Competitive Randomized Algorithms for Non-Uniform Problems

Anna R. Karlin*
Mark S. Manasse*
Lyle A. McGeoch†

Susan Owicki*

Abstract

Competitive analysis is concerned with comparing the performance of on-line algorithms with that of an optimal off-line algorithm. For some problems, randomized on-line algorithms have yielded better performance ratios than deterministic on-line algorithms, assuming that the input sequences are generated by an adversary that has no knowledge about the results of the coin tosses made by the algorithm. In this paper, we present new randomized on-line algorithms for snoopy-caching and the spin-block problem. These algorithms achieve strongly competitive ratios approaching $e/(e-1) \approx 1.58$, a surprising improvement over the best possible ratio in the deterministic case, which is 2. We also consider the situation when the request sequences for these problems are generated according to an unknown probability distribution. In this case, we show that deterministic algorithms that adapt to the observed request statistics also pay at most a factor of $e/(e-1)$ more than the optimal off-line algorithm. Finally, we show that for the 2-server problem on a 3-vertex isosceles triangle, there is a lower bound on the competitive ratio with a limit of $e/(e-1)$. This is in contrast to the 2-server problem on an equilateral triangle where a strongly 3/2-competitive randomized algorithm is known.

1 Motivation and Results

The amortized analysis of on-line algorithms for processing sequences of tasks in dynamic systems has been a subject of great interest in recent years. The approach taken is to compare the performance of a strategy that operates with no knowledge of the future with that of an optimal, clairvoyant strategy, that has complete knowledge of the future and operates optimally given that information. A large number of problems have been studied from this point of view, *cf* [1, 2, 3, 4, 5, 7, 8, 9, 10, 11, 12, 13, 14].

On-line algorithms whose performance is within the smallest possible constant factor of the optimum off-line are said to be ***strongly competitive***. More formally, let A be a (possibly randomized) on-line algorithm, let σ be an input sequence to the algorithm, and let $\mathbf{E}C_A(\sigma)$ be the expected cost A incurs when processing input sequence σ. Let $C_{opt}(\sigma)$ be the cost incurred by the optimal off-line algorithm in processing σ. We consider two types of adversaries.

Our first type of adversary is one that makes its request sequence without regard to the nondeterministic choices made by the on-line algorithm. An on-line algorithm A is said to be ***c-competitive against a weak adversary*** if there is a constant a such that for any fixed input sequence σ,

$$\mathbf{E}C_A(\sigma) \leq c \cdot C_{opt}(\sigma) + a.$$

The constant c is known as the ***competitive factor***.

The second type of adversary is one that can choose each input request depending on the choices made by the algorithm in servicing the previous requests. An on-line algorithm A is said to be ***c-competitive against a strong adversary*** if there is a constant a such that for any input sequence σ generated in this way,

$$\mathbf{E}C_A(\sigma) \leq c \cdot C_{opt}(\sigma) + a.$$

Finally, an algorithm is ***strongly c-competitive against a weak (resp. strong) adversary*** if c is the smallest constant attained by any on-line algorithm.

Observe that if the on-line algorithm is deterministic then a weak adversary can simulate a strong one.

A natural and interesting question is whether there are problems for which the best competitive factor is smaller against weak adversaries than against strong

*DEC Systems Research Center, 130 Lytton Ave., Palo Alto, CA 94301.
†Dept. of Mathematics and Computer Science, Amherst College, Amherst, MA 01002.

adversaries. Two dramatic results of this type have been obtained. The first, due to Borodin, Linial and Saks [3] shows that for an n-state metrical task system where all states are unit distance apart the competitive factor can be improved from $O(n)$ against a strong adversary to $2\mathcal{H}_n$ ($= O(\log n)$) against a weak adversary. A similar result due to McGeoch and Sleator [12] shows that for the paging problem with memory size k (or equivalently a k-server problem with uniform distances), the competitive factor can be improved from k to $\mathcal{H}_k$. Fiat *et al* [7] showed that this is the best possible competitive factor against a weak adversary.

In this paper we present new randomized algorithms for the snoopy-caching and spin-block problems. For both of these problems, it is not possible to construct an on-line algorithm with a competitive ratio better than 2 against a strong adversary. Our randomized algorithms have competitive ratios which approach $e/(e-1) \approx 1.58$. We also show that this is best possible against a weak adversary. An interesting fact about these results is that they are the first strongly-competitive algorithms against a weak adversary for a server problem on a non-uniform graph.

We also present a lower bound result for the 2-server problem. It is known that for the 2-server problem on a graph with all distances equal, there is a strongly 3/2-competitive algorithm against a weak adversary. We show that such a competitive ratio cannot be achieved on a graph with unequal distances. In particular for the 2-server problem on a 3-vertex isosceles triangle, there is a lower bound on the competitive ratio greater than 1.5 and tending to $e/(e-1)$. We also show a lower bound greater than 1.54 on the competitive ratio for the 2-server problem on the 3-4-5 triangle.

Finally, we consider the question of how well a deterministic algorithm can perform if the input sequence is generated according to some unknown but time-independent probability distribution. This question was motivated by the observation that in traces obtained for programs running on snoopy caching multiprocessor systems [6], different programs exhibit vastly different input characteristics (in this case write-run length). In particular, the inter-program input variability was extremely high, while the intra-program input variability was extremely low. This suggests that an algorithm which adapts to the observed input statistics can potentially converge to near-optimal behavior.

We show that adaptive deterministic algorithms of this type for snoopy-caching and for the spin-block problem achieve competitive ratios approaching $e/(e-1)$ if the input sequence is generated according to a fixed probability distribution. We further present a practical and simple version of the adaptive algorithm which is also 3-competitive against a strong adversary.

2 Snoopy Caching

2.1 The Model

A snoopy caching multiprocessor system is a system in which a set of processors each with its own (snoopy) cache are connected over a bus to each other and to a large shared memory. We will assume that the caches and large shared memory have infinite capacity. There is a single address space used by all of the processors; each location in this space is called a *variable*. The memory space is partitioned into *blocks*, groups of variables of uniform size. We let $p-1$ denote the block size. We define the *block snoopy caching* model, as in [9].

All of a processor's memory requests are serviced by its cache. The cost of reading a variable depends on whether the block containing that variable is in the requesting cache or not. If it is, then the read is executed at no cost. If the block isn't in the cache, then the cache must send out a request for the appropriate block. This block is then broadcast on the bus by the cache currently containing the value of the block, and all other caches listen to the bus and retrieve it. The cost of this read is p bus cycles (one cycle for the address and one cycle for each variable in the block).

The cost of a write is either 1 or 0 bus cycles, depending on whether the block is shared or not. If the block is shared, the new value and address of the variable being written to must be broadcast on the bus to maintain cache consistency. We will assume that every write is preceded by a read, since if the cache is holding the block it wishes to write to, this read is free.

We call an on-line algorithm that decides when a block should be invalidated a *snoopy-caching algorithm*. The goal of this algorithm is to minimize the number of bus cycles used. For efficiency, if a block is shared, but only actively used by one cache, the other caches should invalidate that block in order to eliminate the cost of doing updates. On the other hand, there is a large penalty of p bus cycles for invalidating a block from a cache which shortly thereafter needs to access it. Previously, Karlin *et al* [9] discovered a strongly 2-competitive algorithm against a strong adversary. We will now show that it is possible to improve this ratio if the on-line algorithm is randomized and the adversary is weak.

2.2 Randomized Algorithms for Snoopy Caching

For any constant p, define an associated constant e_p as follows:

$$e_p = \left(1 + \frac{1}{p}\right)^p .$$

Theorem 1 *Consider a block snoopy caching multiprocessor system, with page size $p-1$. There is an on-line randomized snoopy caching algorithm A with a competitive factor of*

$$\frac{e_p}{e_p - 1} \longrightarrow_{p\to\infty} \frac{e}{e-1}$$

against a weak adversary. In fact, if all blocks start out shared, the expected cost of A' on any sequence equals $e_p/(e_p-1)$ times the optimal cost.

Proof: Assume initially that the multiprocessor system consists of two caches and a single block B. We may assume that initially block B is shared by both caches, since all costs incurred prior to the time this state is reached can be bounded by a constant. Moreover, if the initial states are the same for the on-line and off-line algorithms, this constant is 0.

Let σ_k be any sequence of requests consisting of k writes by one of the caches, interspersed with any number of reads by that same cache, to some variable in B, followed by a read by the other cache of some variable in B. We call such a sequence a *write run* of length k. Since any algorithm need not invalidate B from either cache until it is written to by the other cache, we can ignore any read subsequent to the one terminating a write run. Hence any sequence σ of requests can be decomposed into subsequences of type σ_k and so a randomized algorithm that achieves a competitive factor of α on any sequence σ_k achieves a competitive factor of α overall.

We construct a randomized algorithm for sequences of type σ_k with the best competitive factor. The choices available to the on-line algorithm for processing sequence σ_k can be described by enumerating a set of deterministic algorithms. Let A_i be the deterministic algorithm that drops the block from the inactive cache after i updates by the active cache. A randomized algorithm is just a choice of a probability distribution π, where π_i is the probability that the randomized algorithm chooses algorithm A_i on any write-run starting from the shared situation.

We observe that the expected cost of algorithm A on sequence σ_k is

$$\mathbf{E}(C_A(\sigma_k)) = \sum_{1\le i\le k} \pi_i(p+i) + (1 - \sum_{1\le i\le k} \pi_i)k.$$

Since the optimal off-line algorithm has

$$C_{opt}(\sigma_k) = \begin{cases} k & k \le p \\ p & k > p \end{cases}$$

our goal is to choose values for π_i such that

$$\begin{cases} \mathbf{E}(C_A(\sigma_k)) \le (1+\alpha)k & k \le p \\ \mathbf{E}(C_A(\sigma_k)) \le (1+\alpha)p & k > p \end{cases} \qquad (*)$$

and α is minimized.

Setting the preceding inequalities to equalities and solving the resulting difference equations, we obtain

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

Solving for α by setting $\sum_{1\le i\le p} \pi_i = 1$, we get

$$\alpha = \frac{1}{\left(1+\frac{1}{p}\right)^p - 1} = \frac{1}{e_p - 1}.$$

Therefore this probabilistic algorithm yields a competitive factor of

$$1 + \alpha = \frac{e_p}{e_p - 1} \longrightarrow \frac{e}{e-1} \approx 1.58.$$

To extend this analysis to multiple blocks, we take the on-line algorithm which treats each block independently as above. This yields an on-line algorithm A for which

$$C_A - \frac{e_p}{e_p - 1} C_{opt} \le c,$$

where c is a constant multiple of the discrepancy between the initial states of the on-line and off-line algorithms.

Finally, we describe the on-line algorithm for k caches, which we call A'. At the beginning of each write run, algorithm A' selects a value i according to the probabilities π_i. It broadcasts updates until the write run exceeds i in length, and if this happens, invalidates the block in all other caches.

To see that A' achieves the same competitive factor, we define a mapping between request sequences, σ, on k caches and request sequences σ' on 2 caches as follows. Partition σ into subsequences τ_i consisting of operations by a single cache. The associated sequence

σ' consists of the concatenation of the sequences τ'_i where τ'_i is the same set of reads and writes as in τ_i, but with the requesting cache replaced by $i \pmod 2$. Trivially $C_{A'}(\sigma) \leq C_A(\sigma')$ and $C_{opt}(\sigma) \geq C_{opt}(\sigma')$, and the theorem is proved. □

Theorem 2 *Let B be any on-line snoopy caching algorithm in a block snoopy caching multiprocessor system with page size p. Then there exists arbitrarily expensive sequences σ for which*

$$\frac{\mathbf{E}(C_B(\sigma))}{C_{opt}(\sigma)} \geq \frac{e_p}{e_p - 1} > \frac{e}{e-1}$$

Proof: Consider the result of executing algorithm B on a sequence of type σ_k. Suppose that algorithm B has probability α_i of dropping the block from the inactive cache after i writes. Take π_i and algorithm A as above. Let k be the smallest index for which $\sum_{1 \leq i \leq k} \alpha_i \leq \sum_{1 \leq i \leq k} \pi_i$. Since the π_i values were chosen so as to set inequalities $(*)$ to equalities, it is easy to verify that $\mathbf{E}C_B(\sigma_k) \geq \mathbf{E}C_A(\sigma_k)$, After each write run, select another k as above to extend the sequence and complete the proof. □

2.3 Randomized Algorithms for Limited Block Snoopy Caching

In this section, we extend the results for block snoopy caching to the more realistic model of finite direct-mapped caches. The slots that may contain blocks in a cache are called *cache lines.* A *direct-mapped* cache i uses a hash function $h_i(B)$ to determine the unique cache line in which a block B will reside. If $h_i(B) = h_i(B')$, then cache i can contain at most one of the blocks B and B' at any time. Since different blocks can occupy the same cache line, a block that is read into a line may displace a block which was written to privately. In this case, the line is *dirty* and must be *written back* to main memory at a cost of p. This model continues to assume that main memory has infinite capacity, a realistic assumption since it must be possible to have some clean copy of each shared variable.

In the *limited block snoopy caching* model, a cache grabs a block that is being read by another cache or being written back if and only if that block was the last to occupy its cache line; that is, only if the last operation on this cache line was to invalidate this block.

Making the assumption that the adversary has to use the same hash function as the on-line algorithm, we can prove a strengthening of Theorem 1.

Theorem 3 *Consider a limited block snoopy caching multiprocessor system, with page size $p - 1$. There is an on-line randomized snoopy caching algorithm A with a competitive factor of*

$$\frac{e_p}{e_p - 1} \xrightarrow{p \to \infty} \frac{e}{e-1}$$

against a weak adversary.

Proof: Without loss of generality we restrict our attention to a single cache line. In this case, a write run, σ_k, is any sequence of requests consisting of k writes by one of the caches, c, interspersed with any number of reads by that same cache, to some variable in B, followed either by a read by any other cache of some variable in B or a read by c to a variable in a block that collides with B. Every sequence of requests can be decomposed into its write runs and its *isolated reads*, the latter being either the result of collisions or initial reads into a cache line. The optimal off-line algorithm and our on-line algorithm incur the same cost for all of the isolated reads.

The on-line algorithm A for the limited block model uses the same probabilities as the block snooping algorithm to determine how many updates to do in a write run before invalidating.

To show that this algorithm achieves the same competitive ratio, we map any sequence σ on which it operates to a sequence σ' in which the costs for σ' and σ have the same optimal cost in the limited block model, σ' is at least as expensive as σ for the on-line algorithm, and the costs of the write runs in σ' are the same for both the block and limited block models.

Each write run in σ which terminates due to a colliding read, is replaced in σ' by the same write run followed by a read for the block by main memory. If the block was shared immediately before the colliding read, then main memory contains a valid copy of the block. Hence for both the off-line and on-line algorithms, the added read by main memory replaces the writeback; if it was necessary it costs p and if not, it's free.

Now consider a write run in σ which terminates with a read. If the line in the reading cache was dedicated to some other block, then this write run is augmented with p extra writes. We claim that adding these writes does not change the optimal algorithm's cost. Indeed, since the terminating read causes that block to become revalidated by every cache that previously held it, the optimal strategy on this write run is to invalidate that block in all caches except for the writing cache. Since we have added extra requests, the on-line cost can not decrease, and since the write run has length at least p, the cost of the write run in both models is equal to the number of steps for which the block is shared plus the cost of the re-read, p. □

2.4 Adaptive Algorithms

Traces obtained for programs running on snoopy caching multiprocessor systems show that different programs exhibit vastly different write-run characteristics. Specifically, the programs analyzed in [6] have the property that either write runs are very short (variables are actively being shared) or they are very long (there is very little sharing going on). In the first case, an algorithm like exclusive write (algorithm A_1 above) performs best; in the second case, an algorithm like pack-rat (algorithm A_∞) performs best.

Motivated by these observations, we studied the following problem. Suppose that the request sequence is generated according to some unknown probability distribution $\mathcal{P}$. How well can an on-line algorithm do?

We begin by observing that the distribution $\mathcal{P}$ is completely characterized by knowing the probability that a write run has length k. If A_i is the deterministic algorithm that drops a block from the inactive cache after i consecutive writes by the active cache, then it is obvious that the best deterministic algorithm A_i to use is that subscripted by i for which $\mathbf{E}C_{A_i}(\sigma(\mathcal{P}))$ is minimized, where $\sigma(\mathcal{P})$ is generated according to $\mathcal{P}$. Call the algorithm that minimizes this expected cost A^*.

Since in practice $\mathcal{P}$ is not known, the obvious approach is for the on-line algorithm to collect statistics on write-run lengths, and on the fly recompute which of the deterministic algorithms minimizes the cost. It is clear that if the sequence is generated from a time-independent distribution, then the sample statistics will converge to their true values and the on-line algorithm will eventually become A^*.

The following theorem characterizes the competitive ratio of A^*.

Theorem 4 *Let A^* be the deterministic algorithm that minimizes the expected cost on request sequences $\sigma(\mathcal{P})$, where $\sigma(\mathcal{P})$ is generated according to the time-independent distribution $\mathcal{P}$. Then*

$$\mathbf{E}C_{A^*}(\sigma(\mathcal{P})) \le \frac{e_p}{e_p - 1}\mathbf{E}C_{opt}(\sigma(\mathcal{P})).$$

Proof: Since for all sequences σ,

$$\sum_i \pi_i C_{A_i}(\sigma) \le \frac{e_p}{e_p - 1} C_{opt}(\sigma),$$

the average over sequences with probabilities determined by $\mathcal{P}$, yields

$$\sum_i \pi_i \mathbf{E}\left(C_{A_i}(\sigma)\right) \le \frac{e_p}{e_p - 1}\mathbf{E}\left(C_{opt}(\sigma)\right).$$

But A^* is that algorithm that minimizes $\mathbf{E}\left(C_{A_i}(\sigma)\right)$. Since a convex combination of positive elements exceeds the minimum element, we obtain

$$\mathbf{E}\left(C_{A^*}(\sigma)\right) \le \frac{e_p}{e_p - 1}\mathbf{E}\left(C_{opt}(\sigma)\right).$$

□

In practice, we feel that it should be sufficient to use the last few write runs as a sample, that is, if the last j write runs have lengths $i_1, i_2, \ldots, i_j$, then we will approximate $\mathcal{P}$ with the distribution $\mathcal{P}'$ consisting of sequences $\sigma_{i_1}, \sigma_{i_2}, \ldots, \sigma_{i_j}$, each with probability $1/j$. It is possible to show that the adaptive algorithm so obtained is still competitive against a strong adversary.

Theorem 5 *Let A' be the deterministic, on-line algorithm that minimizes the expected cost for the distribution $\mathcal{P}'$ obtained by letting $j = 1$. Then A' is 3-competitive against a strong adversary.*

Proof: The algorithm obtained by adapting to the previous sample statistic is the following. Suppose that the last write run had length l. Then the distribution $\mathcal{P}'$ assumes that write runs have length l with probability 1. Consequently, the algorithm that minimizes the expected cost uses algorithm A_p on the next write run if $l \le p$ and algorithm A_1 if $l > p$.

To see that this algorithm is 3-competitive, we simply observe that if A_p is used, then a ratio of 2 between on-line and offline costs is achieved [9]. On the other hand, if A_1 is used, then the previous write run had length greater than p, and consequently the adversary paid at least p for that write run. Hence, we can charge our cost on the current write run to the adversary's cost on the previous write run, yielding an overall competitive ratio of 3. □

The larger the number of sample statistics used, the better the algorithm will perform. In any case, we believe the combination of the strong-adversary competitiveness of this algorithm with its adaptiveness to the potential underlying request distribution makes it eminently practical.

3 Spin-Block

3.1 The problem

Consider a process that is waiting for a lock. There are two choices for the actions that may be taken: The process can *spin*, at a cost proportional to the length of time it spins, or it can *block*. The latter action has some large cost C reflecting the cost of restarting the process and restoring its state, usually referred to as the *context-switch* cost. The difficulty in solving this problem is that the minimum-cost action depends on how long it will be before the lock is freed, information that is unavailable on-line. An on-line algorithm for the spin-block problem must decide how long a process should spin before it blocks.

It is fairly obvious that the spin-block problem is a continuous version of the 2-cache, 1-block snoopy caching problem and as such it is trivial to construct a deterministic on-line algorithm with cost at most twice that of the optimal off-line algorithm—namely have the process spin for an amount of time equal to the cost of a context switch.

Theorem 6 *There is no c-competitive algorithm for the spin-block problem for $c < 2$ against a strong adversary.*

Theorem 7 *There is a simple on-line randomized algorithm A for the spin-block problem which is strongly $e/(e-1)$-competitive against a weak adversary.*

Proof: Let $\pi(t)$ be the density function of the time before a process should block using algorithm A and let σ_τ denote the situation where the lock remains held for time τ. Then the expected cost of the algorithm A that uses density function $\pi(t)$ to determine how long a process should spin before blocking is

$$\mathbf{E}C_A(\sigma_\tau) = \int_0^\tau (t+C)\pi(t)dt + \tau \int_\tau^\infty \pi(t)dt.$$

As in the case of snoopy caching, we would like to choose $\pi(t)$ so that

$$\begin{cases} \mathbf{E}(C_A(\sigma_\tau)) \le (1+\alpha)\tau & \tau \le C \\ \mathbf{E}(C_A(\sigma_\tau)) \le (1+\alpha)C & \tau > C \end{cases}$$

and α is minimized.

Setting the preceding inequalities to equalities and solving the differential equations that result from differentiating twice with respect to τ, we obtain

$$\pi(t) = \begin{cases} \frac{1}{(e-1)C} e^{t/C} & 0 \le t \le C \\ 0 & \text{otherwise} \end{cases}$$

The resulting competitive factor is easily calculated to be $e/e-1$. □

The same results that hold for the adaptive setting of snoopy caching hold for the adaptive setting of the spin-lock problem. As before, the adaptive algorithm collects statistics on spin-length times and chooses the algorithm that minimizes the expected cost.

Theorem 8 *Let A^* be the deterministic algorithm that minimizes the expected cost on lock-waiting time sequences $\sigma(\mathcal{P})$, where $\sigma(\mathcal{P})$ is generated according to a time-independent distribution $\mathcal{P}$. Then*

$$\mathbf{E}C_{A^*}(\sigma(\mathcal{P})) \le \frac{e}{e-1}\mathbf{E}C_{opt}(\sigma(\mathcal{P})).$$

An algorithm that uses sample statistics of lock-waiting times in order to estimate the distribution $\mathcal{P}$ converges to $e/(e-1)$ competitive behavior.

Proof: The proof is identical to the proof of theorem 4 with summations replaced by integrals. □

The convergence of this algorithm to $e/(e-1)$ competitive behavior depends on the fact that accurate statistics can be generated by keeping track of the entire history of lock-waiting times. A practical alternative to this algorithm, similar to that for snoopy caching, is one which only uses the last (or perhaps last few) lock-waiting times in order to determine what to do the next time a process requests a lock.

As in the snoopy caching case, the adaptive algorithm A for deciding how long a process should spin depends on the length of time τ that the lock last remained held. If $\tau < C$, then the process should spin for a time equal to C, otherwise the process should block immediately. Note once again that this is an instance of choosing the algorithm that minimizes the expected cost, under the assumption that the lock-waiting time is equal to τ with probability 1. Furthermore, algorithm A is 3-competitive against a strong adversary. The proof of this fact is virtually identical to the proof of theorem 5.

In practice, the most commonly implemented strategy is to block immediately if the lock is not available, always incurring cost C. It is desirable that an adaptive algorithm not cost substantially more than this simple strategy. Assuming a certain independence in the way processors acquire locks, we can show that competitive advantage can be traded off against a guarantee that the adaptive algorithm does not perform too much worse than the algorithm that always blocks.

We choose a constant α, with $0 \le \alpha \le 1$, that determines the competitiveness and the bound on waiting time. As before, the time that a process spins before blocking depends on τ. If $\tau < \alpha C$, then the process spins for a time equal to αC, otherwise it blocks immediately. A large value of α gives a more competitive algorithm, while a smaller value guarantees that the average waiting time is not much worse than C.

Now assume that the distribution of waiting times is nonincreasing, that is, the probability of waiting between a and b time units is at least as large as the probability of waiting between $a+k$ and $b+k$ time units. This would be the case if the time of lock requests in one process are independent of the times when locks are held by other processes, which is likely to be nearly true in many applications.

Let $f(t)$ be the density function for the distribution of waiting times, and

$$p = \int_{\alpha C}^{\infty} f(t)dt = Pr(\tau > \alpha C).$$

With probability p, the algorithm A_α blocks immediately, and with probability $1-p$ it spins for up to time αC. Thus the expected cost of waiting is

$$\mathbf{E}(C_{A_\alpha}) = pC + (1-p)\left(\int_0^{\alpha C} tf(t)dt + p(C+\alpha C)\right).$$

For fixed p, this cost is maximized (over nonincreasing distributions) when f is uniform between 0 and αC. Thus

$$\mathbf{E}(C_{A_\alpha}) \le pC + (1-p)\left((1-p)\frac{\alpha C}{2} + p(C+\alpha C)\right).$$

The value of p for which this cost is maximized can be determined by differentiation to be

$$p = \frac{2}{2+\alpha}.$$

Substituting in the previous formula and simplifying gives

$$\mathbf{E}(C_{A_\alpha}) \le C(1 + \frac{\alpha^2}{2\alpha+4}).$$

By choosing $\alpha = 0.5$, for example, the expected cost of the adaptive algorithm is bounded by $1.06C$.

4 The 2-Server Problem

It has been shown that there is a strongly $\mathcal{H}_k$-competitive algorithm against a weak adversary for the k-server problem where the distances in the graph are uniform. A natural question that arises is whether there is such an algorithm for the k-server problem where the distances are not uniform.

We answer this question in the negative by showing that for 2 servers on a 3-vertex nonequilateral triangle there is no 3/2-competitive algorithm.

Theorem 9 *Let B be any on-line randomized algorithm for the 2-server problem on a 3-vertex isosceles triangle with distances d, d, and 1. Then there exists an infinite sequence of requests σ for which*

$$\mathbf{E}(C_B(\sigma)) \ge \frac{e_{2d-1} + \frac{1}{4d}}{(e_{2d-1} - 1) + \frac{1}{2d}} \cdot C_{opt}(\sigma)$$

There is a randomized 2-server algorithm that achieves this competitive factor.

Proof: Suppose A is the best on-line algorithm. The idea of the lower bound proof is to explore the tree of possible request sequences, maintaining for each node in the tree a vector in which the i^{th} coordinate is the probability that A has a server at the i^{th} vertex.

By constructing the optimal algorithm's cost using dynamic programming, we can determine positions in the tree where we know with certainty the locations of *opt*'s servers. These are points for which the cost of being in some other state s is equal to the cost of being in the desirable state plus the cost to switch from the desirable state to the undesirable state. Call this set of positions K.

We claim that at each position in K, we may assume that the on-line algorithm A has its servers in the same locations as the off-line algorithm. If not, it is possible to augment the sequence of requests terminating at such a position (in K) with a subsequence that makes the ratio between A's cost and *opt*'s cost larger. Specifically, suppose that after each subsequence which ends at a position $k \in K$, algorithm A is covering one of *opt*'s vertices with probability less than one. By alternating requests for the vertices covered by *opt*, one of two situations will be reached. If there is always some ϵ difference between the positions of A and *opt*, then A's cost will grow arbitrarily. On the other hand, if A covers *opt*'s vertices with probability approaching 1, then A's cost is eventually larger than if it had moved to that position immediately.

Request sequence trees for certain graphs (including isosceles triangles) have the property that every sufficiently long request sequence contains a subsequence beginning and ending with positions in K in which *opt* is known to be in the same state. Suppose that α is the competitive factor achieved by the on-line algorithm. From each minimal subsequence in which *opt*'s servers begin and end in the same state, we can derive an inequality between the expected cost of the on-line algorithm and α times the cost of the off-line algorithm. The variables in these inequalities are the probabilities that A will be in a given state at a given time. The inequalities must all hold since otherwise the corresponding subsequence could be repeated to construct an infinitely long sequence with cost ratio greater than α. Our lower bound is then obtained by minimizing α subject to this set of linear constraints. The upper bound is achieved with an algorithm using the probabilities obtained from the minimization.

For isosceles triangles, inequalities can be derived as follows. Suppose G is a triangle with vertices X, Y, and Z, where the distance between X and Y is one, and where Z is distance d away from the other vertices. In order to calculate the cost of algorithms, we partition request sequences into *phases*. A new phase begins if the location of *opt*'s servers is known and if one of those servers is on Z. By the argument above, we can assume that A's servers are in the same locations at the beginning of a phase. If A can achieve a competitive factor α in general, then it must be able to achieve it in every phase. In discussing possible request sequences, we will never consider requests for vertices that A is covering with probability 1, because such requests can not increase the expected cost ratio between A and *opt*.

After each request an on-line algorithm A must choose the probabilities with which it will cover the vertices. Such probabilities can not be based on the future. Furthermore, they need not consider requests that preceded the time when A and *opt* occupied the same vertices, because such requests can not affect the ratio of costs in later phases. So the probability that a particular vertex is covered depends only on the requests in the current phase.

Let p_i be the probability that A is covering Z after the first i requests of a phase, assuming that Z has not yet been requested. We can divide the analysis of costs into two parts. Let $\sigma(i)$ denote A's expected cost on a phase that has i requests of vertices X or Y before the first request at Z.

If $i < 2d$, the off-line optimal algorithm will shuttle a server between X and Y. When the request at Z arrives, *opt* is known to be covering Z and the previous request, and the phase ends. It has a total cost of i for the phase. Because *opt*'s final configuration is known, A must also reach the same configuration. A's expected cost for the phase must then be

$$\mathbf{E}(C_A(\sigma_i)) = 2d + p_i(1-2d) + \sum_{j=1}^{i-1} p_j.$$

Alternatively, if $i \geq 2d$, *opt* will begin the phase by moving away from Z. When the request at Z arrives, *opt* will move one server to Z, leaving the other to cover whichever of X or Y is requested next. When that next request arrives, the location of *opt*'s servers is known, and the phase ends. Algorithm *opt* has a total cost of $2d$ for the phase. After the first $2d$ requests of X or Y, we know that *opt* is covering X and Y, so algorithm A must cover the same vertices. Its total expected cost for the X and Y requests is

$$d + \sum_{j=1}^{2d-1} p_j.$$

When the request at Z finally arrives, A must cover it by moving from X or Y, at a cost of d. At this point, A can not know which vertex, X or Y, is covered by *opt*. To minimize the cost ratio for the worst-case choice, it must cover X and Y with equal probability. This gives it an expected cost of 1/2 for the final request of the phase. A's total cost for the phase is

$$\mathbf{E}(C_A(\sigma_i)) = 2d + 1/2 + \sum_{j=1}^{2d-1} p_j.$$

Using these costs, it follows that the following inequalities hold if algorithm A achieves a cost ratio of α on all request sequences:

$$\begin{cases} \mathbf{E}(C_A(\sigma_i)) \leq \alpha \cdot i & i < 2d \\ \mathbf{E}(C_A(\sigma_i)) \leq \alpha \cdot 2d & i \geq 2d \end{cases}$$

To obtain the minimum possible α, we set the preceding inequalities to equalities and solve the resulting equations, yielding

$$p_i = (1-\alpha)\left(\frac{2d}{2d-1}\right)^i + \alpha$$

and

$$\alpha = \frac{e_{2d-1} + \frac{1}{4d}}{(e_{2d-1} - 1) + \frac{1}{2d}}.$$

This gives a lower bound on the competitive factor for the isosceles triangle. By using the probabilities obtained above, a randomized algorithm can achieve the same factor. □

Theorem 10 *Let B be any on-line probabilistic algorithm for the 2-server problem on a 3 vertex triangle with distances 3, 4, and 5. Then there exists an infinite sequence of requests σ for which*

$$\frac{\mathbf{E}(C_B(\sigma))}{C_{opt}(\sigma)} \geq \frac{1652}{1069} \approx 1.55.$$

Proof: Similar to previous theorem. □

References

[1] Berman, P., Karloff, H. J., and Tardos, G. A competitive three-server algorithm. *First Annual ACM-SIAM Symposium on Discrete Algorithms,* to appear.

[2] Borodin, A., Linial, N., and Saks, M. An optimal online algorithm for metrical task systems. In *19th Annual ACM Symposium on Theory of Computing*, pages 373–382, New York City, NY, May 1987.

[3] Borodin, A., Linial, N., and Saks, M. An optimal online algorithm for metrical task systems. Submitted for publication.

[4] Chrobak, M., Karloff, H., Payne, T., Vishwanathan, S. New results on server problems. *First Annual ACM-SIAM Symposium on Discrete Algorithms,* to appear.

[5] Chrobak, M. and Larmore, L. L. A new approach to the server problem. Manuscript.

[6] Eggers, S. J. and Katz, R. H. Evaluating the performance of four snooping cache coherency protocols. In *Proceedings of 16th Annual International Symposium on Computer Architecture,* 1989.

[7] Fiat, A., Karp, R. M., Luby, M., McGeoch, L. A., Sleator, D. D., and Young, N. E. *Competitive paging algorithms.* Technical Report CMU-CS-88-196, School of Computer Science, Carnegie Mellon University, 1988.

[8] Irani, S. and Rubinfeld, R. A competitive 2-server algorithm. Manuscript.

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

[10] Manasse, M. S., McGeoch, L. A., and Sleator, D. D. Competitive algortihms for on-line problems. In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing,* pages 322–333, Chicago, Illinois, May 1988.

[11] McGeoch, L. A. *Algorithms for Two Graph Problems.* PhD thesis, Carnegie Mellon University, 1987.

[12] McGeoch, L. A. and Sleator, D. D. *A Strongly Competitive Randomized Paging Algorithm.* Technical Report CMU-CS-89-122, School of Computer Science, Carnegie Mellon University, 1989.

[13] Raghavan, P. and Snir, M. Memory vs. randomization in on-line algorithms. In *ICALP,* Italy, July 1989.

[14] Sleator, D. D. and Tarjan, R. E. Amortized efficiency of list update and paging rules. *CACM,* 28(2):202–208, 1985.