Delayed Information and Action in On-Line Algorithms

Susanne Albers* Moses Charikar† Michael Mitzenmacher‡

Abstract

Most on-line analysis assumes that, at each time step, all relevant information up to that time step is available and a decision has an immediate effect. In many on-line problems, however, the time relevant information is available and the time a decision has an effect may be decoupled. For example, when making an investment, one might not have completely up-to-date information on market prices. Similarly, a buy or sell order might only be executed some time later in the future.

We introduce and explore natural delayed models for several well-known on-line problems. Our analyses demonstrate the importance of considering timeliness in determining the competitive ratio of an on-line algorithm. For many problems, we demonstrate that there exist algorithms with small competitive ratios even when large delays affect the timeliness of information and the effect of decisions.

*Freie Universität Berlin and Max-Planck-Institut für Informatik. Address: Im Stadtwald, 66123 Saarbrücken, Germany. E-mail: albers@mpi-sb.mpg.de

†Computer Science Department, Stanford University, CA 94305, USA. Supported by a Stanford Graduate Fellowship, an ARO MURI Grant DAAH04-96-1-0007 and NSF Award CCR-9357849, with matching funds from IBM, Schlumberger Foundation, Shell Foundation, and Xerox Corporation. E-mail: moses@cs.stanford.edu.

‡Digital Systems Research Center, 130 Lytton Avenue, Palo Alto, CA 94301, USA. E-mail: michaelm@pa.dec.com.

1 Introduction

The theory of on-line algorithms deals with situations where a decision or a series of decisions must be made with limited information, and specifically without knowledge of future events. Implicit in this approach is the idea that the time information becomes available relative to the time decisions take effect can be of paramount importance in algorithm performance. In most on-line analyses, however, the setting chosen for study is the trivial one: at each time-step, all relevant information up to that time-step is available, and a corresponding decision is made.

In many on-line problems the time relevant information is available and the time a decision has an effect are decoupled. This phenomenon arises, for instance, in *investment problems* where one has to decide *whether or not* and *when* to buy an expensive piece of equipment. An example of such investment problems is the standard on-line ski rental problem. In these investment problems, once a decision is taken for buying equipment, it can take some time before the equipment is delivered to the user. For example, it can take a couple of days or weeks to ship a particular model of skis, and even months to deliver and install a new machine in a factory. In such cases, a decision to buy equipment has an effect only later in time and the action corresponding to the decision is delayed.

This can heavily influence the performance of an on-line strategy. As a simple example, consider the ski rental problem. Skis cost r dollars to rent per weekend and b to buy for a season. Suppose an avid skier skis every weekend there is good snow. Whether it is best for her to rent or buy skis for the season depends on the number of good ski weekends. In the on-line version of the problem, we wish to minimize the *competitive ratio*, which is the ratio of the amount actually spent compared with the amount spent by an omniscient optimal algorithm that knows the weather ahead of time. If the skier rents skis s times before buying, this ratio is $\frac{sr+b}{\min\{(s+1)r,b\}}$. When b/r is an integer, an optimal on-line algorithm is to rent skis $s = (b/r) - 1$ times, and then buy; this yields a competitive ratio of $2 - \frac{r}{b}$. If skis take $d \geq 1$ weeks to ship, the analysis of this problem is slightly more complex. If a skier decides to buy the on the sth weekend of snow, we must consider what happens in the intervening $d-1$ other weekends before the skis arrive. If i of the intervening weekends are snowy, then the worst-case ratio between the actual cost and the optimal cost is now

$$\max_{0 \leq i \leq d-1} \frac{(s+i)r+b}{\min\{(s+i)r,b\}}.$$

It is easily checked that this ratio is maximized at one of the extremes $i = 0, d-1$; using this, one can easily determine the best value of s.

In the above example, there is a delay between the time a decision is made and when it has an effect. We refer to this as the *delayed action* model. The parameter d is the maximum delay after which a decision takes effect. For this problem, $d = 0$ gives us the original problem without delay.

Similarly, there are problems where it is natural to consider information that arrives only after some delay. In this scenario, at time step t we might have information about the first $t-d$ time steps only, for some $d \geq 1$. This phenomenon arises, for instance, in on-line *financial games* where we have to devise strategies for converting money from one currency to another or for selecting a portfolio in the stock market [15, 16, 18, 34]. Naturally, we might not have access to the very latest exchange rates or stock prices. We refer to this as the *delayed information* model. Here, the case $d = 1$ corresponds to the original problem without delay.

Related timing problems occur when a *group* of people or agents takes decisions. The group might come together only at particular time instances. The actions are again delayed, in that they can only occur at specific points in time. For example, in the case of investing in manufacturing machinery, one may only be able to make budget decisions in concert with the rest of an organization at specific budgeting periods. Another example is that of an investment club, where a group of people pool their money together and invest in the stock market. All investment decisions can be made only at club meetings which occur at regular intervals of time, e.g. once a month.

We use the term *delayed models* to loosely describe models where there is this type of discontinuity between the time information is available and the time decisions take effect. Such models are naturally motivated by situations where one has incomplete information about the past or a decision will have a delayed effect on the state of the system. Interestingly, they also often have a natural interpretation in terms of a distributed agents acting with limited coordination. In particular, such models correspond nicely to distributed systems where information about the system is updated only after some delay or at specific synchronization points.

Our Contribution: In this paper, we consider several standard on-line problems and examine their generalizations to delayed models. These generalizations are generally quite natural and lead to interesting insight into the original problem. We note that in this initial exploration of delayed models, we have focused on cases where one can modify the original on-line analysis to analyze the delayed version. We believe that the resulting relative simplicity of many of our results demonstrates the naturalness and utility of this model. We expect, however, that delayed models will prove more difficult than their standard counterparts in many instances.

We briefly describe the remainder of the paper. In Section 2, we study the delayed information model applied to the classical problem of on-line scheduling on parallel machines to minimize the makespan. Here a scheduling algorithm must assign new jobs to processors bases on stale load information. Traditional algorithms for on-line scheduling do badly in this scenario. We develop new algorithms for this model and prove almost matching lower bounds. In Section 3, we study the list update problem in the delayed action model and prove nearly tight upper and lower bounds for deterministic on-line algorithms. We also show that a randomized on-line algorithm can only beat the deterministic lower bound if it uses paid exchanges. In Section 4, we generalize an on-line stock market model introduced in [15] by studying natural delayed models. Finally, in Section 5, we apply the delayed action model to the general class of relaxed metrical task systems [4, 9]. Relaxed task systems are an abstract model for problems where one has to decide when it is appropriate to make expensive configuration changes. This class includes the ski rental problem, page migration [13], file replication [13], network leasing [4], and other problems (see [9]). We extend the results of [4, 9] to apply to relaxed task systems with delayed action, effectively handling the delayed models of an entire general class of problems.

Related Work: In subsequent sections, we will mention related work relevant to the specific problems we consider. Here, we offer a brief overview of generally relevant related work.

The importance of when information becomes available has been noted previously, especially in the significant body of work on algorithms with lookahead, e.g. [12, 22, 24, 28]. In the case of on-line decision models, however, the possibility of not having up-to-date information is not generally addressed. For load balancing problems, the question has been considered for statistical models [30, 31, 37]. And recently, [5] considers an on-line load balancing setting where tasks gather some information about system behavior before making a choice of processor.

There is also a large body of work on algorithms with distributed agents, who must coordinate their efforts in the face of some cost for communication, e.g. [3, 6, 11]. These models, however, model communication as an instantaneous event, and hence the communication cost does not directly incorporate a notion of time and delay. Another line of research has addressed distributed decision making when the communication among agents is limited, for example by only allowing local communication. Implicitly this allows distant agents to communicate only after a number of communication rounds. The problems investigated include scheduling, load balancing, routing and general optimization [10, 17, 25, 32, 33].

2 Scheduling

We consider a classical problem in on-line scheduling. A sequence of jobs $J_1, J_2, \ldots$ must be scheduled on m identical parallel machines. Whenever a job arrives, the job must be scheduled immediately on one of the machines, without knowledge of any future jobs. Preemption of jobs is not allowed. The goal is to minimize the *makespan*, i.e., the completion time of the last job that finishes.

The problem was first investigated by Graham [21]. He developed the well-known *List* algorithm that always schedules a job on the least loaded machine. Graham's *List* algorithm is $(2 - \frac{1}{m})$-competitive. The currently best known competitive ratio for this problem is 1.923 obtained by Albers [1].

In a setting with delayed information, we do not have the current loads on the processors available to us. When we are presented with the ith job J_i, we have the loads on the machines from up to d_i requests ago. That is, we know the load after the job J_{i-d_i} was placed. (When $d_i = 1$ always, we have the original problem.) We must decide where to place job J_i based on this old information. We examine the setting where we have a bound on how old the information is at each stage, i.e. $d_i \leq d$, for some d. We will refer to the last $d_i - 1$ jobs whose contribution to the loads is not known as *unknown* jobs and other jobs as *known* jobs.

In this situation, the strategy of placing each job on the processor with the least known load does very badly. In fact the competitive ratio of that strategy can be as bad as $d+1-\frac{d}{m}$ (for $d \leq m$). The problem is that this strategy does not take into account the potential effect of unknown jobs. We will devise new algorithms with better competitive ratios, for two variants of scheduling with delayed information.

In our first model, we assume that in addition to the loads of the machines from d_i requests ago, we also know where the last $d_i - 1$ unknown jobs that were placed. It is simpler to work with a less stringent, but for our purposes equivalent, scenario where we have available a complete history of the process up to d_i requests ago. This scenario describes for instance a centralized scheduling algorithm where the size of every new job is not known to the scheduler immediately on arrival, but is revealed at most d requests later.

In this model, by using specific kinds of deterministic algorithms, we can figure out where the *unknown* jobs were scheduled as follows. Suppose we use a deterministic algorithm that bases its decision on the schedule from d requests ago, i.e. if $d_i < d$ pretend that the state seen by the algorithm is the schedule exactly d requests ago. Because we have complete information about the job history, we can also figure out the complete schedule from $d + 1$ requests ago, $d + 2$ requests ago and so on. Hence we can deduce the state seen by the algorithm while scheduling each of the previous $d - 1$ jobs, and thereby determine where each of the last $d - 1$ unknown jobs were scheduled.

For this model, we consider an algorithm we call *Delayed List* scheduling, as it generalizes Graham's List algorithm. Let w_i be the known load on machine i. (This is the load without the unknown jobs.) Let S denote the total known load on all the machines, i.e. $S = \sum_{i=1}^{m} w_i$. Let u_i be the number of unknown jobs on machine i. Define the *pseudo-load* on a machine to be $u_i + (m - u_i - 1)\frac{w_i}{S}$. The algorithm schedules the new job on the machine which has the lowest pseudo-load. (When $d = 1$, the algorithm is exactly the same as List.)

Lemma 1 *When the Delayed List algorithm places the current job on machine i, the load on machine i is at most $1 + u_i + (m - u_i - 1)\frac{w_i}{S}$ times the optimal load.*

Proof: Let x be the processing time for the ith job. Consider what happens if the algorithm tries to place the current job on machine i. Without loss of generality, suppose all the unknown jobs on machine i have the same processing time, say y. Then $\ell_i = w_i + u_i \cdot y + x$ will be the new load on machine i.

The sum of the processing times of all the jobs in the sequence is at least $S + u_i \cdot y + x$. Thus $OPT \geq \frac{S+u_i \cdot y+x}{m}$. Also, $OPT \geq x$ and $OPT \geq y$. Hence

$$\frac{\ell_i}{OPT} \leq \min\left(\frac{w_i + u_i \cdot y + x}{x}, \frac{w_i + u_i \cdot y + x}{y}, \frac{w_i + u_i \cdot y + x}{(S + u_i \cdot y + x)/m}\right).$$

We obtain the required bound on $\frac{\ell_i}{OPT}$ by maximizing the above function over all possible values of y and x. Let us maximize over y first. We wish to compute

$$\max_y \min\left(\frac{w_i + u_i \cdot y + x}{x}, \frac{w_i + u_i \cdot y + x}{y}, \frac{w_i + u_i \cdot y + x}{(S + u_i \cdot y + x)/m}\right)$$

Let

$$\begin{aligned} f_1(x, y) &= \frac{w_i + u_i \cdot y + x}{x} \\ f_2(x, y) &= \frac{w_i + u_i \cdot y + x}{y} \\ f_3(x, y) &= \frac{w_i + u_i \cdot y + x}{(S + u_i \cdot y + x)/m} \end{aligned}$$

Note that each of the three functions are monotone in y. We want to find the maximum of the lower envelope (i.e. minimum) of these three monotone curves. This must occur either at an end-point of the interval $y = 0$ or $y = \infty$ or at a point where two of the three functions are equal. Further, a point where two functions are equal is a potential maximum only if the value of the third function is greater than the two that are equal.

In fact, our analysis will show that the maximum is achieved when all three functions are equal.

1. Let us first consider the maximum value of the function for end-points of the interval. For $y = \infty$, the value of the function is 1. For $y = 0$, the value of the function is $\min(\frac{w_i+x}{x}, \frac{w_i+x}{(S+x)/m})$. This is maximized when $x = \frac{S+x}{m}$. Hence the maximum value is $1 + (m-1)\frac{w_i}{S}$.

   We now consider the three possible points where two of the functions are equal.

2. Suppose $f_1(x, y) = f_2(x, y) \leq f_3(x, y)$. This implies that $x = y \geq \frac{S+u_i \cdot y+x}{m}$. Hence $f_1(x, y) = f_2(x, y) = u_i + 1 + \frac{w_i}{x}$. Our bound is maximized for the smallest possible value of x. But we also have $x \geq \frac{S}{m-u_i-1}$. Hence, the maximum value is $u_i + 1 + (m - u_i - 1)\frac{w_i}{S}$.

3. Suppose $f_1(x, y) = f_3(x, y) \leq f_2(x, y)$. This implies that $x = \frac{S+u_i \cdot y+x}{m} \geq y$. Hence $f_1(x, y) = f_3(x, y) = m - \frac{S-w_i}{x}$. Our bound is maximized for the largest possible value of x. But we also have $x \leq \frac{S}{m-u_i-1}$. Hence, the maximum value is $u_i + 1 + (m - u_i - 1)\frac{w_i}{S}$.

4. Suppose $f_2(x, y) = f_3(x, y) \leq f_1(x, y)$. This implies that $y = \frac{S+u_i \cdot y+x}{m} \geq x$. Algebraic manipulation yields $f_2(x, y) = f_3(x, y) = u_i + (m - u_i)\frac{w_i+x}{S+x}$, which is increasing in x since $w_i \leq S$. Our bound is maximized for the largest possible value of x. But we also have $x \leq \frac{S}{m-u_i-1}$. Hence, the maximum value is $u_i + 1 + (m - u_i - 1)\frac{w_i}{S}$.

In all cases, $\frac{\ell_i}{OPT} \leq 1 + u_i + (m - u_i - 1)\frac{w_i}{S}$. ■

We use the result of Lemma 1 to bound the competitive ratio of the algorithm.

Theorem 2 *The Delayed List algorithm is* $2 + \frac{d-2}{m}$ *competitive.*

Proof: The algorithm schedules the current job on the machine i which has the lowest value of $c_i = 1 + u_i + (m - u_i - 1)\frac{w_i}{S} \geq \frac{\ell_i}{OPT}$. Now,

$$\sum_{i=1}^{m} c_i \geq \sum_{i=1}^{m} \left[1 + u_i + (m-1)\frac{w_i}{S}\right] = m + d - 1 + m - 1$$

because $\sum_{i=1}^{m} w_i = S$. Hence there must be some c_i with value at most $\frac{2m+d-2}{m} = 2 + \frac{d-2}{m}$. Thus, the competitive ratio of the algorithm is at most $2 + \frac{d-2}{m}$. ■

Theorem 2 shows that by spreading out the unknown jobs appropriately, we can achieve a competitive ratio that grows at a "rate" of d/m. In fact, the analysis in the proof of Lemma 1 shows that given S, x, u_i, and w_i, one can compute precisely the worst case competitive ratio if the algorithm places the current job on machine i. This is a function of S, x, u_i, and w_i, and an exact expression can be obtained. A more intelligent algorithm would compute this function for each machine and place the current job on that machine that minimizes this function. Indeed, this improves the competitive ratio slightly, although it seems difficult to develop a general bound with a better form than Theorem 2. (As an exercise, the interested reader may wish to show that for $d = 2$ this more intelligent algorithm is at worst $2 - \frac{1}{m^2-m+1}$ competitive.) Moreover, the result of Theorem 2 is nearly tight, as the following lower bound shows.

Theorem 3 *There exist sequences where the competitive ratio of any deterministic algorithm for the delayed scheduling problem is* $2 + \frac{d-3}{m+1}$ *when this number is an integer less than or equal to* m.

Proof: Let A be a deterministic algorithm for the delayed scheduling problem with maximum delay d. For the lower bound, assume that when A receives job J_i, it knows the entire schedule after job J_{i-d} was placed. Suppose $d = (r-2)m + r + 1$ for an integer r. We will construct a request sequence consisting of $(r-1)m + 1$ jobs such that the optimal load is 1, but some machine in A's schedule has load r.

The first $m - r$ requests are jobs of size 1. The next $(r-2)m + r + 1$ jobs have size either 0 or 1. An adversary selects at most r of these to have size 1 as follows. Let $f(i)$ be the machine number on which A places job J_i if J_i has size 1. Then, $f(i)$ is a function of a prefix of the entire job sequence, where the prefix has length at most $m - r$. Thus $f(i)$ is a deterministic function, not dependent on the adversary's choices. Consider the sequence of numbers $f(1), \ldots f((r-1)m + 1)$. Now, there must be some machine x that occurs at least r times in this sequence. The adversary chooses r jobs $J_{i_1}, \ldots J_{i_r}$ to be of size 1 such that $f(i_j) = x$ for $1 \leq j \leq r$. It follows that these r jobs end up on machine x in A's schedule. On the other hand, the optimal makespan for this sequence is 1. Thus, the competitive ratio is at least $r = 2 + \frac{d-3}{m+1}$. ■

We now consider a second variant of the problem and a corresponding algorithm. In this scenario, when we are presented with a job J_i, we know the loads on the machines from d_i requests ago, but we do not know the actual schedule or job sizes corresponding to these loads. We assume, however, that each job knows i knows its sequence number i, and the number of jobs already scheduled, or $i - d_i$. (Implicitly, the number of scheduled jobs is increasing, so $i - d_i < k - d_k$ when $i < k$.) Our algorithm will make use of this information in its scheduling decision. This model corresponds to a distributed system where tasks may place themselves on an appropriate server before other tasks reveal their processing times, but through simple shared counters limited information such as the values of i and $i - d_i$ is maintained.

We provide an algorithm for this scenario called the *Delayed Avoid Heavy* algorithm. We describe what happens when the ith job J_i arrives. We say the machine with the kth smallest load from known jobs at this time has rank k. The algorithm uses a constant c as a parameter; this will be specified later. We never schedule a job on the heaviest m/c machines. (For convenience, we will assume that m/c is integral throughout.) Let $b = m(1 - 1/c)$, i.e. the number of machines excluding the heaviest m/c. Let $f(J_i) = (2i - d_i)$. The Delayed Avoid Heavy algorithm schedules job J_i on the machine with rank $b - (f(J_i) \bmod b)$.

For the purpose of analysis, we will divide the jobs into groups. Job J_i is placed in group number $\lfloor f(J_i)/b \rfloor$. The proofs of Lemmas 4 and 5 appear in the appendix.

Lemma 4 *Two jobs J_i and J_k in the same group are assigned to different machines.*

Lemma 5 *The competitive ratio of the Delayed Avoid Heavy algorithm is at most $2 + \frac{2d-2}{b} + c$.*

Substituting $b = m(1-1/c)$ and optimizing for c, we get that, for $c = 1+\sqrt{\frac{2d-2}{m}}$, the competitive ratio of the Delayed Heavy Load algorithm is bounded by $2 + \frac{2d-2}{m} + 2\sqrt{\frac{2d-2}{m}}$. It is possible to get slightly better bounds by being a bit more careful in Lemma 5. However, the expressions that result are far from elegant and the improvements are very minor, so we choose to omit them. The main point is that in this more limited model, by again spreading out the unknown jobs appropriately, we can achieve a competitive ratio that grows at a "rate" of about $2d/m$.

3 List Update

The list update problem is a fundamental problem in the theory of on-line algorithms. It consists of maintaining an unsorted list so as to minimize the total cost of accesses on a sequence of requests.

Formally, we are given n items that are stored in an unsorted linear linked list. A list update algorithm receives a sequence of *requests*, where each request specifies one item in the list. To *serve* a request the algorithm must *access* the requested item, i.e., it starts at the front of the list and proceeds linearly through the items until the desired item is found. Serving an access to the item at position i in the list incurs a cost of i.

In the standard problem, the list may be updated at any time. More specifically, after each request the accessed item may be moved at no extra cost to any position closer to the front of the list. These exchanges are called *free exchanges*. At any time, two adjacent items in the list may be exchanged with cost 1; these exchanges are called *paid exchanges*. The goal is to serve a sequence of requests so that the total cost is as small as possible.

In the problem with delayed action, we assume that an on-line algorithm may update the list only at the end of a *round*, where every round consists of d consecutive requests in the request sequence. Items requested during the round may be moved closer to the front of the list using free exchanges before the next round. Items not requested in the round can be moved only using paid exchanges. Note that when $d = 1$, we have the original standard problem.

To motivate the delayed model, consider the case where the linked list data structure is a shared object among a number of agents. In this case agents may read the list simultaneously without any problems; however, while the data structure is being updated, it may be necessary for consistency to lock the structure. In this case infrequent updates may provide better overall performance.

In the following we concentrate on deterministic on-line algorithms. When analyzing an on-line algorithm, we consider two types of *adversaries* that generate a request sequence and serve the generated sequence *off-line*. The *standard adversary* may update the list after each request. The *limited adversary* can update the list only at the end of each round. We call a deterministic list update algorithm A c-competitive against any standard (limited) adversary ADV if there exists a constant a such that, for every request sequence σ generated by a standard (limited) adversary and for all list lengths n, $C_A(\sigma) \leq c \cdot C_{ADV}(\sigma) + a$.

For the standard list update problem, Sleator and Tarjan [36] showed that the well-known on-line algorithm Move-To-Front (MTF) is 2-competitive. This algorithm moves an item to the front of the list each time it is accessed. This is the best competitive ratio any deterministic on-line algorithm can obtain in the standard model [27].

We prove lower and upper bounds for deterministic on-line algorithms in the list update problem with delayed action.

Theorem 6 *Let A be a deterministic on-line algorithm for the list update problem with delayed action. If A is c-competitive, then $c \geq d$. This lower bound holds for both types of adversaries.*

Proof: In each round the adversary issues d requests to the item that is stored at the last position in A's list. Thus, in each round A incurs a cost of dn.

At the end of each round, the adversary moves the item requested in the next round to the front of the list using paid exchanges. Thus, its cost in each round is at most $n + d - 1$. The ratio of the cost incurred by A to the cost incurred by the adversary is

$$\frac{nd}{n+d-1} = \frac{d}{1+(d-1)/n}$$

and, for large values of n, this expression can be arbitrarily close to d. ■

Next we study an adaptation of the MTF algorithm to the model of delayed action.

Algorithm MTF(d): At the end of each round, the algorithm moves the requested items to the front of the list. At the head of the list, for any two items i and j requested in the round, i precedes j if and only if the last request to i is more recent than the last request to j.

Theorem 7 *The algorithm MTF(d) is $(d+1)$-competitive. This upper bound holds for both types of adversaries.*

Note that for $d = 1$ we obtain the upper bound of 2 achieved by the MTF algorithm in the standard list update problem.

Proof: See the appendix. ■

It is straightforward to modify the above theorem and show

Corollary 8 *If each item is requested at most k times in a round, then MTF(d) is $(k+1)$-competitive.*

This corollary shows that if one is attempting to choose a value of d to balance reading and writing costs, a key parameter to consider is how often items can be requested repeatedly.

Next we consider randomized on-line algorithms and give two lower bounds. The proofs appear in the appendix. None of the randomized on-line algorithms that have been presented so far for the standard list update problem uses paid exchanges, see e.g. [2, 35]. We show that such algorithms cannot be better than d-competitive in the setting with delayed action.

Theorem 9 *Let A be a randomized on-line algorithm for the list update problem with delayed action and suppose that A does not use paid exchanges. If A is c-competitive against any oblivious adversary, then $c \geq d$. This lower bound holds for both types of adversaries.*

If a randomized on-line algorithm uses paid exchanges, our lower bound is slightly weaker.

Theorem 10 *Let A be a randomized on-line algorithm for the list update problem with delayed action and suppose that A does use paid exchanges. If A is c-competitive against any oblivious adversary, then $c \geq d/2$. This lower bound holds for both types of adversaries.*

4 Stock Trading

We consider an on-line stock market model studied in [15] based on similar probabilistic models used for stock price fluctuations (see, e.g., [23]). Consider a game where at each step, the price of a stock either increases by a constant factor $\alpha > 1$ or decreases by a factor $1/\alpha$. The game lasts for n steps, and the price moves up for m of these steps. At each step, one can invest a fraction s of one's wealth in the stock and the rest in cash. If the price moves up, the *return* from that step is the factor $\alpha s + 1 - s$ that the player's wealth increases; if the price moves down, the return $\frac{s}{\alpha} + 1 - s$ is less than 1. The *total return* is the factor by which the player's wealth increases over the course of the game. Following [15], we say in this setting that the on-line trader plays against an (α, m, n)-adversary if an adversary determines the price fluctuations subject to the initial constraints.

We review the relevant results from [15]. Let $R_\alpha(m, n)$ be the optimal on-line return against the (α, m, n)-adversary. We have boundary conditions $R_\alpha(n, n) = \alpha^n$ and $R_\alpha(0, n) = 1$. As the optimal algorithm obtains a return of α^m by investing fully whenever the price will go up, studying the on-line return in sufficient to find the competitive ratio. The return $R_\alpha(m, n)$ satisfies the recurrence

$$R_\alpha(m,n) = \max_{0\le s\le 1} \min\{(\alpha s + 1 - s)\cdot R_\alpha(m-1, n-1), (\frac{s}{\alpha} + 1 - s)\cdot R_\alpha(m, n-1)\},$$

and if we define the partial binomial sum $B(k; n, p) = \sum_{i=0}^{k} \binom{n}{i} p^i (1-p)^{n-i}$, then the solution to the recurrence satisfies

$$R_\alpha^{-1}(m,n) = B(n-m-1; n-1, \frac{\alpha}{\alpha+1}) + \alpha^{n-2m} B(m-1; n-1, \frac{\alpha}{\alpha+1}).$$

An interesting consequence is that even if the number of up movements m is less than the number of down movements, that is $m < \frac{n}{2}$, the on-line player can make a profit. In fact this holds true even if $m = 1$.

We consider an extension of this model to two delayed models. In the first model, we consider the problem when the player initially sets a fraction s of his wealth to remain invested over the next d time steps, and can only change the investment s every d time steps. This model might apply, for example, to an investor who only performs trades at specific or less frequent time intervals, and is unwilling to follow every change in the market. Without loss of generality we assume that n is a multiple of d.

We let $P_\alpha(d, m, n)$ be the optimal on-line return for a player playing against an (α, m, n)-adversary who can change its investment only every d steps. (Of course $P_\alpha(1, m, n) = R_\alpha(m, n)$.) For convenience we drop the α from the notation where the meaning is clear. Also, we call every set of d steps a *round*.

Note then that $P(d, m, n)$ satisfies the following recurrence:

$$P(d,m,n) = \max_s \min_{\substack{i \\ 0\le i\le d,m}} P(d, m-i, n-d)(\alpha^{2i-d}s + 1 - s).$$

That is, for each round, the optimal player chooses the investment s that maximizes his return regardless of the number of up movements the adversary chooses.

Interestingly, the behavior in this delayed model depends precisely on whether the period length d is even or odd.

Lemma 11 *For d even, $P(d, m, n) = 1$ if $m \le n/2$ and $P(d, m, n) = \alpha^{2m-n}$ if $m \ge n/2$.*

Proof: If $m \leq n/2$, then the adversary can arrange so that each round has at least as many down moves as up moves, and hence no round has a return greater than 1. Of course the player can guarantee a return of 1 by not investing, i.e. choosing $s = 0$ in each round.

Similarly, if $m \geq n/2$, then the player can guarantee a total return of α^{2m-n} by investing everything each round, i.e., always choosing $s = 1$. The adversary can ensure that no greater return is possible by alternating up and down moves on the first $2(n-m)$ steps. ■

The analysis for d odd generalizes and makes use of the result from [15] corresponding to the case $d = 1$.

Lemma 12 *Let $N = \frac{n}{d}$ and $M = m - \lfloor \frac{d}{2} \rfloor \frac{n}{d}$. For d odd, $P(d, m, n) = 1$ if $m \leq \lfloor \frac{d}{2} \rfloor N$, $P(d, m, n) = \alpha^{2m-n}$ if $m \geq \lceil \frac{d}{2} \rceil N$, and $P(d, m, n) = R_\alpha(M, N)$ otherwise.*

Proof: The trivial cases where $m \leq \lfloor \frac{d}{2} \rfloor n$ or $m \geq \lceil \frac{d}{2} \rceil n$ handled as in Lemma 11.

Otherwise, the problem is more interesting. We first show in this case that $P(d, m, n) \leq R_\alpha(M, N)$. Suppose that the adversary announces that in each round, there will either be $\lceil \frac{d}{2} \rceil$ or $\lfloor \frac{d}{2} \rfloor$ up moves. Then, in total, each round the invested value changes by a factor of α or $1/\alpha$, and there are M up rounds out of the N total rounds. In this case, the problem reduces to the standard case ($d = 1$) from [15]. In particular, the adversary can guarantee a competitive ratio of no more than $R_\alpha(M, N)$.

To prove the other direction, $P(d, m, n) \geq R_\alpha(M, N)$ we must show that the adversary cannot gain by using any other strategy. We use induction on n. The base case is trivial.

Now suppose the adversary uses $\lfloor \frac{d}{2} \rfloor + j$ up moves in the first round. By induction, the return for the subsequent rounds is $R_\alpha(M - j, N - 1)$. Simple algebraic manipulation (by determining the investor's first investment) yields that the payoff from the first round is

$$\frac{\alpha^{2j-1} - 1}{\alpha - 1}(R_\alpha^{-1}(M - 1, N - 1) - R_\alpha^{-1}(M, N)) + R_\alpha^{-1}(M, N).$$

Hence we have left to show that

$$\left[\frac{\alpha^{2j-1} - 1}{\alpha - 1}(R_\alpha^{-1}(M - 1, N - 1) - R_\alpha^{-1}(M, N)) + R_\alpha^{-1}(M, N)\right] R_\alpha(M - j, N - 1) \geq R_\alpha(M, N).$$

This is a combinatorial identity that can be checked in a straightforward but quite tedious manner; we spare the reader the details. ■

Next we consider our second delayed model. Suppose that information about trades is continuously updated, but remains d steps behind. That is, we only know the results from the first trade after the $(d+1)$st trade completes. Investors can again invest a fraction of their wealth each step (even though they may not have accurate knowledge of how much wealth they have, since not all trade results are known). This model accounts for situations where one receives updates on prices, but not in real-time. Surprisingly, we can show that there exist money-making schemes for arbitrarily large d even when there is only 1 up day. The proof appears in the appendix.

Theorem 13 *There exist money-making schemes for $m = 1$, regardless of n and d.*

5 Delayed Relaxed Task Systems

In this section, we will consider the delayed action model applied to relaxed metrical task systems [4, 9]. An example of a relaxed metrical task system is the ski rental problem described in the introduction. Another example of a relaxed metrical task system is the k-page migration problem

[9, 13]. For this problem, we wish to keep k copies of a page available on a network. When a processor wishes to access a page, it requests a copy from a processor holding that page. The communication cost incurred is proportional to the distance between processors. Alternatively, a page copy may migrate from one processor to another, at a higher communication cost proportional to the distance between processors. In the delayed model, we assume that the time to transfer a page is non-negligible, and hence there is a time between when a migration begins and ends during which the old copy serves these requests.

A relaxed metrical task system is associated with a parameter D and an underlying metrical task system with the same set of configurations. A configuration change in the relaxed task system is D times more expensive than the corresponding change in the underlying task system. Conveniently, we can demonstrate how to find a competitive algorithm for a relaxed metrical task system in the delayed action model, given a competitive algorithm for the associated metrical task system. Hence we can effectively handle an entire general class of problems, generalizing the work of [4, 9] on relaxed metrical task systems to the setting of delayed actions. We begin by defining metrical task systems [14], and then define relaxed metrical task systems. Here we follow [9].

Definition 14 *A* task system, *$\mathcal{P}$, consists of a set of configurations (or states) $\mathcal{C}$ and a distance function between any two configurations $C_1, C_2 \in \mathcal{C}$, denoted* $\mathrm{dist}(C_1, C_2)$. *(this is the* move cost *between the configurations). The task system consists of a set of requests, called tasks. A task r is associated with a service cost in each configuration, denoted $task(C, r)$ (this is the* task cost*). An algorithm for $\mathcal{P}$ is associated with a configuration C_1. Given a request r, the algorithm may serve it by moving to configuration C_2 paying a cost of* $\mathrm{cost}(C_1, C_2, r) = \mathrm{dist}(C_1, C_2) + task(C_2, r)$. *If the move cost function* dist *forms a metric space over $\mathcal{C}$, then the task system is called* metrical.

Definition 15 *A D-*relaxed *task system, D-$\mathcal{P}$, with respect to a task system $\mathcal{P}$ and some parameter $D \geq 1/2$, is the task system with cost, distance, and task functions denoted* cost_D, dist_D *and $task_D$ respectively.* dist_D *and $task_D$ are defined as follows: Given $C_1, C_2 \in \mathcal{C}$,* $\mathrm{dist}_D(C_1, C_2) = D \cdot \mathrm{dist}(C_1, C_2)$. *Given $C \in \mathcal{C}$ and a task r, $task_D(C, r) = \min_{C'} \mathrm{dist}(C, C') + task(C', r)$.*

Consider an algorithm for a task system $\mathcal{P}$. Suppose the algorithm starts out in configuration C_0. It receives a sequence of requests $r_1, r_2, \ldots$. When request r_i is received, the algorithm is in configuration C_{i-1}. The algorithm first moves to configuration C_i and then services request r_i from this configuration. The cost of the configuration change is $\mathrm{dist}(C_{i-1}, C_i)$ and the request service cost is $\mathrm{task}(C_i, r_i)$. In the delayed action model, we distinguish between the *real* state of the algorithm and the *ideal* state of the algorithm. Ideally, the algorithm should be in configuration C_i when it is just about to service request r_i. However, state changes may not be instantaneous, but occur only after a certain delay. Hence, the algorithm's state may not be C_i, but some earlier state C_{i-d_i}, where d_i is some delay parameter. Thus, the algorithm must service the request C_i from state C_{i-d_i}. The request service cost is therefore $\mathrm{task}(C_{i-d_i}, r_i)$. Eventually, the algorithm's real state will go through the same sequence of states as the ideal state, i.e. $C_0, C_1, C_2, \ldots$. Thus, we can think of the configuration change cost as $\mathrm{dist}(C_{i-1}, C_i)$, even though the configuration change may not occur right away. We will assume that the delay is bounded by d, i.e. $d_i \leq d$ for some d. Note that the case $d = 0$ gives us the original task system. We consider algorithms for task systems in the delayed action model and determine their competitive ratio as a function of the maximum delay d. For the analysis, we assume that the adversary does not have any delay associated with its configuration changes.

For an arbitrary metrical task system $\mathcal{P}$, the delayed action model may not be meaningful. In fact, there are task systems $\mathcal{P}$ such that, in the delayed action model, it is impossible to have a finite competitive ratio even for delay $d = 1$, even if there is an algorithm with finite competitive ratio for $d = 0$. For example, this could happen in the case of *forcing* task systems, where the request service costs are either 0 or ∞. For relaxed task systems, however, the delayed action model is meaningful, as we now show.

Let $\mathcal{P}$ be a metrical task system. Let $\text{task}(C, r)$ be the cost of servicing request r from configuration C in $\mathcal{P}$. Let $C_{min}(C, r)$ denote any configuration C' which minimizes $\text{dist}(C, C') + \text{task}(C', r)$. Let $\text{task}_D(C, r)$ be the cost if servicing request r from configuration C in D-$\mathcal{P}$. Then $\text{task}_D(C, r) = \text{dist}(C, C') + \text{task}(C', r)$, where $C' = C_{min}(C, r)$.

Consider an algorithm for D-$\mathcal{P}$. The total cost in servicing a sequence of requests $r_1, r_2, \ldots, r_n$ by moving through the sequence of states $C_O, C_1, C_2, \ldots, C_n$ is

$$\begin{aligned} & \sum_{i=1}^{n} \text{dist}_D(C_{i-1}, C_i) + \sum_{i=1}^{n} \text{task}_D(C_i, r_i) \\ = \; & D \sum_{i=1}^{n} \text{dist}(C_{i-1}, C_i) + \sum_{i=1}^{n} \left(\text{dist}(C_i, C'_i) + \text{task}(C'_i, r_i)\right) \end{aligned}$$

where $C'_i = C_{min}(C_i, r_i)$.

On the other hand, the cost of servicing the request sequence in the delayed model is

$$\begin{aligned} & \sum_{i=1}^{n} \text{dist}_D(C_{i-1}, C_i) + \sum_{i=1}^{n} \text{task}_D(C_{i-d_i}, r_i) \\ \leq \; & D \sum_{i=1}^{n} \text{dist}(C_{i-1}, C_i) + \sum_{i=1}^{n} \left(\text{dist}(C_{i-d_i}, C'_i) + \text{task}(C'_i, r_i)\right) \\ \leq \; & D \sum_{i=1}^{n} \text{dist}(C_{i-1}, C_i) + \sum_{i=1}^{n} \text{dist}(C_{i-d_i}, C_i) + \sum_{i=1}^{n} \left(\text{dist}(C_i, C'_i) + \text{task}(C'_i, r_i)\right) \\ \leq \; & D \sum_{i=1}^{n} \text{dist}(C_{i-1}, C_i) + \sum_{i=1}^{n} \sum_{j=i-d+1}^{i} \text{dist}(C_{j-1}, C_j) + \sum_{i=1}^{n} \left(\text{dist}(C_i, C'_i) + \text{task}(C'_i, r_i)\right) \\ \leq \; & (D + d) \sum_{i=1}^{n} \text{dist}(C_{i-1}, C_i) + \sum_{i=1}^{n} \left(\text{dist}(C_i, C'_i) + \text{task}(C'_i, r_i)\right) \end{aligned}$$

Thus for the purpose of analysis, we can think of the delayed model as being equivalent to model without delay where the cost of moving from configuration C_1 to C_2 is $(D + d)\text{dist}(C_1, C_2)$ and the request service cost is the same as before. The cost estimate we get using this approximation is an upper bound on the actual cost incurred by the algorithm in the delayed model. On the other hand, since we compare with an adversary that does not face delays, the cost for the adversary is the same as for the relaxed task system without delays. This considerably simplifies the analysis. In particular, this means that if we use the same algorithm for the delayed model as for the original relaxed task system, the cost increases by at most a factor of $(1 + \frac{d}{D})$. Hence if A is a c competitive algorithm for the relaxed task system without delays, then A is a $c(1 + \frac{d}{D})$ competitive algorithm for the relaxed task system in the delayed model.

Since the results of [4, 9] show how to turn competitive algorithms for metrical task system into competitive algorithms for relaxed metrical task systems, we now have a means of turning competitive algorithms for metrical task system into competitive algorithms for relaxed metrical task system in the delayed model. The above observation shows that the competitive ratio we achieve for the delayed model is at most a factor of $(1 + \frac{d}{D})$ times the competitive ratio for the original relaxed task system. In fact, it is possible to improve on this observation and get better competitive ratios by modifying the algorithm and/or the analysis of [4, 9] to tailor them to the delayed model. We state some results in the accompanying appendix; their proofs (which are long but not complex) will appear in the full version.

References

[1] S. Albers. Better Bounds for Online Scheduling. In *Proc. 29th Ann. ACM Symp. on Theory of Computing*, pp. 130–139, 1996.

[2] S. Albers, B. von Stengel and R. Werchner. A Combined BIT and TIMESTAMP algorithm for the List Update Problem. *Information Processing Letters*, 56:135–139, 1995.

[3] N. Alon, G. Kalai, M. Ricklin, and L. Stockmeyer. Lower Bounds on the Competitive Ratio for Mobile User Tracking and Distributed Job Scheduling. In *Proc. 33rd Ann. Symp. on the Foundations of Computer Science*, pp. 334-343, 1992.

[4] B. Awerbuch, Y. Azar, and Y. Bartal. On-line Generalized Steiner Problem. In *Proc. 7th Ann. ACM-SIAM Symp. on Discrete Algorithms.* pp. 68–74, 1996.

[5] B. Awerbuch, Y. Azar, A. Fiat, and T. Leighton. Making Commitments in the Face of Uncertainty: How to Pick a Winner Almost Every Time. In *Proc. 28th Ann. ACM Symp. on Theory of Computing*, pp. 519–530.

[6] B. Awerbuch, Y. Bartal and A. Fiat. Competitive Distributed File Allocation. In *Proc. 25 ACM Symp. on Theory of Computing*, pp. 164–173, 1993.

[7] B. Awerbuch, S. Kutten and D. Peleg. Competitive Distributed Scheduling. In *Proc. 24th Ann. ACM Symp. on Theory of Computing*, pp. 571–580, .

[8] Y. Azar, Y. Bartal, E. Feuerstein, A. Fiat, S. Leonardi and A. Rosen. On Capital Investment. In *Proc. 23rd International Colloquium on Automata, Languages, and Programming (ICALP96)*, Springer LNCS, Volume 1099, pp. 514–525, 1996.

[9] Y. Bartal, M. Charikar, and P. Indyk. On Page Migration and Other Relaxed Task Systems. In *Proc. 8th Ann. ACM-SIAM Symp. on Discrete Algorithms.* pp. 43–52, 1997.

[10] Y. Bartal, J. Byers, and D. Raz, Global Optimization Using Local Information with Applications to Flow Control. In *Proc. 38th Ann. Symp. on Foundations of Computer Science*, pp. 303-312, 1997.

[11] Y. Bartal, A. Fiat, and Y. Rabani, Competitive Algorithms for Distributed Data Management. In *Proc. 24th Ann. ACM Symp. on the Theory of Computing*, pp. 39-49, 1992.

[12] S. Ben-David and A. Borodin. A new measure for the study of on-line algorithms. *Algorithmica*, 11:73–91, 1994.

[13] D.L. Black and D.D. Sleator. Competitive Algorithms for Replication and Migration Problems. Technical Report CMU-CS-89-201, Department of Computer Science, Carnegie-Mellon University, 1989.

[14] A. Borodin, N. Linial, and M. Saks, An Optimal On-Line Algorithm for Metrical Task Systems. In *Proc. 19th Ann. ACM Symp on Theory of Computing*, pp. 373–382, May 1987.

[15] A. Chou, J. Cooperstock, R. El-Yaniv, M. Klugerman, and T. Leighton, The Statistical Adversary Allows Optimal Money-Making Trading Schems. In *Proc. 6th Ann. ACM-SIAM Symp. on Discrete Algorithms*, pp. 467–476, 1995.

[16] T.M. Cover. Universal Portfolios. *Mathematical Finance*, 1:1–29, 1991.

[17] X. Deng and C.H. Papadimitriou. Competitive Distributed Decision-Making. In *Proc. 12th IFIP Congress*, pp. 350–356, 1992.

[18] R. El-Yaniv, A. Fiat, R. Karp, and G. Turpin, Competitive Analysis of Financial Games. In *Proc. 33rd Ann. Symp. on Foundations of Computer Science*, pp. 327-333, 1992.

[19] R. El-Yaniv, A. Fiat, R. Karp, and G. Turpin, Optimal Search and One-Way Trading Online Algorithms. manuscript, 1997.

[20] A. Fiat, Y. Mansour, A. Rosén, and O. Waarts. Competitive Access Time via Dynamic Storage Rearrangement. In *Proc. 36th Ann. Symp. on Foundations of Computer Science*. pp. 392–403, 1995.

[21] R.L. Graham. Bounds for certain multi-processing anomalies. *Bell System Technical Journal*, 45: 1563–1581, 1966.

[22] M.M. Halldórsson and M. Szegedy. Lower bounds for on-line graph coloring. In *Proc. 3rd Ann. ACM-SIAM Symp. on Discrete Algorithms*, pp. 211–216, 1992.

[23] J.C. Hull. **Options, Futures, and other Derivative Securities: Second Edition**, Prentice-Hall, Inc, 1993.

[24] S. Irani. Coloring inductive graphs on-line. *Algorithmica*, 11:53–62, 1994.

[25] S. Irani and Y. Rabani. On the Value of Information in Coordination Games. In *Proc. 34th Ann. Symp. on Foundations of Computer Science*, pp. 12–21, 1993.

[26] R.M. Karp. On-line Algorithms Versus Off-line Algorithms: How Much is it Worth to Know the Future?. In *Proc. World Computer Congress*, 1992.

[27] R. Karp and P. Raghavan. From a personal communication cited in [35].

[28] E. Koutsoupias and C.H. Papadimitriou. Beyond competitive analysis. In In *Proc. 35th Ann. Symp. on Foundations of Computer Science*, pp. 394–400, 1994.

[29] M.S. Manasse, L.A. McGeoch, and D.D. Sleator, Competitive Algorithms for On-Line Problems. In *Proc. 20th Ann. ACM Symp. on Theory of Computing*, pp. 322–333, 1988.

[30] M. Mitzenmacher. How Useful is Old Information ? In *Proc. 16th Ann. ACM Symp. on Principles of Distributed Computing*, pp. 83–91, 1997.

[31] R. Mirchandaney, D. Towsley, and J. A. Stankovic, Analysis Effects of Delays on Load Sharing. *IEEE Transactions on Computers*, Vol. 38, pp. 1513–1525, 1989.

[32] C.H. Papadimitriou and M. Yannakakis. On the Value of Information in Distributed Decision Making. In *Proc. 25th ACM Symp. on Principles of Distributed Computing*, pp. 61–64, 1991.

[33] C.H. Papadimitriou and M. Yannakakis. Linear Programming Without the Matrix. In *Proc. 25th ACM Symp. on Theory of Computing*, pp. 121–129, 1993.

[34] P. Raghavan. A Statistical Adversary for On-Line Algorithms. *On-Line Algorithms* DIMACS Series in Discrete Mathematics and Theoretical Computer Science, pp. 79–83, 1991.

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

[36] D. Sleator and R.E. Tarjan, Amortized Efficiency of List Update and Paging Rules. *Communications of ACM*, 28(2):202–208, 1985.

[37] D. Towsley and R. Mirchandaney. The Effect of Communication Delays on the Performance of Load Balancing Policies in Distributed Systems. In *Proc. Second International MCPR Workshop*, pp. 213–226, 1988.

[38] A.C.-C. Yao. Probabilistic computations: Towards a unified measure of complexity. In *Proc. 17th Ann. Symp. on Foundations of Computer Science*, pages 222–227, 1977.

A Selected Proofs

Lemma 4 *Two jobs J_i and J_k in the same group are assigned to different machines.*
Proof: Without loss of generality, assume $i < k$. When scheduling J_i, the algorithm sees the schedule S_i that results after $i - d_i$ jobs have been assigned to machines and when scheduling J_k, the algorithm sees the schedule S_k that results after $k - d_k$ jobs have been assigned. As the earlier job J_i cannot see a more recent schedule than the later be the case that $i - d_i \leq k - d_k$.

Since J_i and J_k are in the same group (say g), $g = \lfloor f(J_i)/b \rfloor = \lfloor f(J_k)/b \rfloor$. Then J_i is assigned to the machine M_i of rank $b - (f(J_i) \bmod b) = b - (f(J_i) - g \cdot b) = (g+1)b - (2i - d_i)$ (in schedule S_i). Similarly, J_k is assigned to the machine M_k of rank $(g+1)b - (2k - d_k)$ in schedule S_k.

Now, schedule S_k results from schedule S_i by the scheduling of an additional $(k - d_k) - (i - d_i)$ jobs. Observe that a machine that has rank r in a certain schedule S has rank at least $r - i$ in the schedule obtained by placing i additional jobs in S. Thus, in schedule S_k, the machine M_i must have rank at least

$$(g+1)b - (2i - d_i) - ((k - d_k) - (i - d_i)) \geq (g+1)b - (k + i - d_k) > (g+1)b - (2k - d_k).$$

This implies that the machines M_i and M_k are distinct. ■

Lemma 5 *The competitive ratio of the Delayed Avoid Heavy algorithm is at most $2 + \frac{2d-2}{b} + c$.*
Proof: When job J_i arrives, we know the loads on all machines except for the contributions to the loads by the last $d_i - 1$ jobs. Let S be the set of the last $d_i - 1$ jobs together with job J_i. Observe that the f values of any two jobs in S can differ by at most $d - 1 + d_i - 1 \leq 2d - 2$. Thus the number of distinct groups that the jobs in S belong to is at most $2 + \lfloor \frac{2d-2}{b} \rfloor \leq 2 + \frac{2d-2}{b}$. Since no two jobs in the same group get placed on the same machine, the maximum number of jobs in S that get placed on the same machine is at most $2 + \frac{2d-2}{b}$, and in particular there are at most $1 + \frac{2d-2}{b}$ unknown jobs on the processor that gets J_i. Let w_i be the known load on the machine on which job J_i is placed. Let S be the total known load on all the machines. Then $w_i/S \leq c/m$. If not, then the loads on the heaviest m/c machines must each be greater than Sc/m, implying that the total load is greated than S. This is clearly not possible. Now, Lemma 1 implies that, after J_i is placed on M_i, the total load on M_i is at most $2 + (2d-2)/b + c$ times the optimal load. Hence the competitive ratio is at most $2 + \frac{2d-2}{b} + c$. ■

Theorem 7 *The algorithm MTF(d) is $(d+1)$-competitive. This upper bound holds for both types of adversaries.*
Proof: We prove the theorem for the standard adversary. For the analysis of the MTF(d) algorithm we consider a slightly different model for updating the list. In this modified model, an on-line algorithm may move an item accessed in a round only on the last request to the item in that round. We consider the algorithm MTF'(d) that moves an item to the front of the list whenever it is requested for the last time in a round. Given any request sequence, at the end of each round the list maintained by MTF'(d) is the same as the list maintained by MTF(d). Thus, in each round the cost incurred by MTF(d) is not higher than the cost incurred by MTF'(d). Therefore, it suffices to show that the cost incurred by MTF'(d) is at most $d+1$ times the cost incurred by the adversary, for any request sequence σ.

We assume that MTF'(d) and the adversary start with the same list. Let $\sigma = \sigma(1), \sigma(2), \ldots, \sigma(m)$ be an arbitrary request sequence and let t denote the point in time *after* the t-th request $\sigma(t)$ is served. We define a potential function Φ. For any time t and any item x in the list, let $r(t,x)$ be the next round in the request sequence in which x is requested. If x is still requested in the current round, then $r(t,x)$ is equal to the current round. Let $n(t,x)$ be the number of remaining requests to x in $r(t,x)$. We have $n(t,x) \leq d$. In *inversion* is an ordered pair (y,x) of items such that x occurs before y in the adversary's list and after y in the list maintained by MTF'(d). At any time the potential Φ is the number of inversions (y, x), where each inversion is multiplied by $n(t, x)$. The value $n(t, x)$ can be seen as the weight of an inversion (y, x).

Consider any request $\sigma(t)$ and suppose that item x is requested by $\sigma(t)$. Let $C_{MTF}(t)$ and $C_{ADV}(t)$ be the actual costs paid by MTF'(d) and the adversary during the service of $\sigma(t)$. Clearly, $C_{MTF}(t) \leq C_{ADV}(t) + inv(t-1, x)$, where $inv(t-1, x)$ is the number of inversions (y, x) immediately before the request. We show that during the service of $\sigma(t)$ the potential decreases by $inv(t-1, x)$ due to inversions removed or due to inversions whose weights change. If x is not requested for the last time in the round, then the number of remaining requests to x in the round decreases by 1, i.e., $n(t-1, x) - n(t, x) = 1$ and the weight of each inversion (y, x) decreases by 1. If x is requested for the last time in the round, $n(t, x)$ can increase, i.e., $n(t, x) \geq n(t, x)$. However, x is moved to the front of the list, which implies that all inversion (y, x) are removed and $n(t, x)$ does not contribute to the potential. In any case, the potential decreases by $inv(t-1, x)$ during the service of $\sigma(t)$. If x is moved to the front of the list, then at most $C_{ADV}(t)$ new inversions (x, z) can be created, each of which increases the potential by $n(t, z) \leq d$. Since $n(t-1, y) = n(t, y)$ for all $y \neq x$, we conclude that at any time t,

$$C_{MTF}(t) + \Delta\Phi \leq C_{ADV}(t) + d \cdot C_{ADV}(t) \leq (d+1)C_{ADV}(t).$$

Finally we have to consider a paid exchange made by the adversary. Each paid exchange can create an inversion, which increase the potential by at most d, but the adversary has to pay a cost of 1. So again

$$C_{MTF}(t) + \Delta\Phi \leq (d+1)C_{ADV}(t).$$

Summing over all the steps of σ and noting $\Phi \geq 0$ yields $C_{MTF}(\sigma) \leq (d+1)C_{ADV}(\sigma)$. ■

Theorem 9 *Let A be a randomized on-line algorithm for the list update problem with delayed action and suppose that A does not use paid exchanges. If A is c-competitive against any oblivious adversary, then $c \geq d$. This lower bound holds for both types of adversaries.*

Proof: An adversary constructs a request sequence in *phases*. In each phase the adversary inspects its current list and requests the n items in ascending order. To each of the n items, the adversary issues d consecutive requests, which form a round. In each phase the adversary incurs a cost of $\sum_{i=1}^{n}(i+d-1) = n(n+1)/2 + n(d-1)$. Since the on-line algorithm can only moves items only after they have been requested, its cost in a phase is at least $\sum_{i=1}^{n} di = dn(n+1)/2$. ■

Theorem 10 *Let A be a randomized on-line algorithm for the list update problem with delayed action and suppose that A does use paid exchanges. If A is c-competitive against any oblivious adversary, then $c \geq d/2$. This lower bound holds for both types of adversaries.*

Proof: We give a probability distribution on request sequences such that the expected cost incurred by any deterministic on-line algorithm is at least $d/2$ times the expected cost incurred by an adversary. The result then follows from Yao's minimax principle [38]. The request sequence is constructed as follows. In each round one of the n items is chosen uniformly at random; this item is requested d times. The expected cost incurred by a deterministic on-line algorithm in a round is $dn/2$ whereas the adversary's cost no more than $n + d - 1$. ■

Theorem 13 *There exist money-making schemes for $m = 1$, regardless of n and d.*

Proof: Let ϵ_i be the investment on the ith day. We may set $\epsilon_i = 0$ at any point after the player sees a result which is an up move. It will also be convenient notationally if we define $\epsilon_i = 0$ for $i \geq n$. If the up move is on day j, then the total return to the player will be

$$(\epsilon_j \alpha + 1 - \epsilon_j) \prod_{i \neq j, i \leq j+d} (\frac{\epsilon_i}{\alpha} + 1 - \epsilon_i).$$

Note that

$$(\frac{\epsilon_a}{\alpha}+1-\epsilon_a)(\frac{\epsilon_b}{\alpha}+1-\epsilon_b) \approx (\frac{\epsilon_a+\epsilon_b}{\alpha}+1-\epsilon_a-\epsilon_b).$$

Also,

$$(\epsilon_a\alpha+1-\epsilon_a)(\frac{\epsilon_b}{\alpha}+1-\epsilon_b) > 1 \textbf{ if } \epsilon_a > \frac{\epsilon_b}{\alpha(1-\epsilon_b)}.$$

Hence, the condition

$$(\epsilon_j\alpha+1-\epsilon_j)\prod_{i\neq j, i\leq j+d}(\frac{\epsilon_i}{\alpha}+1-\epsilon_i) > 1$$

is satisfied if

$$\epsilon_j > \frac{\sum_{i\neq j,i\leq j+d}\epsilon_i}{\alpha(1-\sum_{i\neq j,i\leq j+d}\epsilon_i)}.$$

This condition is easily satisfied by choosing the initial ϵ_i to be suitably small and having the ϵ_i grow geometrically at a suitably small rate (say, less than $\alpha^{1/d}$).

■

B Results on delayed relaxed task systems

Our results generalize the algorithms of [4, 9]; in fact, when $d = 0$, our arguments reduce to theirs.

B.1 Randomized Algorithm

Let A be a c competitive algorithm for $\mathcal{P}$, and let $D \geq 1/2$. We give a randomized algorithm D-Alg that is competitive against adaptive on-line adversaries for D-$\mathcal{P}$ in the delayed model. The algorithm is exactly the same as the algorithm in [4] for relaxed task systems.

Algorithm **D-Alg**.
Algorithm D-Alg simulates a version of algorithm A. At all times, the configuration of D-Alg is equal to that of the simulated version of A.

Upon receiving a request r, with probability $\frac{1}{2D}$, feed A with new request r, and change the configuration to the new configuration of A. With probability $1-\frac{1}{2D}$, the algorithm stays in the same configuration.

Theorem 16 *Let $\mathcal{P}$ be a metrical task system, and let A be c-competitive for $\mathcal{P}$ against adaptive on-line adversaries. Algorithm D-Alg is $(3+\frac{d-1}{D})c$ competitive for D-$\mathcal{P}$ with delay d, against adaptive on-line adversaries, for $D \geq 1/2$.*

B.2 Deterministic Algorithm

For any deterministic algorithm A, request sequence σ and request r, let $\text{cost}_A(\sigma, r)$ (or $\text{cost}_A(r)$ when σ follows from the context) be the cost incurred by A while servicing r from the configuration reached by previously servicing σ. Also, let $\text{cost}_A(\sigma)$ be the total cost of A on σ. Assuming that A is c-competitive for $\mathcal{P}$, we define the competitive algorithm D-DAlg for D-$\mathcal{P}$ as follows. (The algorithm is a modification of the algorithm D-DAlg in [9] for relaxed task systems.)

Algorithm **D-DAlg**.
Algorithm D-DAlg simulates $2D$ copies $A_1 \dots A_{2D}$ of A. Let $\beta = 2+\sqrt{1+\frac{d}{D}}$. The configuration of D-DAlg is always the same as that of A_1. When given a new request r, the algorithm gives it to one of the A_i according to the following rule:

- if there exists $i \geq 2$ such that $\mathrm{cost}_{A_i}(r) \geq \frac{1}{\beta c}\mathrm{cost}_{A_1}(r)$, r is given to A_i (i.e. the simulated configuration of A_i is updated). Then D-DAlg services r remotely, without changing its configuration.
- otherwise, r is given to A_1. Then D-DAlg services r and moves to the new configuration of A_1.

Theorem 17 *Let $\mathcal{P}$ be a metrical task system and let A be a c-competitive deterministic algorithm for $\mathcal{P}$. Then algorithm D-*DAlg *is $\beta^2 c^2$-competitive for the D-relaxed task system D-$\mathcal{P}$.*

We also note that, similar to the results of [9], we can also get slightly better competitive ratios for monotonic task systems as well as randomized algorithms against oblivious adversaries. A more complete discussion will appear in the full paper.