On-line Algorithms: Competitive Analysis and Beyond

Steven Phillips Jeffery Westbrook

January 24, 1998

1 Introduction

The field of **on-line algorithms** addresses problems in which an algorithm is handicapped by lack of knowledge of the future. An on-line algorithm receives a sequence of inputs, and must process each input in turn, without detailed knowledge of later inputs. Typically the on-line algorithm is managing a system with persistent state, and the changes made to the state affect the cost of processing future inputs.

An example is the following idealization of load balancing on a multiprocessor computer. A sequence of jobs appear on-line, each having a known size. Each job must be irrevocably assigned on arrival to one of n machines. The load on a machine is the sum of the sizes of jobs assigned to it. We would prefer the assignments to balance the load on the machines, and in particular to minimize the maximum machine load. Graham showed that the algorithm that assigns each job to the machine that is currently least loaded achieves a maximum load that is never more than $2-1/n$ times greater than the best possible assignment of the jobs [Graham, 1966]. This is an example of what is now called **competitive analysis**. In competitive analysis, an on-line algorithm is evaluated by comparing its performance to the best that could have been achieved if all the inputs had been known in advance.

On-line algorithms have become an active topic of research. Competitive analysis has been applied to a multitude of on-line problems, and the notion of competitive analysis has been refined in a number of ways. This chapter gives an overview of the scope of research in on-line algorithms. The reader will be introduced to the major topics and problems that have been studied to date, as well as a variety of methods and models that can be used to analyze and solve on-line problems.

We start with a study of the ski rental problem in Section 2. This simple problem arises in a number of contexts, and it provides a natural setting in which to start exploring the design

and analysis of on-line algorithms. The basic models and methods of Section 2 are formalized in Section 3, and a review of some more advanced models is given in Section 4, using paging, an archetypal on-line problem, as the backdrop. Section 5 describes some general purpose algorithms that are applicable to large classes of on-line problems. Lastly, Section 6 is a selective compendium of topics in on-line algorithms that is intended to serve as a starting point for the investigation of old or new on-line problems.

2 The Ski Rental Problem

We begin with a classic example of an on-line problem, "ski rental," first posed in this form by Larry Rudolph. Suppose you decide to learn to ski. After each trip, you will make an irrevocable decision whether to stop skiing or continue learning, and you have no idea in advance what your decision will be. Skiing is an equipment-intensive sport, of course, and before each trip you have two options: rent the equipment at x dollars per day, or buy the equipment for a grand total of y dollars. For convenience we assume that $y = cx$ for some integer $c > 1$. Before each trip to the mountain you have to decide whether to rent or buy. You would like to spend as little money on equipment as possible. Buying equipment before even taking one lesson could be a terrible waste, if you decide to stop after the first trip. On the other hand, if you take many trips then at some point it will have been cheaper to buy than rent. At what point should you stop renting and buy?

The ski rental problem is relevant not only to the management of sporting equipment. It is applicable in a wide variety of resource allocation problems. For example, consider power management in a laptop computer. A typical laptop powers down the hard drive when it isn't in use, because a running hard drive consumes battery power. It takes a significant amount of power and time, however, to start the hard drive running again once it has been powered down. If the user of the laptop doesn't use the hard drive for a while, how long should the laptop wait before powering it down? This is just the ski rental problem— running the drive is renting and powering down is buying (since a one-time cost is incurred when the drive is restarted).

A second application is spinning versus blocking at a lock in a parallel program [Karlin et al., 1991]. If a thread blocks at the lock, it doesn't consume valuable computing cycles while waiting for some other thread to free the lock. However, a context switch is needed to restart the thread when the lock is freed, costing some cycles. If instead the thread spins, waiting for the lock to be freed, then it can resume running as soon as the lock is freed, but it consumes cycles while it waits. The problem

of deciding how long to spin before blocking is just ski rental — spinning is renting and blocking is buying. Other applications include snoopy caching [Karlin et al., 1988] and IP-over-ATM networks [Lund et al., 1994a].

Let's head back to the slopes. There is some number t of ski trips that you will take before stopping (or moving on to that great powder basin in the sky). Suppose you are told t in advance. Then it is easy to decide whether to rent or buy. If $tx \leq y$, you should rent, and otherwise you should buy right at the start. This is the **off-line** ski rental problem. The solution to the off-line problem is called the **optimal solution** and the cost of the optimal solution is called the *optimal cost*. The optimal cost is tx for $t \leq c$ and y for $t > c$.

In the *on-line* problem, the rent-or-buy decision must be made prior to each trip, without knowledge of t. All that can be determined after i trips is that $t \geq i$. Consider the following strategy: rent until $c = y/x$ trips have occurred, and then buy if a $(c+1)$st trip happens. How well does this strategy do? If $t \leq c$, then it is optimal— the minimum possible amount is spent. Suppose $t > c$. Then the cost is exactly twice the optimal cost. Therefore, although this strategy can be optimal in some situations, in the worst case it incurs a cost twice the optimal. This worst-case ratio between the cost incurred by the on-line strategy and the optimal cost is called the **competitive ratio**.

One may ask, is there a better strategy given the rules of the game? A strategy is simply a value k: the number of times to rent before buying. The on-line cost using such a strategy is tx for $t \leq k$ and $kx+y$ for $t > k$. Clearly, there is no value of k that is guaranteed to achieve the optimal cost in all cases. Indeed, any k is non-optimal for the case $t = k+1$, since $kx+y \geq (k+2)x > (k+1)x = tx$. This is typical of on-line problems. Without knowledge of the future, there is no on-line algorithm that is always optimal.

Furthermore, it is not hard to see that no strategy can have a competitive ratio that is lower than 2. The worst-case ratio between the on-line cost and the optimal cost is

$$\max\left\{\frac{kx+y}{tx}, \frac{kx+y}{y}\right\}$$

If $k = 0$, then for $t = 1$ the first ratio is y/x, which is at least 2 by assumption. Otherwise, if $kx \leq y$, then the ratio is at least 2 when $t = k$ (the first ratio in the max), and if $kx > y$ the ratio is at least 2 when $t > k$ (the second ratio in the max).

A skier who abides by the rules of the game might be termed a cautious skier. She can be sure that she never pays more twice what she had to. However, skiing is hardly a sport that attracts

cautious people, so we'll change the rules to add an element of daring. (First we had the "green circle" model, next comes the "blue square" model.)

You are now allowed a **randomized** purchasing strategy. Your decision of whether to buy or rent might be made by a coin flip, or based on your horoscope. Your goal is now to minimize the *expected* ratio to the off-line cost, rather than the worst case ratio.

Let strategy A_i be to rent $i-1$ times, then buy on the ith ski trip. A randomized strategy can be summarized by a probability distribution π, where π_i is the probability you use algorithm A_i. Now assume we want the expected ratio to be at most α. Then if we ski exactly t times, as we noted earlier the optimal cost is tx for $t \leq c$ and y for $t > c$. We must therefore choose π to satisfy

$$\begin{aligned} \mathbf{E}[\text{on-line cost}] &\leq \alpha t x \quad t \leq c \\ \mathbf{E}[\text{on-line cost}] &\leq \alpha y \quad t > c \end{aligned}$$

If we set make these inequalities equalities and solve the resulting equations, we get

$$\pi_i = \begin{cases} \frac{\alpha-1}{c}\left(\frac{c}{c-1}\right)^i & i = 1 \ldots c \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

Finally to solve for α we use $\sum \pi_i = 1$, giving

$$\alpha = \frac{1}{\left(1 + \frac{1}{c-1}\right)^c - 1} + 1. \tag{2}$$

It isn't hard to show that no better competitive ratio is achievable. While this no longer seems like simple financial advice to give to a first-time skier, the solution starts to make sense in a situation of hyper-inflation in ski prices. As $c \to \infty$, so renting becomes infinitesimally cheap compared to buying, the competitive ratio tends to

$$\frac{e}{e-1} \approx 1.58$$

This limit is best thought of in terms of *continuous ski rental*. You embark on a ski trip of unknown duration. You may rent skis, but rather than renting in discrete increments, you are charged for the exact length of time you keep the skis. An any point, if you are still skiing you can choose to buy, incurring a large cost but ending the rental charges. For some applications of ski rental this continuous model is more suitable, for example in the laptop power-management

application, where the power consumed by running the disk during one system clock tick is far less than by powering down and restarting the disk.

That completes the "blue square" model, and we are now ready for the "black diamond" model, namely *repeated ski rental*. Say you first learn to ski. Then you decide to try snowboarding. Then in the summer you take up waterskiing. Each time you have a new rental problem, but you have new information. For example, if you quickly tired of snowboarding, the same may be true for water skiing, so you would do better renting. The same issue arises in various applications. In laptop power-management, the system needs to decide over and over when to shut down the disk, and it can observe the user's disk-access behavior to achieve better performance. In a parallel program, a thread might encounter very similar behavior each time it reaches a particular lock, and that behavior can be used to make a more well-informed decision of how long to spin before blocking [Karlin et al., 1991]. In particular, the thread's behavior might be modeled by a probability distribution for the time the thread must wait for the lock. If the thread chooses the amount of time it spins before blocking, the probability distribution can be used to derive the thread's expected cost (in wasted cycles). The best option for the thread is to choose the amount of time it spins so as to minimize this expected cost. Repeated ski rental has also been applied to IP-over-ATM networks [Keshav et al., 1995]. In addition to having good theoretical properties, the algorithms in [Karlin et al., 1991, Keshav et al., 1995] are shown empirically to have better performance than previous algorithms.

3 On-line Adversaries and the Competitive Ratio.

As suggested in the previous section, many algorithms for on-line problems can be studied by *competitive analysis*. Competitive analysis determines for an on-line algorithm the maximum over all possible inputs of the ratio of the cost incurred by the algorithm on that input to the minimum cost possible on that input. Sleator and Tarjan popularized the use of this performance measure in their seminal paper on sequential search and paging [Sleator and Tarjan, 1985a]. Their use of the performance ratio, though not new (having appeared in earlier bin-packing and scheduling work, for example [Graham, 1966]), triggered the proverbial avalanche, and competitive analysis has since been applied to an immense range of on-line problems.

In general, let F denote a family $\{I_1, I_2, \ldots\}$ of instances of an on-line problem. We require that an instance be finite, containing a bounded number of requests. Let $A(I)$ denote the cost of

algorithm A on instance $I \in F$, and let $\mathrm{OPT}(I)$ denote the optimum cost to solve instance I. Then A is *c_F-competitive* if, for all $I \in F$,

$$A(I) \leq c_F \mathrm{OPT}(I) + b_F \tag{3}$$

where c_F and b_F are constant with respect to I. They are in general functions of the family, F.

For example, in the ski rental problem, an instance is simply the number t, and the family of instances is the natural numbers. The competitive ratio of the green circle strategy is 2, because Equation 3 holds with $c_F = 2$ and $b_F = 0$.

As an example of a more complicated situation, consider the *on-line Steiner tree problem*, defined as follows. An instance consists of a connected, weighted graph $G = (V, E)$ of n vertices and m edges, and s set $R \subseteq V$ of *required* vertices. A Steiner tree on R is a minimum-weight subtree of G that contains all the vertices in R. In the on-line problem, the set R is revealed on-line. Initially the subtree is empty. As each node is revealed, enough additional edges must be added to the tree so that the new node is connected to all previously revealed nodes. Once an edge has been added, it cannot be removed. The on-line Steiner tree problem arises in several settings, including facility location planning of telecommunication networks, and as a subproblem of file allocation and load-balancing problems in networks.

Let $F(n, m)$ denote the family of instances of the on-line Steiner tree problem on n-node, m-edge weighted graphs, where the edge weights can be arbitrary. The "greedy algorithm,", which always connects a new node to the previous tree in the cheapest possible way, is $O(\log n)$-competitive with $b_F = 0$ [Imase and Waxman, 1991]. Furthermore, if b_F is required to be 0, then every on-line algorithm for the Steiner problem has a competitive ratio that is $\Omega(\log n)$ [Imase and Waxman, 1991]. Here the competitive ratio is a function of the family of graphs, but independent of the weights. On the other hand, if we define $F(n, m, B)$ to be the family of n-node, m-edge graphs with all edge weights positive and total edge weight bounded by B, then the greedy algorithm is 0-competitive with $b_F = B$, since the weight of any solution is no more than the total edge weight.

The point of this discussion is that when we talk about the competitive ratio of a given algorithm, we will have Equation 3 in mind, but the precise details of the model (the definition of the family) and the restrictions we place on the additive constant b_F may have a rather profound effect on the competitive ratio that we end up with for the algorithm.

3.1 Randomized Algorithms

A randomized algorithm is an algorithm that has access to an oracle that generates random bits when requested. It can use these random bits to "flip coins" and make different decisions based on the random bits. The competitive ratio is defined with respect to expected cost. Randomized algorithm A is *c_F-competitive* if, for all $I \in F$,

$$\mathbf{E}[A(I)] \leq c_F \mathrm{OPT}(I) + b_F \tag{4}$$

The expectation is with respect to the sequence of random choices that A makes.

3.2 Adversaries

It is often useful to think of an on-line problem as a game between two players. One player, the **adversary**, generates requests on-line according to some specified mechanism and is charged a cost for its selections according to some specified rule. For example, it can be charged the optimum cost to satisfy the complete sequence of requests. The other player is the algorithm, which responds to requests by making a decision, and incurs some total cost over the course of the game. The adversary's goal is to maximize the ratio of the algorithm's cost to the adversary's cost, while the on-line algorithm's goal is to minimize it.

The basic definition of competitiveness for non-randomized (deterministic) algorithms, given at the start of Section 3, corresponds to an adversary that has complete knowledge of the algorithm, that is allowed to generate any request sequence, and that is charged the minimum (off-line) cost to perform any request sequence it generates. Such an adversary can first decide how many requests it wants to generate, and then internally simulate the algorithm on each possible sequence of that length to find the sequence that maximizes the ratio between the on-line algorithm's cost and the adversary's cost. This kind of adversary is called a *deterministic off-line* adversary

The definition of competitiveness for randomized algorithms given in Section 3.1 corresponds to an adversary that has complete knowledge of the algorithm but does not know what random bits the algorithm will have. It can generate any request sequence, and is charged the off-line cost. The adversary can internally simulate the algorithm with all possible strings of random bits to find a sequence that maximizes the expected ratio of costs. Such an adversary is called a *randomized oblivious* adversary (because it is oblivious to the actual random choice made by the on-line algorithm). Rather more subtle analyses of randomized algorithms are possible with when the adversary is allowed to know the random bits. We will return to this in a later section.

3.3 Extending the notion of competitiveness.

Our plan is to measure the quality of an on-line algorithm by its competitive ratio. But is this always the best measure? For example, a person who was really considering learning to ski might not be convinced that the deterministic (green circle) and randomized (blue square) algorithms for ski rental are truly the best ones, despite our argument that they are optimal according to the measure of competitive ratio. Such a person might complain that the competitive ratio is not always the right way to measure of quality. For example, one might rather want to minimize the maximum cost incurred (in which case you should always buy right away). A number of other approaches to measuring the quality of an on-line strategy can be found in [Luce and Raiffa, 1957]. The competitive ratio is the only one used in computer science, however, and it seems to be flexible and useful, so we shall limit our attention to this measure.

Another possible complaint about competitive analysis is that the assumption of no knowledge about the future is overly pessimistic. For example, it may be unreasonable to assume that the fact that one has already made i ski trips gives no information about whether or not one will make the $(i+1)$st trip. In real life, people are not truly arbitrary. For some people, the more one has gone, the more likely one is to go again. Other people will try anything once but nothing twice, and so on. An on-line algorithm might be able to take advantage of these characteristics to reduce the overall cost. But the basic definition of competitive analysis does not take this into account. It presupposes a worst-case adversary, which is allowed to generate request in a completely capricious fashion. Such an analysis provides no way to differentiate a ski-rental strategy that tries to take advantage of observed characteristics of the request sequence, and one that always does the same thing (such as buying after y/x trips). In the worst case, both will have a competitive ratio no better than 2. In fact, an algorithm that does well for "reasonable" people might have a competitive ratio higher than 2.

We can account for this kind of situation within the basic framework of competitive analysis, however, by limiting the way that the adversary chooses request sequences. For example, in the "blue square" version of ski rental, the adversary must choose t according to some probability distribution D. Instance I occurs with probability p specified by D.

The definition of competitive ratio is extended to include probability distributions over the family of instances in the following way. An algorithm is c_F competitive on distribution D over a

family of instances F if

$$\mathbf{E}[A(I) - c_F \cdot \mathrm{OPT}(I)] \leq b_F. \tag{5}$$

Here the expectation is over the distribution D on instances in F. By the linearity of expectations, this is equivalent to

$$\mathbf{E}[A(I)] \leq c_F \cdot \mathbf{E}[\mathrm{OPT}(I)] + b_F. \tag{6}$$

As an example, consider a version of ski rental in which the probability that t trips are taken is $e^{-t} \cdot \frac{e-1}{e}$ (the quantity $\frac{e-1}{e}$ is a normalizing constant). This models the case where one is likely to get bored with skiing (or become irreparably injured) as time goes on. The expected number of trips in this situation is $\frac{1}{e-1}$. One on-line strategy in this situation is to always rent, leading to an expected cost of $\frac{x}{e-1}$. This strategy turns out to be optimal, although we do not prove it here. The optimum off-line strategy for a given t is as in Section 2, and the exact closed-form expression for the expected off-line cost can be computed fairly easily. The adversary can maximize the ratio of the on-line to off-line cost by setting $y = x$. In this case the competitive ratio is $\frac{e}{e-1}$.

The reader may recall from Section 2 that the value $\frac{e}{e-1}$ is also the optimal competitive ratio that can be achieved by a randomized algorithm against a worst-case adversary. This is neither a coincidence nor a simple trick perpetrated by the authors of this article, but rather a deep principle of on-line analysis known as *Yao's minimax theorem* [Yao, 1980]. This theorem is actually an adaptation of the famous minimax theorem of game theory [von Neumann and Morgenstern, 1947]. It states that the best ratio achievable by a deterministic algorithm against any distribution is exactly the same as the best ratio achievable by a randomized algorithm against a worst-case adversary.

More formally, for a given on-line problem let F_n denote the family of input instances of size at most n. Let $\mathcal{D}_n$ denote the set of all probability distributions over the instances in F_n. Let $\mathcal{A}_n$ denote the set of deterministic on-line algorithms for instances of size n. We think of a deterministic on-line algorithm as being a decision tree; a node at depth i in the tree represents a decision about how to respond to the ith request. Finally, let $\mathcal{R}_n$ denote the set of probability distributions on the algorithms in $\mathcal{A}_n$. A randomized on-line algorithm is a simply a particular probability distribution $R \in \mathcal{R}_n$ on the deterministic algorithms in $\mathcal{A}$. Yao's minimax principle says

$$\lim_{n\to\infty} \left\{ \max_{D\in\mathcal{D}_n} \min_{A\in\mathcal{A}_n} \frac{\mathbf{E}_{I:D}[A(I)]}{\mathbf{E}_{I:D}[\mathrm{OPT}(I)]} \right\} = \lim_{n\to\infty} \left\{ \min_{R\in\mathcal{R}_n} \max_{I\in F_n} \frac{\mathbf{E}_{A:R}[A(I)]}{\mathbf{E}_{A:R}[\mathrm{OPT}(I)]} \right\}. \tag{7}$$

The notation $\mathbf{E}_{I:D}$ indicates that the expectation is over instances $I \in F_n$ according to the probability distribution D. Similarly, $\mathbf{E}_{A:R}$ indicates the expectation over algorithms $A \in \mathcal{A}_n$ according to the probability distribution R. Several variants of this minimax principle have been used in the analysis of on-line algorithms; see for example [Borodin and El-Yaniv, 1998].

4 Paging: A Classic On-Line Problem

Paging is the archetypal problem in on-line algorithms, the problem that has been the testing-ground for most new methods of analysis of on-line algorithms.

Consider a hierarchical storage system consisting of a small, fast memory, that can store k memory pages, and a large slow device with room for n pages, with $n \gg k$. The classic instantiation is a virtual memory system, with the fast memory constructed of RAM and the slow device being a hard disk. (There are other instantiations, see for example [Keshav et al., 1995, Lund et al., 1994a].) When a program references a memory location in a page that isn't in memory, a *page fault* occurs, and the program is blocked and must wait while the page is loaded into memory. Some page must be evicted from memory to make room for the page being loaded, and a *paging algorithm* chooses which page to evict. The aim of the paging algorithm is to minimize the number of page faults caused by the sequence of memory references.

The first application of competitive analysis to paging occurred in the seminal CACM paper of Sleator and Tarjan [Sleator and Tarjan, 1985a]. They analyzed the performance of the Least Recently Used (LRU) rule, an algorithm widely regarded as very effective in practice. They proved (generalizations of) the following two theorems, whose proofs we provide as examples of the genre.

Theorem 1 *LRU has competitive ratio k.*

Proof: Partition the page reference sequence σ into $\sigma_0, \sigma_1, \ldots, \sigma_i$, such that LRU faults exactly k times in $\sigma_1, \ldots, \sigma_i$ and at most k times in σ_0. For $1 \leq j \leq i$, let p_j be the last page referenced in σ_{j-1}. During σ_j, LRU cannot fault twice on any page, and it cannot fault on p_j. Therefore σ_j contains references to k different pages, all different from p_j, so the optimal algorithm must fault at least once in σ_j. This suffices to prove the theorem. □

Theorem 2 *No deterministic on-line algorithm has a competitive ratio less than k.*

Proof: Let $\mathcal{A}$ be a deterministic algorithm, and consider the case $n = k + 1$. Let σ be the sequence constructed by always referencing the page that is *not* in $\mathcal{A}$'s fast memory. $\mathcal{A}$ faults on each reference, whereas the optimum algorithm, which always evicts the page whose next reference is furthest in the future [Belady, 1966], faults at most once every k references. □

A better competitive ratio is achievable through the use of randomization. [Fiat et al., 1991] gives a simple randomized algorithm that has competitive ratio $2\mathcal{H}_k$, where $\mathcal{H}_k = \sum_{1 \leq i \leq k} 1/k$ is the k harmonic number, and is $\Theta(\log k)$. A more complex algorithm has competitive ratio $\mathcal{H}_k$ [McGeoch and Sleator, 1991], matching a lower bound proved by Fiat *et al.*.

These tight bounds for deterministic and randomized competitive ratios are pleasing, particularly in the case of an empirically good algorithm such as LRU. There are, however, some limitations with these results. First, other non-randomized algorithms that behave worse than LRU in practice, such as First-In First-Out (FIFO), also have a competitive ratio of k. Second, on traces taken from program executions, the performance ratio of LRU is much less than k, or even $\log k$, typically close to 2 [Young, 1991].

Borodin *et al.* added some realism by modeling *locality of reference*, the property of page reference sequences that makes hierarchical storage useful at all. They modeled locality using an access graph, where the reference sequence is restricted to be a walk through the graph. In addition to analyzing LRU in this model, they considered how to use advance knowledge of the access graph, an approach continued by Irani *et al.* [Irani et al., 1992] and Fiat and Karlin [Fiat and Karlin, 1995]. Fiat and Mendel [Fiat and Mendel, 1997] show how to optimally use the access graph, even when it is not known in advance but must be learnt as the requests arrive. In addition, they need only store the most recently seen part of the access graph, rather than the entirety.

As discussed in Section 3.3, an access graph is a way to limit the power of the adversary, restricting the possible reference sequences so that the results obtained have more bearing on paging in practice. A alternative approach was taken in the earliest theoretical treatments of paging in which reference sequences were assumed to be generated by various probability distributions. In one study, each page reference was chosen according to a fixed probability distribution over the set of pages [Franaszek and T.J.Wagner, 1974], while others did probabilistic analyses of LRU in which page references were chosen according to LRU stack-depth (where the ith most recently accessed page has stack-depth i) [Lewis and Shedler, 1973, Shedler and Tung, 1972]. Results in these papers are specific to the particular probabilistic model of reference sequences, however, and we'd like more

general-purpose results.

A step in this direction was the Markov paging model of Karlin *et al.* [Karlin et al., 1992], in which the locality of reference inherent in a program is modeled by a Markov chain. Each node in the chain corresponds to a page and each transition into a state generates a reference to that page. Hence a reference sequence is generated by the probabilistic transitions of the chain. Karlin *et al.* found an algorithm whose expected page fault rate on any Markov chain is within a constant factor of the best possible for that chain. This approach was extended by Lund *et al.* [Lund et al., 1994a], who show that for *any* distribution over page reference sequences, there is a natural algorithm that gets within a constant factor of the best possible expected fault rate for the distribution, which needs only to know, for pages i and j in the fast memory, the probability that i will be referenced before j.

Koutsoupias and Papadimitriou [Koutsoupias and Papadimitriou, 1994b] introduced two other alternative analysis techniques: the diffuse adversary model and comparative analysis. In the first technique, reference sequences are generated by some distribution $\mathcal{D}$ from a set Δ. The on-line algorithm can take advantage of knowing Δ, but doesn't know which $\mathcal{D}$ will be used, and its expected fault rate is compared to the expected optimal fault rate on sequences from $\mathcal{D}$. In comparative analysis, rather than comparing on-line algorithms to the optimal algorithm, we compare two arbitrary classes of algorithms. Koutsoupias and Papadimitriou use the diffuse adversary mode to exhibit a simple family Δ for which LRU is the best on-line algorithm, and comparative analysis to investigate the value of lookahead in paging, comparing algorithms with lookahead to those without.

Torng [Torng, 1995] includes the time to access fast memory, in addition to the time incurred during a page fault, in the cost of servicing a page reference sequence. This has a number of attractive consequences: lookahead provably helps, and on reference sequences exhibiting a natural notion of locality of reference, some algorithms (including LRU) achieve a *constant* competitive ratio. Another model for analyzing on-line algorithms, and algorithms for paging in particular, is Young's loose competitiveness [Young, 1994].

5 General Models for On-line Problems

In this section we review some of the general models and tools that have been developed for on-line problems. If a given on-line problem can be phrased in terms of one of these models, standard

algorithms are available to solve the problem. In this section the adversaries are assumed to be worst-case.

Most of the models we discuss are defined over *metric spaces.* A metric space, M, is a pair (P, δ) where P is a set (finite or infinite) and δ is a distance function from $P \times P$ to $\mathcal{R}$. A typical example of a metric space is the plane with Euclidean distance. The function δ must satisfy several conditions:

1. $\delta(a, a) = 0 \ \forall \ a \in P$.
2. $\delta(a, b) \geq 0 \ \forall \ a, b \in P$.
3. $\delta(a, b) = \delta(b, a) \ \forall \ a, b \in P$.
4. $\delta(a, b) + \delta(b, c) \geq \delta(a, c) \ \forall \ a, b, c \in P$.

5.1 The k-server Model

The k-server model, defined in [Manasse et al., 1988], had its genesis in paging problems. Let M be a metric space and let S be a set of k *servers.* The servers are always located on k not necessarily distinct points $s_1, \ldots, s_k$ in M. An instance of the k-server problem consists of a metric space M, the initial positions of the servers, and a sequence σ of requests, which are revealed one-at-a-time. A request is simply a point r in M. The servers must satisfy the following condition.

> Let r be the most recently revealed request point. There must be at least one server, i, such that $s_i \equiv r$.

After request r is revealed, the set of servers are moved as necessary so that at least one server is located at the request point. Let s'_i denotes the position of server i after being moved. The cost moving the servers is $\sum_i \delta(s_i, s'_i)$. The total cost of servicing σ is the sum over requests of the movement costs.

Example: Paging. The paging problem is modeled as a k-server problem in the following way. There is one point in P for each unique page of memory. There is one server for each cache location. Define $\delta(a, b) = 1$ for any pair $a, b \in P$. If server i is located at point p, then page p is in cache location i. If no server is located at point p then the corresponding page is out of cache. Moving server i from a to b corresponds to ejecting page a and loading b into slot i.

When Manasse, McGeoch, and Sleator [Manasse et al., 1988], first proposed the k-server model, they conjectured that there exists a deterministic algorithm that has competitive ratio of k on any instance of the k-server problem (not just a paging instance). This became the rather famous "k-server conjecture." For deterministic algorithms, k is the best possible ratio. This follows from the lower bound on the competitive ratio of paging algorithms.

So far, the truth of the k-server conjecture has not been resolved, but a good candidate for a deterministic algorithm with ratio k is the *work-function algorithm*, developed independently by several researchers [Borodin et al., 1987], [Chrobak and Larmore, 1992], [Manasse et al., 1988]. The work-function algorithm keeps track of the optimal off-line costs and tries to make moves that keep its cost close to the optimal cost. Given a sequence of m tasks, the optimal off-line cost can be computed with a simple dynamic programming algorithm. Let $g_i(X)$ denote the minimum cost to perform tasks 1 to i and end in configuration X, where $X = (s_1, \ldots, x_k)$. Then $g_0(X) = \delta(X_0, X)$, where X_0 is the start configuration, and

$$g_i(X) = \min_{X'}\{g_{i-1}(X') + \delta(X, X')\}. \tag{8}$$

The optimal cost to process the sequence is $\min_X g_m(X)$.

The function $g_i(\cdot)$ is an example of a *work function*. A work function is a function mapping the metric space M to the positive reals, with the property that the value of the work function at point lower-bounds the optimal cost to process requests 1 through i and end at position j.

Work Function Algorithm. Let r be the tth request, and let X_{t-1} be the configuration of the on-line servers when r arrives. Let X_t be the configuration X, $r \in X$, that minimizes $g_t(X) + \delta(X_{t-1}, X_t)$.

Koutsopias and Papadimtriou [Koutsoupias and Papadimitriou, 1994a] showed that the work-function algorithm is $2k - 1$ competitive, but the proof is beyond the scope of this survey. Note that the work-function algorithm requires space $\Theta(n^k)$ space, which is quite prohibitive for large k.

Sometimes it is useful to compute the optimal off-line strategy. For example, it is useful in the design of a prefetching strategy for a data-independent computation flow, as might occur in a large matrix application such as Gaussian elimination or an eigenvalue calculation. Chrobak *et al.* [Chrobak et al., 1991] showed how to reduce finite instances of the off-line k-server problem to an instance of the network flow problem. If there are n points in the metric space, and there are m requests, the flow instance consists of a network of size $O(k(m + n))$. Using a standard

network flow algorithm [Tarjan, 1983] a solution to the off-line problem can be found in time $O(k^2(m+n)^2 \log(k(m+n)))$.

5.2 Metrical Task Systems

Borodin, Linial and Saks [Borodin et al., 1987] proposed a more general model of on-line computation called *metrical task systems.* A metrical task system consists of a set of n states S and a distance function δ so that (S, δ) is a metric space. At any time, the system must be in exactly one of the n states. A sequence σ of *tasks* is revealed, one task at a time. Task t_i is an arbitrary cost vector, $(c_{i1}, \ldots c_{in})$, that maps states to costs. Value c_{ij} is a non-negative real number. If the system is in state j at the time task i is processed, the cost of the task is then c_{ij}. The cost vectors can be entirely arbitrary and need have no relation to each other or the state transition function. If the function δ is not symmetric but still obeys the triangle inequality then (S, δ) is just called a *task system.*

The notion of *lookahead* arises in metrical task systems. Informally, an on-line problem has *lookahead* k if an on-line algorithm is allowed to know the next k requests when making decisions about how to change state. More formally, an on-line problem modeled as a metrical task system proceeds in a series of rounds. At the start of a round, the on-line algorithm is shown the next k tasks. The on-line algorithm can then change the state of the system at cost $\delta(s, s')$, where s and s' are the old and new states, respectively. Finally, the on-line algorithm "accepts" the next task t, at cost $c_{ts'}$. This ends the round. The value of k may be a function of the number of states n, but it cannot be a function of the number of tasks. An off-line algorithm, which can see the entire sequence of tasks, is said to have *unbounded* lookahead.

Example: sequential search. This problem models searching for an item in a linked list containing n items. After each search, the list can be rearranged to move more commonly accessed items forward. Section 6.1 gives more details. The metrical task system (S, δ) has one state for each of the $n!$ permutations of the list. The cost of moving between two states is given by the number of inversions between the two lists: it is not hard to show that one list can be rearranged to any other with a number of exchanges equal to the number of inversions between source and target list. An access to element x is modeled by a task t such that c_{ts} is the position of item x in permutation s. This model has lookahead 0.

Example: paging. There are $\binom{m}{k}$ states, one for each possible subset of k pages in cache out of m total pages. The cost of moving between states is equal to the the number of pages they differ in. An access to page x is modeled by a task t such that c_{ts} is 0 if s contains x in cache and $+\infty$ otherwise. The paging problem has lookahead 1. This reduction can be extended to model the k-server problem as a metrical task system by letting the cost to change state be the minimum distance servers must move to make the change.

It is evident that the task system model is very general and encompasses a large range of on-line problems. Borodin, Linial and Saks gave a general-purpose algorithm for the model.

The Borodin-Linial-Saks algorithm. This algorithm for task systems is a straightforward extension of the work function algorithm of Section 5.1, although in fact the BLS algorithm appeared first. As before, we keep track of the optimal off-line cost with a function $g_i(s)$, which denotes the minimum cost to perform tasks 1 to i and end in state s. This function can be computed as in the section on k-servers. Let r be the tth request, and let s_{t-1} be the current state when r arrives. Move to the state s_t that minimizes $g_t(s_t) + \delta(s_t, s_{t-1}) + c_{ts_t}$.

When δ is not symmetric, it is important that $\delta(s_t, s_{t-1})$ be used in the minimization, rather than $\delta(s_{t-1}, s_t)$. The formulation given above is different from that in [Borodin et al., 1987] but the two are essentially equivalent.

Borodin, Linial, and Saks prove that this algorithm is $2n-1$-competitive on any metrical task system and $O(n^2)$-competitive on any task system that is not metrical. Furthermore, they show that no deterministic algorithm is better than $2n-1$-competitive. This result is rather depressing, since the number of states could be very large, as in the sequential search example above. There the number of states is $k!$, where k is length of the list. On the other hand, there are practical problems such as multi-processor page migration (see Section 6.3) in which the number of states is much smaller and plenty of processing time is available, and the BLS algorithm is certainly applicable in these situations.

Like the work function algorithm, the BLS algorithm requires lookahead 1. The paging example shows that there cannot be a general algorithm with lookahead 0 that has bounded competitiveness. Let $\delta(a, b) = 0$ for all states a, b. If the on-line algorithm is in state i, an adversary simply creates a task that has cost 1 in state i and cost 0 in all other states. The off-line cost is always 0 while the cost to the on-line algorithm can be made arbitrarily high by creating a sufficiently long sequence.

Recently Bartal *et al.* [Bartal et al., 1997a] found a randomized algorithm for metrical task systems that achieves a competitive ratio that is polylogarithmic in n. The algorithm is rather complicated, however, and lies beyond the scope of this section.

5.3 Request-Answer Games

The most general model of on-line problems that appears in the literature is *request-answer* games [Ben-David et al., 1990]. This model exploits the relationship between on-line algorithms and game theory hinted at in Section 3.2 An on-line problem can be regarded as a game between two players, the algorithm and the adversary. At each round of the game, the adversary makes a move, which is a request plus a list of legal responses (answers) and associated costs. The algorithm then selects one of the answers to respond with. Two types of adversaries have been considered: oblivious adversaries, which do not know what responses the algorithm gives and must decide on the next move in the dark, and adaptive adversaries, which are given complete information about the algorithm's responses. The distinction is only relevant when the algorithm is randomized. If the algorithm is deterministic, and adversary can simulate the algorithm and so know exactly what moves are made.

The adversary is charged a cost for the sequence of requests it generates. The cost can be assessed in two ways. The adversary may be charged the minimum cost to answer the requests using the given answers. This is the off-line adversary. Alternatively, the adversary may have an associated on-line algorithm that it must choose in advance. The cost incurred by this associated on-line algorithm is the cost charged to the adversary. This is the on-line adversary. Again, the distinction is moot if the algorithm is deterministic, because the adversary can simulate the algorithm, choose a sequence of requests, and then select an "on-line" algorithm that just happens to exactly minimize the cost on that particular sequence.

The algorithm wins against a particular adversary if the cost incurred by the algorithm is no more than a constant c times the cost incurred by the adversary. In this case we say the algorithm is c-competitive against the adversary. Because the request-answer model is so general, there cannot be an algorithm that is always competitive against all adversaries.

6 The Trail Map: A selective guide to On-line problems

In recent years there has been a flurry of work on on-line problems. Competitive analysis has been applied to a mountain of old and new problems. This section contains a guide to some of the major areas that have received attention.

6.1 Data structure problems

Most dynamic data structures can be studied as on-line decision problems. So far, however, only two have received much attention.

In the *sequential search* or *list-update* problems, a dictionary of keys is stored in a linked-list data structure. The list may be searched for a particular key by scanning the list in order from front to back, stopping the scan as soon as the item is found. After each search, the list can be reordered by swapping adjacent pairs of elements. By moving items that are frequently searched for to the front of the list, the overall search time can be decreased, however the total cost of a request sequence is the sum of the search time for elements plus the cost of the swaps. Swaps that move the searched-for item forward are free. A popular optimization is the "move-to-front" rule: after a search succeeds, the found element is moved to the front of the list.

Early work on sequential search (see for example [Rivest, 1976]) studied the performance of the move-to-front heuristic against an adversary that generates requests using a fixed probability distribution on the list elements. The adversary is charged the cost incurred by an static list that is optimally sorted for the given distribution (*i.e.*, the item with highest probability of access is at the front, then the second highest probability, and so on). Chung *et al.* [Chung et al., 1985] showed that move-to-front never incurs a cost greater than $\pi/2$ times the cost incurred by the optimal static list, and [Gonnet et al., 1979] showed a distribution on which move-to-front did this badly.

A sea-change occurred in 1985, starting with [Bentley and McGeoch, 1985] and followed shortly by Sleator and Tarjan's seminal paper on competitive analysis [Sleator and Tarjan, 1985a], which gave the first analysis of an on-line algorithm against the optimal off-line adversary. Sleator and Tarjan showed that move-to-front is 2-competitive. It is not hard to see that this is the best possible competitive ratio for a deterministic algorithm by considering a sequence in which the adversary always accesses the last element in the on-line algorithm's list. Since the early 1990's research has concentrated on randomized algorithms. The best known algorithm is 1.6-competitive against an oblivious adversary [Albers and Mitzenmacher, 1996], and a lower bound of 1.5 is known for all

randomized algorithms [Teia, 1993].

Another important data structure for storing and searching is the dynamic binary tree. An item is searched for using the standard binary tree search algorithm. After each search, the tree may be reorganized by doing a series of *rotations*. Let (u, v) be a tree edge such that v is the right child u. A rotation of edge (u, v) makes u the new left child of v, the old left child of v becomes the new right child of u, and v replaces u as the left or right child of the old parent of u. Rotations are a fundamental method of reorganizing binary trees, and more information can be found in a standard introduction algorithms textbook such as [Cormen et al., 1990]. The goal of the rotation is to reduce the search time, by rotation frequently accessed items nearer to the root of the search tree.

Any standard balanced search tree such as AVL-trees or red/black trees (see [Cormen et al., 1990]), ensures that the cost of an individual search is $O(\log n)$ and that $O(\log n)$ rotations are done per search. Since the optimal cost is at least 1 per search, balanced trees are trivially $O(\log n)$-competitive. Although there has been considerable work on the problem no one has yet produced a reorganization algorithm that is provably better than $O(\log n)$-competitive. Conversely, no lower bound greater than 1 is known. It is known, however, that of all dynamic binary search tree algorithms that have so far been proposed, only one can be better than $O(\log n)$ competitive. That candidate is the splay tree data structure of Sleator and Tarjan [Sleator and Tarjan, 1985b]. Sleator and Tarjan showed that splay trees are $O(\log n)$-competitive, and conjectured that they are in fact $O(1)$-competitive. This conjecture has been shown true or nearly true in certain special cases but the competitiveness of splay trees remains open in the general case, and is one of the most interesting open problems in the theory of data structures and on-line algorithms. See the survey on on-line data structure problems by Albers and Westbrook in [Fiat and Woeginger, 1998] (also available at `http://www.research.att.com/ jeffw/data-survey.ps`).

6.2 Network admission control

The explosive growth of the Internet and World Wide Web, and the promise of revolutionary services combining voice, video and data communications in Asynchronous Transfer Mode (ATM) networks provide a fertile source of on-line problems. The main focus has been on admission control and routing problems in ATM networks, starting with Garay and Gopal [Garay and Gopal, 1992] and Garay *et al.* [Garay et al., 1993], who studied preemptive call control on a single link or a line

network. Awerbuch *et al.* [Awerbuch et al., 1993] applied approximation techniques for multicommodity flow problems to give an elegant analysis of the following problem: calls are offered to a network on-line, each call having a bandwidth requirement, source and destination nodes and duration. The goal is to accept or reject calls, routing accepted calls subject to edge capacity constraints, so as to maximize the *throughput*, defined as the average over time of the aggregate bandwidth of accepted calls. Subject to a restriction that no call bandwidth is more than $1/\log\rho$ times the minimum edge capacity, they give an algorithm that is $\log\rho$-competitive, which is best possible, within constant factors, for deterministic algorithms. Here ρ is the product of the number of nodes in the network and the ratio of the maximum to minimum job duration. The algorithm essentially uses shortest-path routing with a load-dependent cost metric. More specifically, the cost of an edge is exponential in the load (fraction of capacity used), the cost of a path is the sum of its edge costs, and a call, if admitted, is routed on a least cost path. Load-dependent routing has been used in practice as an important component of established networks; Gawlick *et al.* [Gawlick et al., 1994] show that the exponential cost model gives good results in a practical setting.

A number of variants of competitive routing and admission control have been developed, for example where tight bounds are achieved for tree networks, arrays and hypercubes, where the objective is to minimize congestion rather than throughput. A good survey of these results is given by Plotkin [Plotkin, 1995].

6.3 Data management in networks

The problem of managing data in a network of computers can be abstracted as follows. A collection of n computers is connected by some kind of connection network. The network is modeled as an undirected graph whose nodes are computers and whose edges correspond to direct connects between computers. Each edge has a weight, which is a measure of the cost to send a message across the corresponding connecting. (The cost may model delay, dollar cost per message, or any other actual cost measure.) The cost to send a message of size q bytes, $q \geq 1$, from machine i to machine j is modeled as $q \cdot \delta(i,j)$, where $\delta(i,j)$ is the total weight of the shortest path connecting i and j.

Each computer has some amount of local storage and supports a collection of local processes (users, or programs). A large collection of data, which we call the "file," is to be read or written concurrently by the processes. The file is partitioned into D records; a *request* at machine i is either

a read or a write of an individual record. There may be one or more complete copies of the file stored in the network, each stored at a single machine. If machine i generates a read request, that request is satisfied at zero cost if the machine is storing a copy of the file. Otherwise, a message must be sent through the network to the nearest machine j storing a copy of the file. The requested record is then sent back, at cost $q\delta(i,j)$ for some constant q. For convenience, we assume $q = 1$. A write request is slightly more complicated. If there is a single copy in the network, then the write request is handled the same way as a read request. If there are multiple copies, all copies must be kept consistent. Hence if a machine generates a write request, all machines containing a copy of the file must be notified. The message cost is lower-bounded by the weight of a minimum Steiner tree containing the requesting machine and all machines with a copy. This can be achieved (within constant factors) if the machines can store and forward messages intelligently (*i.e.*, each machine transmits messages only to its adjacent neighbors in the Steiner tree. (Computing a Steiner tree is NP-hard, of course, but although we use a Steiner tree to model costs, in practice a Steiner tree can be approximated within a factor of 2 by a minimum spanning tree, which is fast to compute.)

The overall goal is to minimize total message cost. As the request patterns change, it may be advantageous to make new copies of the data file. For example, a machine that is generating many reads can benefit from keeping a copy locally, although the addition of a new copy will increase the message cost of subsequent writes. Similarly, if there are many write requests compared to reads, it may be useful to decrease the number of copies. A copy of the file can be placed at machine i by transmitting the D records to i from the nearest machine j with a copy. The cost is $D \cdot \delta(i,j)$. A copy can be discarded from a machine at no cost. Of course, there must always be one copy of the file in the network.

Although this model is idealized it captures much of the main difficulties in handling files in networks. The static problem of assigning a fixed set of copies has been extensively studied. For the static problem a probabilistic distribution is typically assumed and then the expected cost per unit time is calculated for various cost models. Dowdy and Foster give a survey of these results [Dowdy and Foster, 1982].

The on-line problem has been studied in various settings. Two interesting restrictions are *replication*, in which there are only read requests, and *migration*, in which the file can be moved but there is never more than one copy. Migration and replication where the earliest of this class of problem to be studied in an on-line setting [Black and Sleator, 1989]. The general problem, which

allows for reads, writes, and multiple copies, is called *file allocation*.

Consider replication in a network of two machines connected by an edge. Initially there is one copy, at machine a. Now suppose b begins to generate read requests. When should a copy be placed at b? A little reflection reveals that this is exactly the ski rental problem, with the rent cost 1 and the buy cost D. Hence there is a two-competitive deterministic algorithm and an $e/(e-1)$ randomized algorithm. For general networks, however, the problem becomes much harder. The off-line adversary, having decided what machines will need copies in the end, will achieve minimum cost by replicating all copies at the start along a minimum Steiner tree connecting the machines. The on-line algorithm must essentially generate on-line an approximation to this minimum Steiner tree. The competitive ratio for this problem is $\Theta(\log n)$ [Imase and Waxman, 1991].

On-line migration has the advantage of avoiding the problem of maintaining consistency and is popular in designs of virtual shared memory. There are several deterministic and randomized algorithms, all with low constant competitive ratios, in the range 2–5. See [Bartal et al., 1997b] for further details.

For file allocation, deterministic and randomized algorithms are known that are $O(\log n)$-competitive (the best possible up to constant factors). If the network is a bus or a tree, small constant competitive ratios can be achieved, in the range 2–3. In general networks, there is also a problem with simply keeping track of the nearest copy of the file. If the cost of this data tracking problem is included, the best-known competitive ratio increases to $O(\log^2 n)$. See [Lund et al., 1994b] for more information.

There are various related problems, such as providing fault tolerance [Westbrook and Zuck, 1994] or handling bounded storage at each machine $\Theta(n)$ [Bartal et al., 1992]. A completely unexplored problem is when there is a cost per unit storage.

6.4 Robot Searching and Navigation

In this class of problems, an automaton is placed in an unknown environment, and asked to find its way to a certain location, or to find an object hidden in the environment. It discovers the nature of the environment only by exploring. In general, the goal is to limit the amount of time-consuming exploring required to reach the goal.

In the most abstract setting, the environment consists of a directed graph. The robot is started at a some node in the graph, and asked to reach a particular other node. Each node can be uniquely

identified by the robot, and when at a node, the robot can see and identify the set of incident edges. But it does not know where an edge leads until it has crossed the edge.

As an example, we can state the following problem, due in its original form to Chrobak and Larmore. A downhill skier, having inadvertently missed his turn and plunged off a cliff, finds himself in the afterlife, still wearing his skis. The afterlife consists of a completely flat, foggy, snowy 1-dimensional space, in which the skier is standing at Cartesian coordinate zero. A supernatural being appears, states that a door into Heaven is located somewhere in the one-dimensional space, and then disappears again. What strategy should the skier adopt to find the doorway with the minimum amount of tedious cross-country poling? (Because of the fog, the skier has to actually ski right up to the doorway to find it.) The optimal strategy is to head right to the doorway, but the skier does not know in which of the two directions the doorway is located. But, if the skier adopts the strategy of moving in one direction for 1 time unit, then returning to the start, then going in the other direction for two time units, then returning, then going in the first direction for four time units, and so forth, the skier will eventually find the doorway having traveled no more than 9 times the optimal distance. Hence this strategy is 9-competitive.

More geometrical papers concern a robot traveling in a two-dimensional plane that contains obstacles. An obstacle is a closed two-dimensional curve (i.e., a square or circle). The robot must find a path to the target (the location searched for) that avoids all the obstacles. Generally it is assumed that the robot knows its initial position and the location of the target, in Cartesian coordinates. An obstacle is discovered only when the robot runs into it.

In the model introduced by Blum *et al.* [Blum et al., 1991] it is assumed that the robot is told the dimensions of an obstacle once it encounters the obstacle, and that the robot has perfect position information. A navigation algorithm is called $c(n)$-competitive if the length of the path traveled by the robot is no more than $c(n)$ times the length of the shortest obstacle-avoiding path. In brief, it is known that if the obstacles are axis-parallel simple rectangles then there are $O(\sqrt{n})$-competitive navigation algorithms (and slightly better randomized algorithms [Berman et al., 1996]). If the obstacles can be concave, however, then even if the sides of obstacles are axis parallel, no algorithm is better than $\Omega(n)$-competitive.

A somewhat different model and complexity measure has been adopted by the theoretical robotics community [Lumelsky and Stepanov, 1987]. Here the obstacles can be bounded by arbitrary curves, and the shape of the obstacle must be discovered by traversing its boundary. The

distance traveled by the robot is compared against the sum, P, of the perimeters of the obstacles. It is shown that in the worst-case any algorithm must travel distance at least P. How close the robot can come to that bound depends on how much knowledge the robot has about its position. With complete, exact knowledge of Cartesian coordinates there is an algorithm that never travels more than $2P$ [Lumelsky and Stepanov, 1987]. With only partial position information (perhaps more interesting in practice) the robot multiplicative factor varies from 3 to $O(\log n / \log \log n)$. For references to more recent work see [Angluin et al., 1996].

6.5 Graph Theory

One can create an on-line version of almost any graph-theoretic problem by requiring that the graph be revealed in an on-line fashion. This can mean that the vertex set is known initially and edges of the graph are added one-at-a-time, or that nothing is known initially, and at each step a vertex plus all edges to already revealed nodes are given. In addition, one can allow node or vertex deletions in an on-line fashion. On-line algorithms for graph problems can often serve as simple approximation algorithms for problems that are hard to solve optimally, such as the Steiner tree problem in networks.

In the on-line matching problem, at each step a vertex is revealed, along with all its edges to previously revealed vertices. The new vertex can be matched to some previously revealed vertex, or it can be left unmatched in the expectation that some vertex to be revealed subsequently can be matched to the current vertex. Once the choice has been made, however, it cannot be unmade. Node and edges are never deleted. The matching constructed on-line is compared to the maximum matching in the full graph. No constant is allowed in the competitive ratio. One can also ask to construct a matching of maximum weight, when each edge has a weight. For unweighted matching, [Karp et al., 1990] low constant competitive ratios are achievable. For example, the simple greedy strategy, which matches each vertex as it arrives, if possible, is 2-competitive. Weighted matching [Kalyanasundaram and Pruhs, 1993] is harder.

A generalized version of the on-line Steiner tree problem (see Section 3) in which nodes must be connected with multiple paths, is examined in [Awerbuch et al., 1996]. The on-line traveling salesman problem is examined in [Ausiello et al., 1995].

Substantial research has been done on coloring a graph on-line (see for example [Halldórsson, 1992]). The nodes and incident edges are revealed one at a time, and as each node is revealed it must be

colored so that no adjacent vertex has the same color. The goal is to minimize the number of colors, colored.

6.6 Scheduling and Load Balancing

Many scheduling and load-balancing problems are essentially on-line in nature. Scheduling problems arise, for example, in a computer operating system when users present tasks to be run on the system, and the tasks must be scheduled without knowledge of future task arrivals. An example of a load balancing problem is when a program running on a parallel machine spawns processes in an unpredictable way, and to optimize the program's performance, the processes must be partitioned on-line among the machine's processors.

On-line scheduling and load balancing is an active field of study. Representative topics are on-line scheduling of sequential jobs on parallel machines to minimize the makespan (maximum completion time) [Shmoys et al., 1991], scheduling parallel jobs on parallel machines [Feldmann et al., 1993], load balancing when each job can be handled only by a subset of the machines and we wish to minimize the maximum load [Azar et al., 1992] or the L_p norm of the machines [Awerbuch et al., 1995], preemptively scheduling jobs in a single processor to minimize the flow time, or average waiting time of jobs [Motwani et al., 1993], scheduling to minimize the average waiting time of precedence-constrained jobs [Hall et al., 1996]. Work in this area has had an impact on virtual-circuit routing (e.g. [Aspnes et al., 1993]) and algorithms for maximum-flow [Phillips and Westbrook, 1993].

To give a sense of the typical approach to on-line scheduling and load balancing problems, we present a single problem in some detail. Consider a system consisting of m identical machines. A set of jobs is presented to the system, and a scheduler must assign each job to a machine, so as to minimize the maximum load. Here the load of a machine is the sum of the weights of jobs assigned to it. In perhaps the first on-line treatment of a scheduling problem, Graham observed that the *list processing* algorithm, which assigns each job in turn to the machine whose current load is smallest, has a competitive ratio of $2 - 1/m$.

The argument is follows. Suppose a collection of jobs has been assigned to the machines using Graham's algorithm. Let i be the most loaded machine, j be the job that was last assigned to machine i, let s be its size, and let w be the load on machine i just before job j was assigned. The algorithm's maximum load is $s + w$. Let Opt be the maximum load of the optimal algorithm.

Clearly

$$\text{Opt} \geq s. \tag{9}$$

In addition, the total weight of all jobs must be at least $mw + s$, because at the time job j was assigned to machine i, all other machines must have had load at least w. Therefore

$$\text{Opt} \geq w + s/m. \tag{10}$$

Combining these inequalities gives

$$(2 - 1/m)\text{Opt} \geq w + s, \tag{11}$$

which proves that list processing is $2 - 1/m$ competitive. This bound is shown to be tight by the job sequence consisting of $m(m-1)$ jobs of size 1 followed by a single job of size m.

Recently attention has been focused on the problem of determine the best-possible competitive ratio for this problem (see [Albers, 1997], and the references therein). Currently the best known bounds are due to Albers, who demonstrates an algorithm with competitive ratio at most 1.923, and proves that no algorithm can be better than 1.852-competitive.

6.7 Finance

A natural application area for on-line methods is finance. The worst-case nature of competitive analysis offers an appealing alternative to methods of mathematical finance which are based on detailed probabilistic models approximating the behaviour of economic variables. On the other hand, the worst-case nature of competitive analysis is also problematic: if the markets are truly adversarial, the best investment strategy might be never to invest (and instead spend it all skiing). Therefore research in this area has focused on creating on-line models that, while remaining worst-case, allow some realistic constraints to be included. Raghavan's statistical adversary model [Chou et al., 1995, Raghavan, 1992] allows an adversary to pick any input sequence (for example a sequence of daily share prices) as long as the sequence exhibits a particular statistical property (for example, a mean value within some range). Al-Binali [al Binali, 1997] introduces a competitive risk-reward framework, in which an algorithm uses a particular forecast (for example, that interest rates will rise 1% within 6 months). This framework distinguishes between the restricted competitive ratio, which is the competitive ratio on inputs for which the forecast is correct, and the unrestricted competitive ratio. Let OC be the best possible unrestricted competitive ratio.

The reward of the algorithm is the ratio of its restricted competitive ratio to OC, while the risk is the ratio of the algorithm's unrestricted competitive ratio to OC. The risk measures how badly the algorithm does when the forecast is wrong, the reward quantifies the benefit when the forecast is right.

7 Research Issues and Summary

As we have suggested in our Trail Map, on-line problems are ubiquitous in computer science. There is much research into on-line algorithms that is called by other names, an important research goal is to further integrate competitive analysis into these areas.

The foundational questions in the study of on-line algorithms concern the right model and right measure of goodness. Designing algorithms to have good competitive ratios is a useful exercise that brings additional insight into the problem at hand. Since the standard competitive ratio is a worst-case measure, however, it may sometimes be too pessimistic, and variants of the standard competitive analysis have been proposed to address this issue. An important avenue of research is to provide a unifying framework for these variants, and to better understand when each variant it is likely to be a good predictor of actual performance.

Such research is both empirical and theoretical. As described in Section 4, the paging problem has become a proving ground both for empirical studies and for attempts to refine competitive analysis to provide a measure that better distinguishes between algorithms that have substantially different behavior in practice, and to capture situations in which worst-case adversaries are very unlikely. A skier doesn't have to use that same pair of skis on all terrain; she can buy different equipment for powder, moguls, ice, or slush.

The classic open problem in competitive analysis is resolving the "k-server conjecture" (see Section 5). Although substantial progress has been made on this problem a tantalizing gap still remains.

8 Defining Terms

On-line problem: A problem in which an algorithm receives a sequence of inputs, and must process each input in turn, without detailed knowledge of future inputs.

Off-line problem: A decision problem in which an algorithm is given the entire sequence of inputs

in advance.

On-line algorithm: An algorithm that solves an on-line problem.

Competitive analysis: A performance analysis in which an on-line algorithm is evaluated by comparing its performance to the best that could have been achieved if all the inputs had been known in advance.

Optimal cost: The minimum cost to process an input sequence.

Competitive ratio: The worst-case ratio between the cost incurred by an on-line algorithm and the optimal cost.

Randomized algorithm: An algorithm that uses random bits to make decisions.

Adversary: The input sequence can be thought of as being generated by an adversary that uses information about the past moves of the on-line algorithm to choose inputs that maximize the ratio between the cost to the algorithm and the optimal cost.

References

al Binali, S. (1997). The competitive analysis of risk taking with application to online trading. In *36th Annual Symposium on Foundations of Computer Science.*

Albers, S. (1997). Better bounds for online scheduling. In *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing*, pages 130–139, El Paso, Texas.

Albers, S. and Mitzenmacher, M. (1996). Average case analyses of list update algorithms, with applications to data compression. In *Proc. of the 23rd International Colloquium on Automata, Languages and Programming, Springer Lecture Notes in Computer Science, Volume 1099*, pages 514–525.

Angluin, D., Westbrook, J., and Zhu, W. (1996). Robot navigation with range queries. In *Proc. 28th ACM Symposium on the Theory of Computing*, pages 469–478.

Aspnes, J., Azar, Y., Fiat, A., Plotkin, S., and Waarts, O. (1993). On-line load balancing with applications to machine scheduling and virtual circuit routing. In *Proc. 25th ACM Symposium on the Theory of Computing*, pages 623–631.

Ausiello, G., Feuerstein, E., Leonardi, S., Stougie, L., and Talamo, M. (1995). Competitive algorithms for the on-line traveling salesman problem. In *Proceedngs of International Workshop on Algorithms and Data Structures*, volume 955 of *Lecture Notes in Computer Science*.

Awerbuch, B., Azar, Y., and Bartal, Y. (1996). On-line generalized steiner problem. In *Proc. of 7th ACM-SIAM Symposium on Discrete Algorithms*, pages 68–74.

Awerbuch, B., Azar, Y., Grove, E. F., Kao, M.-Y., Krishnan, P., and Vitter, J. S. (1995). Load balancing in the L_p norm. In *36th Annual Symposium on Foundations of Computer Science*, pages 383–391, Milwaukee, Wisconsin. IEEE.

Awerbuch, B., Azar, Y., and Plotkin, S. (1993). Throughput-competitive online routing. In *34th IEEE Symposium on Foundations of Computer Science*. 32–40.

Azar, Y., Broder, A., and Karlin, A. (1992). On-line load balancing. In *Proc. 33rd IEEE Symposium on Foundations of Computer Science*, pages 218–225. To appear in *Theoretical Compute Science*.

Bartal, Y., Blum, A., Burch, C., and Tomkins, A. (1997a). A polylog(n)-competitive algorithm for metrical task systems. In *Proc. 29th ACM Symposium on Theory of Computing*, pages 711–719.

Bartal, Y., Charikar, M., and Indyk, P. (1997b). On page migration and other related task systems. In *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 43–52, New Orleans, Louisiana.

Bartal, Y., Fiat, A., and Rabani, Y. (1992). Competitive algorithms for distributed data management. In *Proc. of the 24th Symposium on Theory of Computation*, pages 39–48.

Belady, L. (1966). A study of replacement algorithms for virtual storage computers. *IBM Systems Journal*, 5:78–101.

Ben-David, S., Borodin, A., Karp, R., Tardos, G., and Widgerson, A. (1990). On the power of randomization in on-line algorithms. In *Proc. 22nd Symposium on Theory of Algorithms*, pages 379–386.

Bentley, J. L. and McGeoch, C. C. (1985). Amortized analyses of self-organizing sequential search heuristics. *Communications of ACM*, 28(4):404–411.

Berman, P., Blum, A., Fiat, A., Karloff, H., Rosén, A., and Saks, M. (1996). Randomized robot navigation algorithms. In *Proc. 7th ACM-SIAM Symp. on Discrete Algorithms*, pages 75–84.

Black, D. L. and Sleator, D. D. (1989). Competitive algorithms for replication and migration problems. Technical Report CMU-CS-89-201, Department of Computer Science, Carnegie-Mellon University.

Blum, A., Raghavan, P., and Schieber, B. (1991). Navigating in unfamiliar geometric terrain. In *Proc. 23rd STOC*, pages 494–504.

Borodin, A. and El-Yaniv, R. (1998). *Online algorithms and competitive analysis*. Cambridge University Press.

Borodin, A., Linial, N., and Saks, M. (1987). An optimal online algorithm for metrical task systems. In *Proc. 19th Annual ACM Symposium on Theory of Computing*, pages 373–382.

Chou, A., Cooperstock, J., El-Yaniv, R., Klugerman, M., and Leighton, T. (1995). The statistical adversary allows optimal money-making trading strategies. In *Proceedings of the 6th Annual ACM-SIAM Symposium on Discrete Algorithms*.

Chrobak, M., Karloff, H., Payne, T. H., and Vishwanathan, S. (1991). New results on server problems. *SIAM Journal on Discrete Mathematics*, 4:172–181. Also in Proceedings of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms, San Francisco, 1990, pp. 291-300.

Chrobak, M. and Larmore, L. L. (1992). The server problem and on-line games. In *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, volume 7, pages 11–64.

Chung, F. R. K., Hajela, D. J., and Seymour, P. D. (1985). Self-organizing sequential search and hilbert's inequality. In *Proc. 17th Annual Symposium on the Theory of Computing*, pages 217–223.

Cormen, T., Leiserson, C., and Rivest, R. (1990). *Introduction to Algorithms*. McGraw-Hill, New York, NY.

Dowdy, L. W. and Foster, D. V. (1982). Comparative models of the file assignment problem. *ACM Computing Surveys*, 14(2):287–313.

Feldmann, A., Kao, M. Y., Sgall, J., and Teng, S. H. (1993). Optimal online scheduling of parallel jobs with dependencies. In *Proc. 25th ACM Symposium on Theory of Computing*, pages 642–651. ACM.

Fiat, A. and Karlin, A. (1995). Randomized and multipointer paging with locality of reference. In *Proc. 27th ACM Symposium on Theory of Computing*, pages 626–634.

Fiat, A., Karp, R., Luby, M., McGeoch, L. A., Sleator, D., and Young, N. (1991). Competitive paging algorithms. *Journal of Algorithms*, 12:685–699.

Fiat, A. and Mendel, M. (1997). Truly online paging with locality of reference. In *Proc. 38th Symposium on Foundations of Computer Science (FOCS)*.

Fiat, A. and Woeginger, G. (1998). *Survey papers on online algorithms and competitive analysis*. To appear.

Franaszek, P. and T.J.Wagner (1974). Some distribution-free aspects of paging performance. *Journal of the ACM*, 21:31–39.

Garay, J. and Gopal, I. (1992). Call preemption in communication networks. In *Proc. Infocom 1992*.

Garay, J., Gopal, I., Kutten, S., Mansour, Y., and Yung, M. (1993). Efficient online call control algorithms. In *Proc. 2nd Israel Symposium on Thoery of Computing and Systems*, pages 285–293.

Gawlick, R., Kamath, A., Plotkin, S., and Ramakrishnan, K. (1994). Routing and admission control of virtual circuits in general topology networks. Technical Report BL011212-940819-19TM, AT&TBell Laboratories.

Gonnet, G. H., Munro, J. I., and Suwanda, H. (1979). Towards self-organizing linear search. In *Proc. 19th Annual IEEE Symposium on Foundations of Computer Science*, pages 169–174.

Graham, R. L. (1966). Bounds for certain multiprocessing anomalies. *Bell System Technical Journal*, 45:1563–1581.

Hall, L., Shmoys, D., and Wein, J. (1996). Scheduling to minimize average completion time: Off-line and on-line algorithms. In *Proc. of 7th ACM-SIAM Symposium on Discrete Algorithms*, pages 142–151.

Halldórsson, M. M. (1992). Parallel and on-line graph coloring algorithms. In *Proc. 3rd Int. Symp. on Algorithms and Computation*, pages 61–70. Lecture Notes in Computer Science, Springer-Verlag.

Imase, M. and Waxman, B. M. (1991). Dynamic steiner tree problem. *SIAM J. Discrete Math.*, 4:369–384.

Irani, S. and Karlin, A. (1996). Online computation. In Hochbaum, D., editor, *Approximation algorithms*, pages 521–564. PWS, New York.

Irani, S., Karlin, A., and Phillips, S. (1992). Strongly competitive algorithms for paging with locality of reference. In *3rd Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 228–236.

Kalyanasundaram, B. and Pruhs, K. (1993). Online weighted matching. *Journal of Algorithms*, 14:478–488. Preliminary version appeared in SODA '91.

Karlin, A., Li, K., Manasse, M., and Owicki, S. (1991). Empirical studies of competitive spinning for shared memory multiprocessors. In *Proc. 13th ACM Symposium on Operating Systems Principles*.

Karlin, A., Manasse, M., Rudolph, L., and Sleator, D. (1988). Competitive snoopy caching. *Algorithmica*, 3(1):79–119.

Karlin, A., Phillips, S., and Raghavan, P. (1992). Markov paging. In *Proc. 33rd IEEE Symposium on Foundations of Computer Science*, pages 208–217.

Karp, R. M., Vazirani, U. V., and Vazirani, V. V. (1990). An optimal algorithm for on-line bipartite matching. In *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing*, pages 352–358, Baltimore, Maryland.

Keshav, S., Lund, C., Phillips, S., Reingold, N., and Saran, H. (1995). An empirical evaluation of virtual circuit holding time policies in IP-over-ATM networks. *IEEE Journal on Selected Areas in Communications*, 13(8):1371–1382.

Koutsoupias, E. and Papadimitriou, C. (1994a). On the k-server conjecture. In *Proc. 25th Symposium on Theory of Computing*, pages 507–511.

Koutsoupias, E. and Papadimitriou, C. H. (1994b). Beyond competitive analysis. In *35th Annual Symposium on Foundations of Computer Science*, pages 394–400, Santa Fe, New Mexico. IEEE.

Lewis, P. and Shedler, G. (1973). Empirically derived models for sequences of page exceptions. *IBM J. Res. and Develop.*, 17:86–100.

Luce, R. D. and Raiffa, H. (1957). *Games and Decisions*. John Wiley and Sons.

Lumelsky, V. J. and Stepanov, A. A. (1987). Path-planning strategies for a point mobile automaton moving amidst unknown obstacles of arbitrary shape. *Algorithmica*, 2:403–430.

Lund, C., Phillips, S., and Reingold, N. (1994a). IP over connection-oriented networks and distributional paging. In *35th IEEE Symposium on Foundations of Computer Science*, pages 424–435.

Lund, C., Reingold, N., Westbrook, J., and Yan, D. (1994b). On-Line Distributed Data Management. In *Proceedings of the 2nd Annual European Symposium on Algorithms, ESA '94*, volume 855 of *Lecture Notes in Computer Science*, pages 202–214, Utrecht, The Netherlands. Springer-Verlag.

Manasse, M., McGeoch, L. A., and Sleator, D. (1988). Competitive algorithms for online problems. In *Proc. 20th Annual ACM Symposium on Theory of Computing*, pages 322–333.

McGeoch, L. and Sleator, D. (1991). A strongly competitive randomized paging algorithm. *J. Algorithms*, 6:816–825.

Motwani, R., Phillips, S., and Torng, E. (1993). Non-clairvoyant scheduling. In *Proc. 4th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 422–431. Also to appear in *Theoretical Computer Science*, Special Issue on Dynamic and On-Line Algorithms.

Phillips, S. and Westbrook, J. (1993). Online load balancing and network flow. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing*, pages 402–411, San Diego, California.

Plotkin, S. (1995). Competitive routing of virtual circuits in atm networks. *IEEE J. Selected Areas in Comm.*, pages 1128–1136. Special issue on Advances in the Fundamentals of Networking.

Raghavan, P. (1992). A statistical adversary for on-line algorithms. DIMACS *Series in Discrete Mathematics and Theoretical Computer Science*, 7:79–83.

Rivest, R. (1976). On self-organizing sequential search heuristics. *Communication of the ACM*, 19:63–67.

Shedler, G. and Tung, C. (1972). Locality in page reference strings. *sicomp*, 1:218–241.

Shmoys, D. B., Wein, J., and Williamson, D. P. (1991). Scheduling parallel machines on-line. In McGeoch, L. A. and Sleator, D. D., editors, *On-line Algorithms*, volume 7 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 163–166. AMS/ACM.

Sleator, D. and Tarjan, R. E. (1985a). Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28:202–208.

Sleator, D. D. and Tarjan, R. E. (1985b). Self-adjusting binary search trees. *J. Assoc. Comput. Mach.*, 32:652–686.

Tarjan, R. E. (1983). *Data Structures and Network Algorithms*. SIAM, Philadelphia.

Teia, B. (1993). A lower bound for randomized list update algorithms. *Information Processing Letters*, 47:5–9.

Torng, E. (1995). A unified analysis of paging and caching. In *36th IEEE Symposium on Foundations of Computer Science*, pages 194–203.

von Neumann, J. and Morgenstern, O. (1947). *Theory of Games and Economic Behavior*. Princeton University Press.

Westbrook, J. and Zuck, L. (1994). Adaptive algorithms for paso systems. In *Proc. ACM Symp. on Principles of Distributed Computing (PODC 94)*, pages 264–273.

Yao, A. C. (1980). Probabilistic computations: Towards a unified measure of complexity. In *Proc. 12th ACM Symposium on Theory of Computing*.

Young, N. (1991). The k-server dual and loose competitiveness for paging. *Algorithmica*. To appear. Rewritten version of "On-line caching as cache size varies", in The 2nd Annual ACM-SIAM Symposium on Discrete Algorithms, 241-250, 1991.

Young, N. E. (1994). The k-server dual and loose competitiveness for paging. *Algorithmica*, 11(6):525–541.

9 Further Information

Irani and Karlin [Irani and Karlin, 1996] provide a short survey of the field from a more theoretical viewpoint, while the book of Borodin and El-Yaniv [Borodin and El-Yaniv, 1998] is an extensive

treatment of on-line algorithms. A collection of surveys of various subfields has been edited by Fiat and Woeginger [Fiat and Woeginger, 1998]. Marek Chrobak and John Noga have collected a bibliography of papers on on-line problems, available at

`http://www.cs.ucr.edu/~marek/pubs/online.bib`.

We have extended that bibliography with recent papers and our revised version is available at

`http://www.research.att.com/~phillips/online.bib`.

Current research into on-line problems and algorithms is published in a number of conferences and journals. Currently there is a lag time of several years between journal submission and final publication, so conferences provide the forum for fast dissemination of cutting-edge research. A partial list of larger conferences includes the ACM Symposium on Theory of Computing (STOC), the IEEE Conference on Foundations of Computer Science (FOCS), the ACM-SIAM Symposium on Discrete Algorithms (SODA), the International Colloquium on Automata, Languages, and Programming (ICALP), and the European Symposium on Algorithms (ESA). There are also numerous regional conferences. Announcements about conferences can generally be found in the *Communications of the ACM* or *IEEE Spectrum*.

A partial list of journals includes *Journal of the ACM*, *SIAM Journal on Computing*, *Journal of Algorithms*, *Algorithmica*, *Information Processing Letters*, and *Theoretical Computer Science*.