the workstation that is available for D steps is either 0 or at most

$$\begin{aligned} (1 &- 2^{-c_1 h} n^{c_2(D-d)/D-2}/d)^d \\ &\leq (1 - n^{-c_1(1+\epsilon)+c_2-c_2 d/D-2}/d)^d \\ &\leq e^{-n^{c_2(1-d/D)-c_1(1+\epsilon)-2}} \\ &\leq O(1/n) \end{aligned}$$

provided that $c_2 \geq 3+2c_1$, $\alpha \geq 1$ and $\epsilon \leq 1$. Henceforth, we will assume that α is large enough and that c_2 and ϵ are selected so that these conditions are satisfied.

Combining the previous two bounds with the bound of Lemma 3.3, we find that

$$Pr[\mathcal{S}'] \geq 1 - 2^{-c_1/2} - e^{-2^{c_1/2}} - O(1/n).$$

We next show that $Pr[\mathcal{S}_{\text{win}}]$ is large. The proof is nearly identical to that of Lemma 2.2. In particular, we construct an injection

$$f : \mathcal{S}' \rightarrow \mathcal{S}_{\text{win}}$$

for which $\forall s' \in \mathcal{S}'$

$$Pr[f(s')] \geq (1 - \Theta(c_2/\alpha))Pr[s'].$$

The injection is constructed by identifying the workstation $\mathcal{W}_i$ and flip x for which $\mathcal{C}_h$ is first Heads, and then changing the $(x-d)$th flip of $\mathcal{C}_h$ for $\mathcal{W}_i$ to be Heads instead of Tails. If $z' = 2^{-c_1 h} n^{c_2 x/D-2}/d$ is less than $1/2$, then we also change the xth flip of $\mathcal{C}_h$ for $\mathcal{W}_i$ to be Tails instead of Heads.

As a result, we can conclude that

$$\begin{aligned} Pr[\mathcal{S}_{\text{win}}] &\geq (1 - O(c_2/\alpha))Pr[\mathcal{S}'] \\ &\geq 1 - O(c_2/\alpha + 2^{-c_1/2} + 1/n). \end{aligned}$$ □

Lemma 3.5 *With probability $1 - O(1/n)$, for all but $\frac{\epsilon}{2}\log n$ values of $h \leq (1+\epsilon)\log n$, $\mathcal{C}_h$ will become Heads and will first become Heads for a workstation which will be available for at least d steps before $\mathcal{C}_{h+1}$ first becomes Heads.*

Proof:

We first consider the case when h is even. Then the probability that the condition of Lemma 3.4 holds will be independent for each h. In particular, the probability that the condition of Lemma 3.4 fails for more than $\frac{\epsilon}{4}\log n$ even values of h is at most

$$\binom{\frac{1+\epsilon}{2}\log n}{\frac{\epsilon}{4}\log n}(O(c_2/\alpha + 2^{-c_1/2} + 1/n))^{\frac{\epsilon}{4}\log n}.$$

This probability is $O(1/n)$ provided that $\epsilon \leq 1$, c_1 is a sufficiently large constant multiple of $1/\epsilon$, c_2 is $O(1/\epsilon)$, and ϵ is a sufficiently large constant multiple of $1/\log \alpha$. All of these conditions (as well as the prior constraints in the constants) can be met provided that α is a sufficiently large constant and where $\epsilon = \Theta(1/\log \alpha)$, $c_1 = \Theta(\log \alpha)$, and $c_2 = \Theta(\log \alpha)$.

An identical argument can be use to show that the probability that the condition in Lemma 3.4 fails for more that $\frac{\epsilon}{4}\log n$ odd values of h is at most $O(1/n)$. Thus, with probability $1 - O(1/n)$, the condition of Lemma 3.4 fails for at most $\frac{\epsilon}{2}\log n$ values of h. □

Theorem 3.6 *With probability $1 - O(1/n)$, the scheduling algorithm will result in all $\log n$ jobs being completed and at most $\epsilon \log n$ instances where a job is killed and restarted on another workstation.*

Proof:

Let I_h denote the interval of time between the step when $\mathcal{C}_h$ is first Heads and the step when $\mathcal{C}_{h+1}$ is first Heads. If both of $\mathcal{C}_h$ and $\mathcal{C}_{h+1}$ are neither early nor late, we say that I_h is *good*. Otherwise, we say that I_h is *bad*. By Lemma 3.2, we know that with probability $1 - O(1/n)$, there are at most $\frac{\epsilon}{3}\log n$ bad intervals.

Since no two good intervals overlap, we can again use Lemma 3.2 to show that with probability $1 - O(1/n)$, there are at most $\frac{\epsilon}{6}\log n$ good intervals I_h during which another coin (other than $\mathcal{C}_h$ or $\mathcal{C}_{h+1}$) first becomes Heads.

By Lemma 3.5, we know that with probability $1 - O(1/n)$, for all but at most $\frac{\epsilon}{2}\log n$ values of h, $\mathcal{C}_h$ will become Heads for a workstation which will be available for at least d steps before $\mathcal{C}_{h+1}$ first becomes Heads.

Combining the previous three facts and adding failure probabilities, we can conclude that with probability $1 - O(1/n)$, there are at least

$$(1+\epsilon)\log n - \frac{\epsilon}{3}\log n - \frac{\epsilon}{6}\log n - \frac{\epsilon}{2}\log n = \log n$$

values of h for which $\mathcal{C}_h$ first becomes Heads for a workstation that will be available for at least d steps before any other coin first becomes Heads. The job that is assigned to such a workstation is guaranteed to be completed before the algorithm attempts to schedule another job. Since only $(1+\epsilon)\log n$ attempts are made to schedule a job, only $\epsilon \log n$ can end in failure. □

The result in Theorem 3.6 can be shown to be tight or nearly tight in several respects. For example, by considering a randomized adversary of the type outlined in Remark 2.5 (with $\rho = 1/\log n$), it can be shown that if $D = O((d \log n)/\log\log n)$, then no scheduling algorithm will have better than a $\Theta(1/\log\log n)$ chance of scheduling even one job, no matter how many swaps and restarts it makes.

4 Extensions

4.1 Obtaining Higher Efficiency

The algorithm described in Section 3 is inefficient by a factor of α. This is because there is a workstation that was available for $D = \alpha d \log n$ steps, but we only completed $\log n$ d-step jobs with high probability. In what follows, we will show how to attain higher efficiency by allowing the user to schedule up to $\alpha/\epsilon = O(1)$ jobs at the same time. (Each workstation still only actively works on one job at a time, of course, and there are no precedence constraints between jobs scheduled at the same time.)

The improvement is quite simple. Assume that some workstation will be available for D^* steps. Then the user runs α/ϵ versions of the algorithm described in Section 3 with $d = \frac{D}{\alpha \log n}$ and $D = D^*/(2+1/\epsilon)$. Each version is assigned a unique priority and works independently from the others. (In fact, we can think of the α/ϵ versions as if each was being run by a separate user.) Whenever a higher-priority job is scheduled on a workstation, that workstation simply appears to be unavailable to all lower priority versions. (The reverse is not true. In other words, a workstation running a low-priority job will appear to be available to a higher-priority user.)

The key to proving that $(\alpha/\epsilon)\log n$ jobs are completed with high probability rests on the fact that each version can consume at most $(1+\epsilon)d\log n$ available steps. (This is because a version starts at most $(1+\epsilon)\log n$ jobs, each of which has length at most d.) Hence, even if all α/ϵ versions consume $(1+\epsilon)d\log n$ steps on a single workstation, some workstation will still be available for at least

$$\begin{aligned} D^* - \frac{\alpha}{\epsilon}(1+\epsilon)d\log n &= D(2+1/\epsilon) - \frac{D}{\epsilon}(1+\epsilon) \\ &= D \end{aligned}$$

steps, which means that every version of the algorithm will be able to complete $\log n$ jobs with high probability by Theorem 3.6.

The total amount of work accomplished by the α/ϵ versions is

$$\begin{aligned} \frac{\alpha}{\epsilon}d\log n = \frac{D}{\epsilon} &= \frac{D^*}{\epsilon(1/\epsilon+2)} \\ &= \frac{D^*}{1+2\epsilon} \geq (1-2\epsilon)D^*. \end{aligned}$$

By making ϵ be small, this amount can be made arbitrarily close to D^*.

The preceding analysis ignored the scenario when there is more than one workstation that is available for D^* steps. We show how to exploit the available capacity in multiple workstations in the following section where we also handle the case of multiple users.

4.2 The Case of Multiple Schedulers

One particularly nice aspect of our scheduling algorithm is that (with only small modifications) it can be used simultaneously by multiple individuals without coordination. (Alternatively, it can be used by a few individuals who want to use multiple workstations — this is reducible to the case where there are multiple users.)

For example, consider a scenario where there are at least k workstations that will be available for at least D^* steps each. In this case, each user will be instructed to use the algorithm of Section 3 with $D = D^*/(2+1/\epsilon)$. For simplicity, the users will be prioritized, but they will not otherwise interact. Each user runs the scheduling algorithm as if he/she were the only individual running background jobs. Workstations will be considered to be available to a user iff they are not running a higher-priority job. Higher-priority jobs will always interrupt lower-priority jobs. Knowledge of k is not needed and is used only for the purposes of the analysis. In order to attain near maximum use of the available time in the k workstations, we will assume that there are at least $(\alpha/\epsilon)k$ users, each with $\log n$ d-step jobs where $d = D/(\alpha \log n)$. Somewhat surprisingly, the $(\alpha/\epsilon)k$ users will get all of their $\log n$ jobs run (in sequence) with high probability. This is because each of the $(\alpha/\epsilon)k$ users can consume at most $(1+\epsilon)d\log n$ available time using the algorithm of Section 3. This means that at least one of the k workstations will still be available for

$$\begin{aligned} D^* &- \frac{(\alpha/\epsilon)k(1+\epsilon)d\log n}{k} \\ &= (2+1/\epsilon)D - (\frac{1+\epsilon}{\epsilon})D = D \end{aligned}$$

steps no matter what the $(\alpha/\epsilon)k$ users do. Hence, by Theorem 3.6, each of the $(\alpha/\epsilon)k$ users will get all $\log n$

jobs done with high probability. (Actually, we need to boost the success probability of the analysis in Theorem 3.6 in order to make the preceding result hold with probability $1-O(1/n)$, but this is easy to do by adjusting the constant factors.)

It is worth noting that the preceding result attains greater efficiency since we are able to accomplish

$$\frac{\alpha}{\epsilon}kd\log n = \frac{kD}{\epsilon} = \frac{kD^*}{\epsilon(1/\epsilon+2)} = \frac{kD^*}{1+2\epsilon} \geq (1-2\epsilon)kD^*$$

productive work.

4.3 The Case when D^* is Unknown

The algorithms described thus far used knowledge of D^* in order to schedule the jobs. In what follows, we show how to modify the algorithms so that dependence on D^* is no longer required.

In the case when D^* is not known, the scheduler partitions time into intervals as follows. The first interval lasts until some workstation has been available for $\alpha\log n$ steps. During this time, the scheduler runs the algorithm for d-step jobs where $d=1$.

For $i>1$ the ith interval starts after the $(i-1)$st interval has finished and lasts until some workstation has been available for $\alpha 2^{i-1}\log n$ steps (counting from the beginning of the interval, only). During this time, the scheduler runs the algorithm for d-step jobs where $d=2^i$.

Even without knowing the value D^*, $\log n$ jobs of length $O(D^*/\log n)$ will still be completed with probability $1-O(1/n)$. Jobs of shorter length will also be completed, although the number of killed jobs could grow as large as $\epsilon\log n\log(D^*/\log n)$. The algorithms in Sections 4.1 and 4.2 can be modified in a similar manner in order to improve efficiency.

In fact, if the users each have a unique priority, and if each user is allowed to run α/ϵ jobs at the same time, then for all k, the kth user will be able to get $(1-2\epsilon)D_k^*$ work done with high probability where D_k^* is the amount of time available on the kth most available machine.

5 Acknowledgments

We would like to thank Allan Borodin, Leonid Levin, Prabhakar Raghavan, Mike Sipser, Bob Tarjan, Al Vezza, and Joel Wein for helpful remarks, suggestions, and references.

References

[ABFR94] B. Awerbuch, Y. Bartal, A. Fiat, and A. Rosén. Competitive Non-Preemptive Call-Control. In *Proc. of the 5th Ann. ACM-SIAM Symp. on Discrete Algorithms*, pages 312-320, January 1994.

[ABF93] B. Awerbuch, Y. Bartal, and A. Fiat. Competitive Distributed File Allocation. In *Proc. of the 25th Ann. ACM Symp. on Theory of Computing*, pages 164-173, May 1993.

[ABM93] Y. Azar, A. Broder, and M. Mannase. On-line choice of on-line algorithms. In *Proc. 4th ACM-SIAM Symp. on Discrete Algorithms*, pages 432–440, 1993.

[AGH94] A. Aggarwal, Juan Garay, and Amir Herzberg. Adaptive video on demand. In *Proc. Thirteenth ACM PODC Symp.*, page 402, 1994. also appeared as an IBM Research Report, RC19770, Oct, 1994.

[BCLR95] S. Bhatt, F. Chung, T. Leighton, and A. Rosenberg. Optimal strategies for stealing cycles. Unpublished manuscript, 1995.

[BL94] R.D. Blumofe and C.E. Leiserson. Scheduling Multithreaded Computations by Work Stealing. In *Proc. of the 35th Ann. IEEE Symp. on Foundations of Computer Science*, pages 356-368, November 1994.

[CEL93] J. Cooperstock, R. El-Yaniv, T. Leighton. The Statistical Adversary Allows Online Foreign Exchange with no Risk. Proceedings of SODA '95.

[CV92] B. Chandra and S. Vishwanathan. Constructing Reliable Communication Networks of Small Weight On-line. Journal of Algorithms, 1992.

[EFKT92] R. El-Yaniv, A. Fiat, R. Karp, and G. Turpin. Competitive Analsys of Financial Games. In *Proc. of the 33th Ann. IEEE Symp. on Foundations of Computer Science*, pages 327-333, October 1992.

[EK93] R. El-Yaniv and R. Karp. The Mortage Problem. In *Proc. of the 2nd Ann. Israeli Symp. on Theoretical Computer Science*, May 1993.

[FKT89] U. Faigle, W. Kern and György Turán. On the Performance of On-Line Algorithms for Partition Problems. *Acta Cybernetica* 9, pages 107-119, 1989.

[HS92] M.M. Halldórsson and M. Szegedy. Lower Bounds for On-Line Graph Coloring. In *Proc. of the 3rd Ann. ACM-SIAM Symp. on Discrete Algorithms*, pages 211-216, January 1992.

[Irani90] S. Irani. Coloring Inductive Graphs On-Line. In *Proc. of the 31st Ann. IEEE Symp. on Foundations of Computer Science*, pages 470-479, October 1990.

[IW91] M. Imase and B.M. Waxman. Dynamic Steiner Tree Problem. In *SIAM Journal on Discrete Mathematics*, 4(3):369-384, August 1991.

[KP94] B. Kalyanasundaram and K.R. Pruhs. Fault-Tolerant Scheduling. In *Proc. of the 26th Ann. ACM Symp. on Theory of Computing*, pages 115-124, May 1994.

[KMRS88] A.R. Karlin, M.S. Manasse, L. Rudolph, and D.D. Sleator. Competitive Snoopy Caching. In *Algorithmica*, 3(1):79-119, 1988.

[KVV90] R.M. Karp, U.V. Vazirani, and V.V. Vazirani. An Optimal Algorithm for On-Line Bipartite Matching. In *Proc. of the 22rd Ann. ACM Symp. on Theory of Computing*, pages 352-358, May 1990.

[LT94] Richard J. Lipton and Andrew Tomkins. Online interval scheduling. In *Proc. 5th ACM-SIAM Symp. on Discrete Algorithms*, pages 302–311, Arlington, VA, January 1994.

[ST85a] D.D. Sleator and R.E. Tarjan. Amortized Efficiency of List Update and Paging Rules. In *Communications of the ACM*, 28(2) pages 202-208, 1985.

[ST85b] D.D. SLEATOR AND R.E. TARJAN. Self-Adjusting Binary Search Trees. *Journal of the ACM*, 32:652–686, 1985.

[Vish90] S. Vishwanathan. Randomized Online Graph Coloring. In *Proc. of the 31st Ann. IEEE Symp. on Foundations of Computer Science*, October 1990.

[WY93] J. Westbrook. and D.K. Yan. Greedy On-Line Steiner Tree and Generalized Steiner Problems. In *Proc. of the 3rd Workshop in Algorithms and Data Structures*, Also *Lecture Notes in Computer Science*, vol. 709, pages 622-633, Montréal, Canada, 1993, Springer-Verlag.