INTERNATIONAL COMPUTER SCIENCE INSTITUTE

1947 Center Street • Suite 600 • Berkeley, California 94704 • 1-510-642-4274 • FAX 1-510-643-7684

On-Line Algorithms Versus Off-Line Algorithms: How Much is it Worth to Know the Future?

Richard M. Karp*

TR-92-044

July 1992

Abstract

An *on-line algorithm* is one that receives a sequence of requests and performs an immediate action in response to each request. On-line algorithms arise in any situation where decisions must be made and resources allocated without knowledge of the future. The effectiveness of an on-line algorithm may be measured by its *competitive ratio*, defined as the worst-case ratio between its cost and that of a hypothetical off-line algorithm which knows the entire sequence of requests in advance and chooses its actions optimally. In a variety of settings, we discuss techniques for proving upper and lower bounds on the competitive ratios achievable by on-line algorithms. In particular, we discuss the advantages of randomized on-line algorithms over deterministic ones.

*Presented at the World Computer Congress in Madrid, Spain, September 1992.

1 Introduction

1.1 On-Line Algorithms Versus Off-Line Algorithms

An *on-line algorithm* is one that receives a sequence of *requests* and, performs an immediate action in response to each request. Each sequence of requests and corresponding actions has an associated *cost.*

On-line algorithms arise in many different settings. For example, in the case of on-line bin packing each request is an object to be packed, and the corresponding action is to pack the object in some bin. In the context of a database system each request may be a query or an update, and the corresponding action involves retrieving data from, and possibly modifying, the database. In an investment situation, a request might consist of a price quotation for a commodity, and the action might be to buy or sell some amount of the commodity.

On-line algorithms may be contrasted with *off-line algorithms*, which receive the entire sequence of requests in advance; an off-line algorithm is required to take an action in response to each request, but the choice of each action can be based on the entire sequence of requests. In other words, an off-line algorithm knows the future, but an on-line algorithm does not. It is clear that, in many settings, ignorance of the future is a great disadvantage (think of stock market investment, for example). Thus, on-line algorithms will often perform much worse than their off-line counterparts.

It is easy to say what is meant by an optimal off-line algorithm; for each sequence of requests, such an algorithm selects that sequence of actions which minimizes the cost. Defining a measure of performance for on-line algorithms is more problematic, since it is usually the case that, whatever actions an on-line algorithm takes in response to an initial sequence of requests, there will be a sequence of further requests that makes the algorithm look foolish. It is perhaps because of this difficulty in defining "rational" or "optimal" performance that on-line algorithms have been relatively neglected in fields like computer science, operations research and economics. For example, scheduling theory has focused on off-line models, which assume, often inappropriately, that the entire stream of jobs arriving at a service facility is known in advance.

The most common approach to evaluating on-line algorithms is to assume a specific stochastic model of the source of requests. Within such a model, an on-line algorithm may be considered optimal if it chooses its actions so as to minimize the expected cost, where the cost depends on the sequence of requests generated by the stochastic source and on the sequence of actions chosen by the algorithm in response to these requests. Much of the theory of stochastic games, stochastic control, stochastic scheduling, performance analysis of computer systems and probabilistic analysis of algorithms is based on this approach. However, the choice of a stochastic model requires data that may not be readily available about the request sequences that have been observed in the past, as well as faith that the future will resemble the past.

1.2 The Competitive Ratio

This paper is concerned with an alternative to stochastic models - a worst-case approach in which an on-line algorithm is evaluated by comparing its cost with that of an optimal off-

line algorithm processing the same sequence of requests. We follow [ST], which defines the *competitive ratio* of an on-line algorithm as the maximum, over all possible input sequences, of the ratio between the cost incurred by the on-line algorithm and the cost incurred by an optimal off-line algorithm. An optimal on-line algorithm is then one whose competitive ratio is least. The concept of competitive ratio is related to the minimax regret concept in game theory, and we shall often view the situation as a game between an *on-line player* who selects the on-line algorithm and an *adversary* who chooses the request sequence to maximize the ratio between the algorithm's cost and that of an optimal off-line algorithm.

One virtue of the competitive ratio approach is that it avoids commitment to a particular stochastic model. However, the approach is a pessimistic one, since it assumes, in effect, that the request sequence will be chosen by an all-knowing adversary. Ideally, what we would probably like in practice is an on-line algorithm that performs well on typical request sequences and also has a small competitive ratio, so that its behavior can never be too catastrophic.

Let us define the setting more rigorously. An *on-line problem* is specified by:

- A set R of *requests*;
- A set A of *actions*;
- A *cost function* $c : \bigcup_{n=1,2,\ldots} R^n \times A^n \rightarrow \Re^+$, where $\Re^+$ denotes the nonnegative reals.

For any request sequence $r \in R^n$, define $Opt(r)$ as $\min_{a \in A^n} c(r, a)$.

An *on-line algorithm* $\mathcal{A}$ is determined by a function $f_{\mathcal{A}} : R^+ \rightarrow A$, where R^+ is the set of all finite nonempty sequences of requests. In response to a sequence of requests $r = r_1, r_2, \ldots, r_t$ the algorithm performs the sequence of actions $\mathcal{A}(r) = f_{\mathcal{A}}(r_1), f_{\mathcal{A}}(r_1 r_2), \ldots, f_{\mathcal{A}}(r_1 r_2 \ldots r_t)$ and incurs the cost $c(r, \mathcal{A}(r))$. Note that, by this definition, an on-line algorithm is deterministic. In Section 4 we will extend the definition to encompass randomized on-line algorithms.

For any positive constant d, the on-line algorithm $\mathcal{A}$ is said to be *d-competitive* if there exists a constant b such that, for all request sequences r, $c(r, \mathcal{A}(r)) \leq dOpt(r) + b$. The *competitive ratio* of $\mathcal{A}$ is defined as the greatest lower bound of the set of c such that $\mathcal{A}$ is c-competitive. Some authors use a variant of these definitions, in which b is required to be zero.

In the following sections we shall explore what is known about the construction of on-line algorithms with an optimal competitive ratio.

2 Some Examples

2.1 The Ski Rental Problem

To illustrate the concept of competitive ratio L. Rudolph introduced the ski rental problem. Suppose the cost of renting a pair of skis for a ski trip is 1, and the cost of buying a pair of skis is s. Not knowing how many ski trips we will take, how do we decide whether to rent or buy? Here there is only one possible request ("Take ski trip") and three possible actions ("rent", "buy" and "use skis already bought"), with costs 1, t and 0, respectively, where the third action can be invoked only if the second action has occurred previously. Clearly, any sensible on-line algorithm is of the form "rent for the first k trips, then buy, then use the skis already bought." On a sequence of t requests, the cost incurred by the algorithm

is t if $t \leq k$ and $k + s$ otherwise, and the cost incurred by an optimal off-line algorithm is $\min(s, t)$.

The problem is to choose the parameter k to minimize the competitive ratio. For a given k, a request sequence of length $k+1$ maximizes the ratio between the on-line algorithm's cost and that of an optimal off-line algorithm. In other words, the adversary should continue the ski trips until the on-line algorithm buys a pair of skis, and then stop. The on-line algorithm's cost on such a sequence is $k + s$, the optimal off-line cost is $\min(k + 1, s)$, and the competitive ratio is $\frac{k+s}{\min(k+1,s)}$. Assuming that s is an integer, the competitive ratio is minimized by setting $k = s - 1$, thus achieving a competitive ratio of $\frac{2s-1}{s}$. One way to view this is that the on-line player should rent until enough ski trips have occurred so that, by hindsight, he would have done better to buy skis initially.

2.2 Paging

Consider a computer system which has a fast memory with capacity for k pages of data and a slow memory with unlimited capacity. A set of n pages is to be kept in storage at all times, where $n > k$; at any given time, k pages will reside in the fast memory, and the other $n - k$ will reside in the slow memory. When a program requests access to a page that lies in the slow memory a *page fault* is said to occur; a vacancy must then be created by evicting some page from the fast memory, and the requested page must be brought into the vacant position.

The goal of an on-line paging algorithm is to minimize the number of page faults that occur; on-line algorithms differ in the choice of a page to evict when a page fault occurs. In terms of our formalism, there are n possible requests, corresponding to the n pages in the system, n actions of the form "evict page i," and a "no op" action. The cost function is designed to charge 1 for each page fault; in addition, invalid actions such as evicting a page that does not lie in the fast memory or responding to a page fault with a no op action incur an infinite cost.

Following [ST] we shall show that no deterministic algorithm can have competitive ratio less than k. To do so we exhibit an *adversary strategy*; i.e., a rule which, given any deterministic on-line paging algorithm $\mathcal{A}$, constructs arbitrarily long "bad" request sequences. In the present case the adversary strategy is obtained by restricting attention to a set S of $k + 1$ pages and producing an infinite request sequence which causes $\mathcal{A}$ to incur a page fault after every request. The sequence is obtained by simulating $\mathcal{A}$ and, at each step, requesting a page in S that does not lie in fast storage. On the other hand, for any sequence of requests drawn from S, an off-line algorithm can use its knowledge of the future to ensure that at least k requests occur between any two consecutive page faults; whenever a page fault occurs, the off-line algorithm evicts that page in fast storage for which the next request is furthest off in the future. It follows that $\mathcal{A}$ does not achieve a competitive ratio less than k.

It is easy to construct an on-line paging algorithm that is k-competitive. We challenge the reader to do so.

2.3 List Processing

The paper [ST] considers the problem of maintaining a data structure consisting of a linear list of n items. Each request is of the form "Access x, " where x is an item. The cost of accessing a given item is equal to its position in the current list. When the item in position

i is requested there are i possible actions, corresponding to moving the item forward any number of positions between 0 and $i-1$; there is no extra cost for this forward motion.

Sleator and Tarjan ([ST]) prove that *Move-to-Front* - the algorithm that moves each item to the front of the list whenever it is accessed - is 2-competitive. The proof is instructive because it involves the important concept of a *potential function*. Imagine that, as requests are received, the Move-to-Front algorithm and Opt, the optimal off-line algorithm (aided by its knowledge of future requests), process the requests and maintain their lists of the n items. At any step, let p be Move-to-Front's list and let q be Opt's list. At a step in which the two lists coincide any access request will cause the two algorithms to incur the same cost. Thus, Opt has the potential to outperform Move-to-Front only when the lists differ. This observation suggests defining a *potential function* $\Phi(p,q)$ which measures the extent to which the two lists differ, and hence the potential for Opt to outperform Move-to-Front. In this case it is best to define $\Phi(p,q)$ as the number of adjacent interchanges of items needed to pass from the list p to the list q. It is then easy to show that, at each step, $c_{on} + \Delta\Phi \leq 2c_{off}$, where c_{on} and c_{off}, respectively, denote the cost incurred by Move-to-Front and by Opt at the step, and $\Delta\Phi$ denotes the change in potential at the step. It then follows by simple algebra, from the fact that Φ is nonnegative and initially zero, that for any request sequence the cost incurred by Move-to-Front is at most twice that incurred by Opt.

It is also possible to prove that no deterministic on-line algorithm can achieve a competitive ratio less than 2. The reader is invited to provide the required adversary argument.

The field of dynamic data structures provides a great deal of scope for the study of on-line algorithms. In particular, the paper [ST1] introduces the *splay-tree algorithm* for executing sequences of insertions, deletions and accesses to a binary search tree. Assuming that each operation of following or changing a pointer has unit cost, Sleator and Tarjan conjecture that the splay-tree algorithm has a bounded competitive ratio, and prove the conjecture in some interesting special cases.

2.4 Multiprocessor Scheduling

In the *multiprocessor scheduling problem* each request is a task of a given size, and each action is to assign the current task to one of m processors. The cost of a sequence of requests and actions is the *makespan*, defined as the maximum, over all processors, of the sum of the sizes of the tasks assigned to the processor. In [Gra], Graham considers a natural on-line algorithm called Low. At each step, Low assigns the current task to the most lightly loaded processor; i.e., the one for which the sum of the sizes of the assigned tasks is least. Graham shows that the competitive ratio of Low is $2 - \frac{1}{m}$. A worst-case request sequence consists of $m(m-1)$ tasks of size 1 followed by a task of size m; for this example Low achieves a makespan of $2m-1$, and the optimal makespan is m.

For $m = 2$ and $m = 3$ Low achieves the smallest competitive ratio of any on-line algorithm. For $m = 3$ the competitive ratio of Low is $5/3$, and the following simple case analysis shows that no on-line algorithm can beat $5/3$. Suppose the first three tasks to arrive are of size 1. They must be packed on distinct processors; if not, the sequence could end and the algorithm would be paying twice the optimal cost. If the next three tasks are of size 3 then, again, they must be assigned to distinct processors; if not, the sequence could end and the algorithm would be paying $7/4$ of the optimal makespan. Thus, after the arrival of the first six tasks, the sum of the sizes of the tasks assigned to each processor is 4. If the next task is of size 6 then, however it is assigned, the makespan is 10. On the other

hand, one can easily see that the optimal makespan for all seven tasks is 6. This completes the proof that the competitive ratio of an on-line algorithm for $m = 3$ is at least $5/3$.

The worst case for Low occurs when all processors are equally loaded and a large task arrives. Intuitively, it seems that a better competitive ratio can be achieved by avoiding the case of equal loading and, instead, keeping the loads on the processors somewhat staggered. Pursuing this idea, [BFKV] gives an algorithm whose competitive ratio is bounded above by a universal constant less than 2, for all m; thus, when m is sufficiently large, this algorithm has a better competitive ratio than Low.

2.5 Interval Coloring

As a final introductory example we consider the *interval coloring problem*, in which each request is an interval on the real line and each action assigns a *color* to the current request. No two overlapping intervals may receive the same color. The cost of a sequence of requests is the number of colors used. This problem can be interpreted as a scheduling problem, in which each interval represents the time span of some task, and the color represents the processor assigned to execute the task. Kierstead and Trotter [KT] give an on-line algorithm with optimal competitive ratio for this problem. As each interval I arrives it is assigned a positive integer $h(I)$ called its *height* and a color as follows:

- If I does not intersect any previous interval of height 1 then $h(I)$ is set equal to 1; otherwise, $h(I)$ is set equal to the least $j > 1$ such that I does not intersect more than two previous intervals of height j.
- One color is reserved for intervals of height 1, and, for each $j > 1$, three colors are reserved for each height j; interval I is assigned any color, among those reserved for its height, that has not been assigned to any previous interval that intersects I.

If k is the maximum height assigned to any interval then:

- The Kierstead-Trotter algorithm uses at most $3k - 2$ colors;
- There is a set of k mutually intersecting intervals; thus, any algorithm requires at least k colors.

It follows that, if the optimal number of colors is k, then the Kierstead-Trotter on-line algorithm will require at most $3k - 2$ colors. It can be shown by means of an adversary argument that, for any online algorithm, and any k, there exists an instance for which the optimal number of colors required is k but the algorithm requires $3k - 2$ colors. Thus, the Kierstead-Trotter algorithm is an optimal on-line algorithm.

3 The k-Server Problem

In this section we discuss the most thoroughly studied of all on-line problems - the famous k-server problem of Manasse, McGeoch and Sleator([MMS]). The problem is set in a metric space; i.e., a set of points M together with a distance function d, defined for each pair x, y of points in M, and satisfying

- $d(x, y) = d(y, x)$ (symmetry) and
- $d(x, z) \leq d(x, y) + d(y, z)$ (triangle inequality).

The execution of an on-line algorithm for the k-server problem begins with k *servers* located at given points in the metric space. Each request is a point in the space, and each action causes one of the k servers to move to the most recent request point. The cost of the action is the distance moved by the server. A server remains stationary unless it is selected to move to a request point. The paging problem with n pages and a fast memory of size k can be viewed as the special case of the k-server problem in which the metric space has n points and the distance between any two distinct points is 1.

Many natural algorithms for the k-server problem fail to achieve a bounded competitive ratio. For example, consider the following *greedy algorithm*: "Answer each request by moving the closest server." In any metric space where arbitrarily small positive distances occur, the greedy algorithm can be defeated by placing requests alternately at two points that are sufficiently close together. The greedy algorithm will build up an unbounded cost by shuttling the same server back and forth forever, but, on the same request sequence, an optimal off-line algorithm will keep its cost bounded by stationing a server permanently at each of the two points.

3.1 The Manasse-McGeoch-Sleator Lower Bound

In [MMS] it is shown that, in any metric space with at least $k+1$ points, no on-line algorithm for the k-server problem has competitive ratio less than k. The proof is by an adversary argument that generalizes the one given above for the paging problem. One can assume without loss of generality that the k servers are initially located at distinct points. The adversary restricts attention to the k initial server locations plus one additional point. The adversary then constructs an infinite request sequence by requesting, at each step, the unique point among these $k+1$ that the on-line algorithm $\mathcal{A}$ does not have covered by a server. If the successive requests occur at $r_1, r_2, \ldots, r_t, \ldots$ then $\mathcal{A}$ incurs cost $d(r_1, r_2) + d(r_2, r_3) + \ldots + d(r_t, r_{t+1})$ in processing the first t requests. The rest of the proof consists of showing that, if an off-line algorithm is free to choose the initial positions of its servers, then it can restrict its cost for processing these same t requests to at most $\frac{d(r_1,r_2)+d(r_2,r_3)+\ldots+d(r_t,r_{t+1})}{k}$. Since the initial positions of the servers only affect the cost of an optimal off-line algorithm by a constant, it follows that $\mathcal{A}$ does not achieve a competitive ratio less than k.

In [MMS] Manasse, McGeoch and Sleator conjecture that this lower bound is tight; i.e., that for every metric space M, and every k, there is a k-competitive on-line algorithm for the k-server problem. At present we are very far from proving or disproving this conjecture. As we shall discuss in Section 4.4, a much weaker related conjecture has been proven: for every k, and every metric space, there is a $(k(\frac{5}{4}2^k - 2))^2$-competitive on-line algorithm for the k-server problem; this result shows that a bounded competitive ratio is always achievable.

3.2 The k-Server Problem on a Tree

The Manasse-McGeoch-Sleator conjecture has been proven for $k = 2$, for the k-server problem in any metric space with $k+1$ points, and for certain special metric spaces. For example, [CL1] gives a k-competitive on-line algorithm for the k-server problem on a treelike road network. By "treelike road network" we mean a topological tree in which every arc is a Jordan curve, the metric space is the set of points in the union of these arcs, and the distance between two points is the length of the unique path joining them. At a typical step the k servers are located at points of the road network and a request occurs at some point x. All servers begin moving toward x at the same rate. Each server s continues moving

along the path from itself to x until it becomes blocked by some other server; i.e., until some other server occupies the closed path between x and the current position of s. A tie, in which two servers occupy the same point at the same time, is broken arbitrarily. This algorithm technically violates the rules of the game, since more than one server moves at a time. However, the algorithm is easily converted to one in which only one server moves at a time and the total distance traveled by servers is no greater than in the original algorithm. The idea is that, since only one server actually reaches the request point, each of the other servers can simply calculate the "virtual motion" that it would have carried out under the original algorithm, and execute an actual move only when a virtual motion brings it to the current request point.

To prove that their algorithm is k-competitive Chrobak and Larmore devise a potential function $\Phi(p, q)$, where p is a k-tuple of points giving the positions of the on-line algorithm's servers, and q is a k-tuple of points giving the positions of the off-line algorithm's servers. The function Φ has the following properties:

- At each step of the off-line algorithm, Φ increases by at most k times the cost of the step;
- At each step of the on-line algorithm, Φ decreases by at least the cost of the step.

These two properties, together with the fact that Φ is nonnegative and initially zero, prove that, on any given request sequence, the cost incurred by the on-line algorithm is at most k times the cost incurred by the off-line algorithm.

4 The Power of Randomization

Adversary constructions are the principal means of proving lower bounds on the competitive ratio achievable for a given problem. To prove that the competitive ratio C is not achievable such a construction takes as input an arbitrary deterministic on-line algorithm $\mathcal{A}$ for the problem and produces an infinite family $\mathcal{R}$ of request sequences such that, as the length of the sequence $r \in \mathcal{R}$ increases, the ratio $\frac{c(r,\mathcal{A}(r))}{Opt(r)}$ eventually exceeds every number less than C. The construction usually depends on the ability to simulate $\mathcal{A}$. This suggests consideration of randomized on-line algorithms, which toss coins in the course of their execution. It seems that the unpredictability of such randomized algorithms should make it more difficult for an adversary to construct bad request sequences. In this section we consider the question of whether randomized on-line algorithms are more powerful than deterministic ones.

We may view a randomized algorithm as playing a game against a deterministic adversary. In each play of the game the adversary chooses the request sequence $\underline{r} = r_1, r_2, \ldots, r_n$ and its own sequence of actions $\underline{b} = b_1, b_2, \ldots, b_n$, and the on-line algorithm chooses its sequence of actions $\underline{a} = a_1, a_2, \ldots, a_n$. Since the on-line algorithm is randomized the sequence of actions it chooses may vary, even when the adversary's behavior remains fixed. For a positive constant C, the on-line algorithm is said to be C$-$*competitive* if, for every adversary,

$$E[c(\underline{r}, \underline{a}) - Cc(\underline{r}, \underline{b})]$$

is bounded above by a constant.

4.1 Types of Adversaries

To complete our definition of the game between the randomized on-line algorithm and the adversary we must specify the information available to each player when it makes each of

its decisions. It is always assumed that the adversary knows the text of the randomized on-line algorithm; however,the adversary does not have access to the algorithm's random coin tosses. Three different types of adversaries are indicated schematically below:

- *The Oblivious Adversary*
  $r_1(b_1)(a_1)r_2(b_2)(a_2)\ldots$

- *The Adaptive On-Line Adversary*
  $r_1(b_1)a_1r_2(b_2)a_2\ldots$

- *The Adaptive Off-Line Adversary*
  $r_1a_1r_2a_2\ldots r_na_nb_1b_2\ldots b_n$

Here the left-to-right sequence indicates the time sequence of the requests and actions, and parentheses indicate actions that are kept secret. Thus, the advantage of the adaptive on-line adversary over the oblivious adversary is that it gets to observe the actions of the on-line algorithm. The adaptive off-line adversary has the further advantage that it gets to observe all the algorithm's actions before choosing any of its own actions.

Against deterministic on-line algorithms all three adversary types are of equivalent power; there is no advantage to observing the on-line algorithm's actions, since the adversary can compute these actions by simulating the algorithm. Against randomized on-line algorithms the adaptive off-line adversary type is clearly the most powerful, and the oblivious adversary type is the least powerful. The competitive ratio that is achievable will depend on the adversary type considered.

The paper [BBKTW] proves the following theorems:

Theorem 1 *If there is a C-competitive randomized algorithm against adaptive off-line adversaries, then there is a C-competitive deterministic algorithm; in other words, there is no advantage to randomizing when playing against adaptive off-line adversaries.*

Theorem 2 *If there is a C-competitive randomized algorithm against oblivious adversaries and a D-competitive randomized algorithm against adaptive on-line adversaries, then there is a CD-competitive deterministic algorithm.*

It follows that the best competitive ratio achievable by a randomized algorithm against adaptive on-line adversaries is at least the square root of the best competitive ratio achievable by a deterministic algorithm.

4.2 Example: The Paging Problem

Recall that, for the paging problem, the best competitive ratio achievable by a deterministic on-line algorithm is k. It can be shown ([FKLMSY], [MS]) that the best competitive ratio achievable against oblivious adversaries is $H_k = 1 + 1/2 + \ldots + 1/k$, whereas, against adaptive on-line or off-line adversaries, the best competitive ratio is k. Thus, for this problem, randomized algorithms are more powerful than deterministic ones against oblivious adversaries, but not against adaptive adversaries.

4.3 Example: Bipartite Matching

A *matching* in a graph is a set of edges, no two of which meet at a common vertex. A bipartite graph is one in which the vertex set is the union of two disjoint sets, often referred to as the boys and the girls, such that each edge joins a boy with a girl. We consider an on-line version of the problem of constructing a large matching in a bipartite graph. In this version the boys arrive one-by-one. As each boy arrives the algorithm is told which edges are incident with the boy, and chooses at most one of these edges to be included in the matching; of course, the algorithm is not permitted to choose two edges incident with the same girl.

We must change the definition of competitive ratio to account for the fact that we are dealing with a maximization problem rather than a minimization problem. Let us say that a randomized algorithm is C-competitive against a type of adversary if, for every adversary of that type, the expected size of the matching produced by the algorithm, minus C times the expected size of the matching produced by the adversary, is bounded below by a constant; the competitive ratio of an algorithm is the least upper bound of the C's for which it is C-competitive.

The best competitive ratio achievable by a deterministic algorithm for this problem is $1/2$. An adversary can hold any deterministic algorithm to $1/2$ by repeating the following two-step cycle: present a boy who is adjacent to two girls; whichever girl the algorithm chooses, present a second boy who is adjacent to that girl, but not to the other one. This will limit the algorithm to matching only half the boys, even though there exists a matching that covers all the boys.A more complicated argument shows that the best competitive ratio achievable by a randomized on-line algorithm against adaptive on-line adversaries is also $1/2$.

It is shown in [KVV] that the best competitive ratio achievable by a randomized on-line algorithm against oblivious adversaries is $1-1/e$, where e is the base of natural logarithms. The algorithm that achieves this ratio is simple to state, but not so easy to analyze: choose a random ordering of the girls and, as each boy arrives, match him with the first girl, if any, who is incident with him and has not been chosen before. The adversary strategy that limits every randomized on-line algorithm to a competitive ratio of $1-1/e$ is also simple: let there be n boys and n girls; choose a random ordering $g_1, g_2, \ldots, g_n$ of the girls, and make the first boy adjacent to all the girls, the second boy, to $g_1, g_2, \ldots, g_{n-1}$ and, in general, make the ith boy adjacent to $g_1, g_2, \ldots, g_{n-i+1}$. In other words, the girls drop out in a random order.

This example illustrates what seems to be a rather general phenomenon: randomization helps considerably against oblivious adversaries, but not against adaptive adversaries. A good exercise for the interested reader is to confirm that this phenomenon also holds for the ski rental problem and the list processing problem presented in Section 2.

4.4 The Harmonic Algorithm for the k-Server Problem

It seems that a good on-line algorithm for the k-server problem should tend to favor servers that are close to the request point. On the other hand, the simple greedy algorithm, which always chooses the closest server, is easily foiled by an adversary because of its predictability, and fails to achieve a bounded competitive ratio. In this section we discuss a randomized on-line algorithm called the *Harmonic Algorithm* which behaves in a way that is intuitively reasonable, and yet contains enough randomness to present difficulty to an adversary. The Harmonic Algorithm is defined by the property that, each step, the probability that a given

server is the one to move is inversely proportional to that server's distance from the request point.

Call an adaptive on-line adversary *lazy* if, whenever possible, it requests a point that is occupied by one of its servers but not occupied by one of the on-line algorithm's servers. The behavior of an on-line algorithm against a lazy adversary is relatively simple since, at any given step, there is at most one *vacancy* that is occupied by one of the adversary's servers but not by one of the algorithm's servers. Thus, one can analyze the movement of a single vacancy rather than many servers.

It is possible to show that, in any metric space, the Harmonic Algorithm is $\frac{k(k+1)}{2}$-competitive against all lazy adversaries. Since lazy behavior on the part of an adversary seems reasonable, this leads to the conjecture that, in any metric space, the Harmonic Algorithm is $\frac{k(k+1)}{2}$-competitive against all adaptive on-line adversaries. However, the best result proven so far is that the Harmonic Algorithm is $k(\frac{5}{4}2^k - 2)$-competitive against adaptive on-line adversaries ([Gro]). It follows from Theorem 2 that, in any metric space, there exists a $(k(\frac{5}{4}2^k - 2))^2$-competitive deterministic on-line algorithm.

To prove his upper bound on the competitive ratio of the Harmonic Algorithm Grove gives an ingenious construction of a potential function $\Phi(p, q)$, where p is the configuration of the algorithm's servers and q is the configuration of the adversary's servers, such that:

- At any step in which the adversary incurs a cost of D, Φ increases by at most $Dk(\frac{5}{4}2^k - 2)$;
- At any step of the on-line algorithm, the expected decrease in Φ is greater than or equal to the expected cost of the step.

A further reason for interest in the Harmonic Algorithm stems from the work of Coppersmith, Doyle, Raghavan and Snir on resistive metric spaces. A metric space is called *resistive* if there exists an electric network with a node for each point in the space, such that the distance between any two points is equal to the resistance across the corresponding node pair in the network. In [CDRS] it is shown that, in any resistive metric space, a variant of the Harmonic Algorithm achieves a competitive ratio of k against all adaptive on-line adversaries. It follows that, in any resistive metric space, there is a k^2-competitive deterministic on-line algorithm.

5 Conclusion

On-line algorithms arise in any situation where decisions must be made and resources allocated with incomplete knowledge of the future. Typical settings include robot motion planning, investment analysis, bin packing, storage allocation, cache management, file migration, scheduling, graph coloring, maintenance of data structures and databases, facilities location and capacity expansion of networks. In all of these areas, interesting mathematical arguments have yielded lower and upper bounds on the competitive ratios that can be achieved. Lower bound proof usually depend on adversary constructions, and upper bounds are usually obtained by analyzing algorithms with the help of potential functions.

In connection with each particular application, a question arises as to whether the competitive ratio of an algorithm is likely to be correlated with its performance in practice. Unfortunately, it is far from clear that this is the case. The use of the competitive ratio as a performance measure is roughly equivalent to assuming that request sequences are being generated by a devilish adversary with unlimited computational power and complete

knowledge of the algorithm he is trying to foil. This paranoid view is hard to justify. A challenge for the future is to restrict the adversary or strengthen the on-line algorithm in some way, without retreating all the way to the classical assumption that the request sequences are generated by a fixed stochastic source. Some work along these lines has already been done: [BIRS] considers the paging problem under natural restrictions on the pairs of page requests that can occur consecutively in a request sequence, and [Ir] considers on-line graph coloring with natural restrictions on the type of graph that can be presented, while [CGS] considers the effect of giving the on-line algorithm the ability to look ahead some distance into the request sequence. In other applications it may be reasonable to restrict the adversary's computational power or to assume that, instead of getting to choose each request separately, the adversary, knowing the on-line algorithm, gets to choose a simple deterministic or randomized algorithm for generating the request sequence; the on-line algorithm then faces the task of inferring the adversary's request generation algorithm without incurring too much extra cost. Yet another approach is to mix competitive analysis with average-case analysis by looking for on-line algorithms which not only have a small competitive ratio, but also perform well against "typical" request sequences.

6 Acknoledgement

The author withes to thank Allan Borodin and Amos Fiat for sparking his interest in on-line algorithms.

References

[BBKTW] S. Ben-David, A. Borodin, R.M. Karp, G. Tardos, and A. Wigderson. "On the Power of Randomization in Online Algorithms". *Proc. of the 22nd Ann. ACM Symp. on Theory of Computing*, pages 379–386, May 1990.

[BFKV] Y. Bartal, A. Fiat, H. Karloff, and R. Vorha. "New Algorithms for an Ancient Scheduling Algorithm". To appear in *Proc. of the 24th Ann. ACM Symp. on Theory of Computing*, May 1992.

[BIRS] A. Borodin, S. Irani, P. Raghavan, and B. Schieber. "Competitive Paging with Locality of Reference". In *Proc. of the 23rd Ann. ACM Symp. on Theory of Computing*, pages 249–259, May 1991.

[CDRS] D. Coppersmith, P. Doyle, P. Raghavan, and M. Snir. "Random Walks on Weighted Graphs and Applications to On-line Algorithms". In *Proc. of the 22nd Ann. ACM Symp. on Theory of Computing*, pages 369–378, May 1990.

[CGS] F. Chung, R. Graham, and M. Saks. "A Dynamic Location Problem for Graphs". To appear in *Combinatorica.*

[CL1] M. Chrobak and L. Larmore. "An Optimal On-line Algorithm for the Server Problem on Trees". *SIAM Journal of Computing*, Vol. 20, pages 144–148, 1991.

[FKLMSY] A. Fiat, R.M. Karp, M. Luby, L.A. McGeoch, D.D. Sleator, and N.E. Young. "Competitive Paging Algorithms". *Journal of Algorithms*, Vol. 12, pages 685–699, 1991.

[Gra] R.L. Graham. "Bounds for Certain Multiprocessing Anomalies". *Bell System Technical Journal*, Vol. 45, pages 1563–1581, 1966.

[Gro] E. Grove. "The Harmonic k-Server Algorithm is Competitive". In *Proc. of the 23nd Ann. ACM Symp. on Theory of Computing*, pages 260–266, May 1991.

[Ir] S. Irani. "Coloring Inductive Graphs On-Line". *Proc. of the 31st Symp. on Foundations of Computer Science*, pages 470–479, 1990.

[KT] H.A. Kierstead and W.T. Trotter. "An Extremal Problem in Recursive Combinatorics". *Congressus Numerantium*, Vol. 33, pages 143–153, 1981.

[KT] H.A. Kierstead and W.T. Trotter. "On-Line Graph Coloring". *On-Line Algorithms, DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, eds. L.A. McGeoch and D.D. Sleator, Vol. 7. pages 85–92, 1992.

[KVV] R.M. Karp, U.V. Vazirani, and V.V. Vazirani. "On-Line Algorithms for Bipartite Matching". *Proc. of the 22nd Ann. ACM Symposium on Theory of Computing*, pages 352–358, 1990.

[MMS] M.S. Manasse, L.A. McGeoch, and D.D. Sleator. "Competitive Algorithms for On-line Problems". *Journal of Algorithms*, Vol. 11, No. 2, pages 208–230, 1990.

[MS] L.A. McGeoch and D.D. Sleator. "A Strongly Competitive Randomized Paging Algorithm". *Algorithmica*, Vol. 6, No. 6, pages 816–825, 1991.

[ST] D.D. Sleator and R.E. Tarjan. "Amortized Efficiency of List Update and Paging Rules". *Communications of the ACM*, Vol. 28, No. 2, pages 202–208, February 1985.

[ST1] D.D. Sleator and R.E. Tarjan. "Self-Adjusting Binary Search Trees". *Journal of the ACM*, Vol. 32, No. 3, pages 652–686, 1985.