Memory versus randomization in on-line algorithms

by P. Raghavan
M. Snir

On-line algorithms service sequences of requests, one at a time, without knowing future requests. We compare their performance with the performance of algorithms that generate the sequences and service them as well. In many settings, on-line algorithms perform almost as well as optimal off-line algorithms, by using statistics about previous requests in the sequences. Since remembering such information may be expensive, we consider the use of randomization to eliminate memory. In the process, we devise and study performance measures for randomized on-line algorithms. We develop and analyze memoryless randomized on-line algorithms for the cacheing problem and its generalizations.

Those who do not remember the past are condemned to relive it . . .

Santayana

. . . unless they act randomly.

1. Introduction

An algorithm is said to be *on-line* if it decides how to satisfy each request of a sequence of requests on the basis of knowledge of the past requests in the sequence but with no knowledge of future requests. *On-line problems*, i.e., those using on-line algorithms for their solution, arise frequently in operations research and computer science [1–3]. On-line algorithms have been analyzed with probabilistic models for the distribution of requests. More recently, *competitive analysis* has been used to dispense with such probabilistic assumptions [3]. In it, one compares, for each sequence of requests, the performance of the on-line algorithm to the performance of an optimal off-line algorithm (one with full knowledge of future requests) on that same sequence. Bounds derived in this manner hold for any probability distribution on the inputs (and even hold when past requests do not predict future requests in any way). Sleator and Tarjan [3] apply this approach to the analysis of the move-to-front (MTF) heuristic for maintaining a linear search list and to the analysis of the least-recently-used (LRU) policy for cache management. Karlin et al. [4] adopt this approach to the analysis of snoopy cacheing protocols.

To gain some insight into general principles for designing on-line algorithms that perform well in the competitive sense, several authors have suggested extending these special cases (MTF for lists and LRU for paging) to obtain general frameworks for the study of on-line algorithms. Manasse et al. [5] introduced the *k-server problem*, and Borodin et al. [6] introduced the more general *metrical task systems*. An intriguing conjecture of Manasse et al. [5] has recently generated much work on particular cases of the server problem [7–9].

Competitive analysis ignores the *computational resources* of the on-line algorithm. Indeed, the lower bounds that are typically proved for the performance of on-line algorithms hold for algorithms that are limited only

by their lack of knowledge of the future—in fact, the bounds even apply to algorithms that consume arbitrary amounts of time or space to service each request.

From a practical standpoint, an important requirement of an on-line algorithm is that it maintain very little state information (memory) from the past and that its response to each request be easy to compute. For example, the LRU cacheing algorithm, defined below, must maintain the order of the last access to each page in memory. Such state memory is expensive and slow to update in hardware, as pointed out by So and Rechtschaffen [10]. For LRU *paging*, one would have to maintain the order of the last access to each main-memory frame (since main memory is fully associative), which is not feasible. Previous theoretical studies have not touched on the issue of the memory resources required by an on-line cacheing algorithm. Randomization is an alternative to using large state memory. We refer here to algorithms that make probabilistic choices during execution, and we study their performance under worst-case inputs.

In competitive analysis, a deterministic on-line cacheing algorithm is analyzed by comparing its performance with the performance of an off-line algorithm on a "worst-case" sequence of requests that can be assumed to be generated by an adversary. While there is such a single, natural definition of performance for deterministic on-line algorithms, a number of natural definitions arise for the performance of randomized on-line cacheing algorithms, depending on the power the adversary is assumed to have. These are examined in Section 2. We explore the relations between the various definitions, expanding on results first derived by Ben-David et al. [11]. (These definitions and results are easily extended to the more general *server problem* or to *metrical task systems*; these general problems are defined below.) Some of our results are derived with the use of theorems on infinite games and are presented in Appendix A.

In Section 3, we analyze a very simple randomized cacheing algorithm, namely random replacement, and show that by one measure, it is as good as LRU. This algorithm, using no information from the past, is memoryless; it uses up to $\log m$ random bits, where m is the cache size, on each request. We show that there is a direct trade-off between the number of memory bits and the number of random bits used by optimal on-line cache-replacement algorithms, and that no memoryless algorithm performs better than random replacement.

In Section 4, we extend the random replacement algorithm to give a solution to the *weighted-cache problem*, a problem of practical interest, for which no provably good algorithm was known before. A deterministic on-line algorithm has subsequently been found by Chrobak et al. [8] for one case of this problem.

On-line algorithms typically use memory to maintain statistics on past events. This is replaced in randomized, memoryless algorithms by probabilistic processes whose probability distributions implicitly reflect these statistics. In Section 5, we present two instances of this technique: Deterministic graph-traversal algorithms are replaced by probabilistic, memoryless random walks, and counters are replaced by "probabilistic counters." This approach is used to derive simple memoryless algorithms for two types of on-line problems. We derive a simple algorithm, which is memoryless and randomized, for the k-server problem in a metric space with n points. We give a bound on the performance of this algorithm for the cases $k = 2$ and $k = n - 1$ and present a tantalizing conjecture that, if true, would yield a bound on our algorithm's performance for arbitrary k. We also derive memoryless algorithms for *metrical task systems*.

2. Cacheing algorithms

Definitions

We study cacheing algorithms by using a simple two-level store, consisting of a *main memory* and a *cache*, as the model. Our model is essentially that of Sleator and Tarjan, with added provisions for studying randomized algorithms and the amount of state information required by on-line algorithms. The *main memory* consists of a (potentially infinite) number of locations, each of which always contains one copy of a distinct *item*. The *cache* consists of m locations, each capable of storing one such item. The cacheing algorithm is given a sequence $v_1, v_2, \cdots, v_n$ of *references* to items. The cache is initially "empty," i.e., containing none of the items. A *hit* is said to occur on the ith reference if v_i is one of the items in the cache after reference $i - 1$; otherwise, a *miss* is said to occur. When a miss occurs on the reference to v_i, the cacheing algorithm selects a cache location, specified by an integer in $[1, \cdots, m]$. The item at that location is evicted, and item v_i is loaded in its place. The algorithm is *on-line* if the selection of an item for eviction at step i depends only on $v_1, \cdots, v_i$; otherwise it is *off-line*.

Let X denote the space of all the items that can be requested in a reference. The set of items residing in the cache is thus a point in X^m. (Assume that the "null item" is in X, to allow for empty cache locations.) We now define a cache-management algorithm that consists of an automaton with a finite set S of states. The response of this automaton to a reference is specified by a function F that depends on the current state of the automaton, the m items in the cache, and the item newly requested from X; it specifies, in general, a new state for the automaton, together with the new set of items in the cache:

$$F: S \times X^m \times X \rightarrow S \times X^m. \quad (1)$$

We impose the following condition on F: The set of items in the cache after the request is serviced must include the item just referenced:

$$F(s, x_1, \cdots, x_m, y) = (s', y_1, \cdots, y_m) \Rightarrow y \in \{y_1, \cdots, y_m\},$$

where $s, s' \in S$; $x_j, y_j, y \in X$ for $j = 1, \cdots, m$.

The definition above permits very general cacheing algorithms. For example, it allows for more than one item to be evicted on a miss, or for the cache contents to be changed on a hit. However, we may assume without loss of generality that $\{y_1, \cdots, y_m\} \subseteq \{x_1, \cdots, x_m, y\}$; that is, no load occurs on a hit, and a unique load (of the missing item) occurs on a miss. The reason we may assume this is as follows [4]. For any algorithm that does not satisfy this rule, we can derive a new algorithm that satisfies this rule and that makes a number of misses on any sequence which is no larger than the number of misses of the original algorithm. As a result, only one copy of any item ever resides in the cache.

Since the cost of evicting/missing is the same for all items, we may assume that F does not depend on the identities of the items; that is, for any permutation σ on X,

$$F(s, x_1, \cdots, x_m, y) = F[s, \sigma(x_1), \cdots, \sigma(x_m), \sigma(y)].$$

Given the above restrictions, we may represent on-line cacheing algorithms in the following form. The algorithm is defined by the functions *Hit* and *Miss*:

$$Hit: [1, \cdots, m] \times S \rightarrow S, \tag{2}$$

$$Miss: S \rightarrow [1, \cdots, m] \times S. \tag{3}$$

If a hit occurs at location j (i.e., $y = x_j$) when the automaton is in state s, the state is updated to $Hit(j, s)$. If a miss occurs in state s, the item at location j is evicted (and the missing item is loaded instead); also, the state is updated to s', where $Miss(s) = (j, s')$. (Note that j and s' are independent of y because the items are assumed to be indistinguishable.) We adopt a natural measure of the state information maintained by the algorithm: its *memory*, defined to be $\log_2 |S|$. An algorithm whose memory is 0 is called *memoryless*.

The restriction of cacheing algorithms from the general form given by (1) to the special formulation given by (2) and (3) does not affect performance (i.e., the number of misses); however, it may affect the size of the state memory (this was pointed out to us by Marek Chrobak). Since the latter form is closer to the way cacheing algorithms are actually implemented (decisions do not depend on the actual addresses in the sequence of requests), we use it for the analysis of cacheing.

In a randomized algorithm, the state transitions may be probabilistic. Thus, $Hit(j, s)$ may be a probability distribution on S, and $Miss(s)$ a probability distribution on $[1, \cdots, m] \times S$. We can also describe such a randomized cacheing algorithm by means of two functions:

$$Hit: [1, \cdots, m] \times S \times \Omega \rightarrow S \tag{4}$$

and

$$Miss: S \times \Omega \rightarrow S \times [1, \cdots, m], \tag{5}$$

where Ω is a probability space. At each step i, the algorithm makes a random choice of a point $\omega_i \in \Omega$, and the corresponding deterministic transition is executed. All choices are independent. We measure the *randomness* of the algorithm by the *entropy* of the probability space Ω. If Ω is discrete and ω_i occurs with probability p_i, the randomness equals

$$-\sum_{i=1}^{m} p_i \log_2 p_i .$$

For a cacheing algorithm A, we denote by $C_m^A(v_1, \cdots, v_n)$ the number of misses on the sequence of accesses $v_1, \cdots, v_n$ when algorithm A is used on a cache of size m. If the algorithm is randomized, this number is a random variable.

- *Example algorithms*

LRU (least recently used) Whenever a miss occurs, the least recently referenced item in the cache is evicted. The state encodes the order of the most recent reference to the items in the cache, and is updated appropriately at each reference. The algorithm is deterministic and uses $|S| = m!$ states [thus, $\Theta(m \log m)$ memory][1] for a cache with m locations.

Random Whenever a miss occurs, a cache location is chosen at random and the item in it is evicted. The algorithm is memoryless but uses $\log m$ bits of randomness per miss.

FIFO (first-in, first-out) Whenever a miss occurs, the item that has been in the cache for the longest period is evicted. The scheme can be implemented using a mod m counter to point to the item to be evicted at the next miss; the counter is incremented following the eviction. This is a deterministic algorithm that uses m states, i.e., $\log m$ bits of memory.

FWF (flush when full [4]) Initially, all cache locations are marked as "available." Whenever a miss occurs, an arbitrary available item is evicted, and the newly loaded item is marked as "unavailable"; if there are no available entries, all entries are marked available before the eviction

[1] Henceforth, "$\log x$" is used to denote $\log_2 x$.

occurs. Note that FIFO is a particular case of FWF, in which one always evicts the first available entry. The FWF algorithm uses m bits of memory and has no randomization.

RFWF (random flush when full [12]) This is like FWF, with the following two modifications: First, a random available item is selected for eviction, and second, an item is marked unavailable whenever it is accessed on a hit. The algorithm uses m memory bits and has up to $\log m$ random bits of randomness per miss.

- *Performance measures*

We compare the performance of an on-line cacheing algorithm with the performance of a cache managed by an *adversary* that also generates the sequence of references. Following [3], we find it instructive to compare an on-line algorithm having a cache containing M locations with an adversary having a cache containing m locations, $m \leq M$. The measure of performance we use is known as *competitiveness*.

Several definitions of competitiveness are possible, according to the assumptions made about the information available to the adversary when it generates the sequence of references and when it manages its cache. An *oblivious* adversary fixes the entire sequence of references $v_1, v_2, \cdots$ in advance; an *adaptive* adversary sees the state of the on-line algorithm after i references, and chooses v_{i+1} accordingly. An oblivious adversary corresponds to the situation in which the sequence of references cannot be affected by the decisions made by the on-line cache-management algorithm. An adaptive adversary corresponds to the situation in which such an effect is possible, e.g., in a reactive, real-time system, in which the cache behavior may affect the computation performed. Another example stems from operating systems, in which page tables and associated information are some of the items being referenced.

An *on-line* adversary manages its own cache on line; if a miss occurs in the adversary cache at step i, the choice of an item for eviction depends only on the first i references. An *off-line* adversary has no such restriction on its cache-management algorithm. We can assume that an off-line adversary always uses an optimal cacheing algorithm. One such algorithm, known as Min, whenever a miss occurs, evicts the item in the cache whose next reference is furthest into the future. The Min algorithm produces the smallest number of misses on every sequence of references [13]. An off-line adversary provides a suitable yardstick when a specific, finite computation task is to be performed and it is feasible to program the cache for that specific task. In general, an on-line adversary provides a more suitable yardstick for a system that handles a potentially infinite sequence of references. We thus have four types of adversaries: oblivious on-line, oblivious off-line, adaptive on-line, and adaptive off-line. There is no difference, however, between oblivious on-line and off-line adversaries: Whatever cacheing algorithm is used by the adversary can be executed equally well by an on-line adversary as by an off-line adversary. Thus, in reality, we have three distinct types of adversaries, listed here in order of increasing power:[2]

- *Oblivious (o)* The adversary generates a fixed sequence of references $v_1, v_2, \cdots$; for the first n references, it incurs a cost equal to the smallest possible number of misses on such a sequence of references (for a cache of size m).
- *Adaptive, on-line (an)* The adversary generates reference v_i and updates its cache in response to v_i as a function of the responses of the on-line algorithm to the first $i - 1$ requests.
- *Adaptive, off-line (af)* The adversary generates reference v_i as a function of the responses of the on-line algorithm to the first $i - 1$ requests. For the first n references, it incurs a cost equal to the smallest possible number of misses on such a sequence of references (for a cache of size m).

A cacheing algorithm A is said to be *c-competitive* against an adversary B if there exists a constant C such that

$$\limsup_{n\to\infty}[C_M^A(v_1, \cdots, v_n) - c \cdot C_m^B(v_1, \cdots, v_n)] < \infty, \text{ a.s.} \quad (6)$$

(a.s. = almost surely—i.e., with probability one), where $v_1, v_2, \cdots$ is the sequence of references generated by the adversary, $C_M^A(v_1, \cdots, v_n)$ is the number of misses incurred by A for this sequence (on a cache of size M), and $C_m^B(v_1, \cdots, v_n)$ is the number of misses incurred by the adversary for this sequence (on a cache of size m). Specifically, we have the following definition.

Definition 1

A cacheing algorithm A is said to be $c(M, m)$-competitive *against an adversary B if*

$$\limsup_{n\to\infty}[C_M^A(v_1, \cdots, v_n) - c(M, m) \cdot C_m^B(v_1, \cdots, v_n)] < \infty, \text{ a.s.} \quad (7)$$

This definition allows us to compare the on-line and off-line algorithms in a general setting in which the on-line algorithm potentially has a larger cache than the off-line algorithm. Similarly, we say that an algorithm A is *$c(M, m)$-competitive against adaptive on-line (or adaptive off-line, or oblivious) adversaries* if it is $c(M, m)$-

[2] A preliminary version of this paper, appearing in the *Proceedings of ICALP 89* [14], distinguishes only two situations. As a consequence, several of the theorems reported there are either ambiguous or incorrect as stated.

competitive against any adversary of the corresponding type.

We define $\mathcal{C}^A_{af}(M, m)$, the *adaptive off-line competitiveness coefficient* of algorithm A, to be the least upper bound on $c(M, m)$ such that A is $c(M, m)$-competitive against adaptive off-line adversaries. The *adaptive on-line competitiveness coefficient*, $\mathcal{C}^A_{an}(M, m)$, and the *oblivious competitiveness coefficient*, $\mathcal{C}^A_{o}(M, m)$, are similarly defined. We have

$$\mathcal{C}^A_{o}(M, m) \leq \mathcal{C}^A_{an}(M, m) \leq \mathcal{C}^A_{af}(M, m).$$

We show later that each of these inequalities can be strict.

Given a sequence of requests $v_1, \cdots, v_n$, let $C^{Opt}_m(v_1, \cdots, v_n)$ denote the optimal cost of servicing $(v_1, \cdots, v_n)$. When algorithm A is deterministic, the adversary can predict its moves, and there is no difference between oblivious and adaptive adversaries. Thus, for deterministic algorithms, all three definitions of competitiveness coalesce and are equivalent to the following definition: If algorithm A is deterministic, it is $c(M, m)$-competitive if and only if

$$\limsup_{n\to\infty}[C^A_M(v_1, \cdots, v_n) - c(M, m) \cdot C^{Opt}_m(v_1, \cdots, v_n)] < \infty \tag{8}$$

for any sequence $v_1, v_2, \cdots$ of references. We denote by $\mathcal{C}^A_{d}(M, m)$ the least upper bound on $c(M, m)$ such that a deterministic algorithm A is $c(M, m)$-competitive.

We define $\mathcal{C}_{af}(M, m)$, the *competitiveness coefficient of cacheing for adaptive off-line adversaries*, to be the least upper bound on $c(M, m)$ such that there is a cacheing algorithm A that is $c(M, m)$-competitive against adaptive off-line adversaries. Thus,

$$\mathcal{C}_{af}(M, m) = \inf \mathcal{C}^A_{af}(M, m),$$

where the infimum is taken over all on-line cacheing algorithms. The *competitiveness coefficient of cacheing for adaptive on-line adversaries*, $\mathcal{C}_{an}(M, m)$, the *competitiveness coefficient of cacheing for oblivious adversaries*, $\mathcal{C}_{o}(M, m)$, and the *competitiveness coefficient of cacheing for deterministic algorithms*, $\mathcal{C}_{d}(M, m)$, are similarly defined. A cacheing algorithm A is *optimal against adaptive off-line adversaries* if $\mathcal{C}^A_{af}(M, m) = \mathcal{C}_{af}(M, m)$. A similar definition is used for adaptive on-line and oblivious adversaries, and for deterministic algorithms.

We have

$$\mathcal{C}_{o}(M, m) \leq \mathcal{C}_{an}(M, m) \leq \mathcal{C}_{af}(M, m) \leq \mathcal{C}_{d}(M, m).$$

The following two theorems give more information on the relationships among these coefficients.

Theorem 2.1
Let A be a randomized cacheing algorithm that is $c(M, m)$-competitive against adaptive off-line adversaries. Then there is a deterministic algorithm $\hat{A}$ that is $c(M, m)$-competitive.

The theorem is proved in Appendix A. The proof uses a formulation of the cacheing problem as an infinite game, and standard results on infinite games. The theorem was first proved by Ben-David et al. [11] using the alternative definition of competitiveness presented in the section of Appendix A on alternative definitions (Definition 3), which permits a simpler proof.

The theorem implies that randomization does not yield more competitive algorithms against adaptive off-line adversaries. The proof is not constructive. Moreover, the resulting deterministic algorithm uses an infinite control (has an infinite number of states), even if the original randomized strategy has a finite control. Thus, randomization may still result in more practical algorithms. This last result does not extend to oblivious or adaptive on-line adversaries.

The following theorem is a generalization of one due to Ben-David et al. [11] and, in fact, uses the more stringent definition of competitiveness given in Appendix A (Definition 3).

Theorem 2.2
Let A be a cacheing algorithm with $\mathcal{C}^A_{an}(M, n) = a$, and let Q be a cacheing algorithm with $\mathcal{C}^Q_{o}(n, m) = b$, where $m \leq n \leq M$. Then $\mathcal{C}^A_{af}(M, m) \leq ab$.

Proof Let D be an adaptive off-line adversary, with a cache of size m. Let D' be the adaptive on-line adversary defined as follows: D' generates references as D does; D' manages its own cache on-line, using algorithm Q. Thus, the cache management of adversary D' at step i depends on the sequence of references up to step i but does not depend on the previous actions of A. Let α be the sequence of random choices of algorithm A, and let β be the sequence of random choices of algorithm Q. These two random sequences are independent. We denote by $\Pr_{\alpha,\beta}$ the joint distribution induced by these sequences and denote the marginal distributions by $\Pr_\alpha$ and $\Pr_\beta$. Since A is a-competitive against adaptive on-line adversaries, we have

$$\Pr_{\alpha,\beta}\{\limsup_{k\to\infty}[C^A_M(v_1, \cdots, v_k) - a \cdot C^{D'}_n(v_1, \cdots, v_k)] < \infty\} = 1.$$

Since Q is b-competitive against oblivious adversaries, we have

$$\Pr_{\beta}\{\limsup_{k\to\infty}[C^Q_n(v_1, \cdots, v_k) - b \cdot C^{Opt}_m(v_1, \cdots, v_n)] < \infty\} = 1,$$

for any fixed sequence of references $v_1, v_2, \cdots$. Thus,

$$\Pr_{\beta}\{\limsup_{k\to\infty}[C_n^{D'}(v_1,\cdots,v_k) - b\cdot C_m^{Opt}(v_1,\cdots,v_k)] < \infty|\alpha\} = 1,$$

and

$$\Pr_{\alpha,\beta}\{\limsup_{k\to\infty}[C_n^{D'}(v_1,\cdots,v_k) - b\cdot C_m^{Opt}(v_1,\cdots,v_k)] < \infty\} = 1.$$

It follows that

$$Pr_{\alpha,\beta}\{\limsup_{k\to\infty}[C_M^{A}(v_1,\cdots,v_k) - ab\cdot C_m^{Opt}(v_1,\cdots,v_k)] < \infty\} = 1. \quad \square$$

Theorems 2.1 and 2.2 imply the following:

$$\mathcal{C}_o(M,m) \le \mathcal{C}_{an}(M,m) \le \mathcal{C}_{af}(M,m) = \mathcal{C}_d(M,m) \le \mathcal{C}_o(m,m)\cdot\mathcal{C}_{an}(M,m).$$

Fiat et al. [12] have shown that the RFWF algorithm has an oblivious competitiveness coefficient of $2H_m$ when $m = M$. (H_m is the mth harmonic number: $H_m \equiv 1 + 1/1 + 1/2 + \cdots + 1/m$; $\ln m \le H_m \le \ln m + 1$.) They also showed that no cacheing algorithm has an oblivious competitiveness coefficient smaller than H_m. More recently, McGeoch and Sleator [15] have shown that $\mathcal{C}_o(m,m) = H_m$. On the other hand, Sleator and Tarjan [3] have shown that the LRU algorithm is $M/(M-m+1)$-competitive and that no deterministic algorithm has a lower competitiveness coefficient. Manasse et al. [5] have extended the lower-bound proof to the more general server problems considered in Section 5. Their argument actually holds for randomized algorithms and adaptive on-line adversaries (see Theorem 5.6). We thus have

$$\mathcal{C}_{an}(M,m) = \mathcal{C}_{af}(M,m) = \mathcal{C}_d(M,m) = M/(M-m+1)$$

and

$$\mathcal{C}_o(m,m) = H_m .$$

The value of $\mathcal{C}_o(M,m)$, for $m \neq M$, is not known. Also, for any randomized cacheing algorithm A, we have

$$\frac{\mathcal{C}_{af}^{A}(M,m)}{H_m} \le \mathcal{C}_{an}^{A}(M,m) \le \mathcal{C}_{af}^{A}(M,m).$$

We show in Theorems 3.2, 3.3, and 3.4 that both inequalities can be tight for particular algorithms.

The following lemma proves to be of use in our analyses of randomized cacheing algorithms in subsequent sections.

Lemma 2.3
Let $X_1, X_2, \cdots$ be a sequence of random variables such that $\mathrm{E}[X_1] \le \beta < 0$, *a.s., and*

$$\mathrm{E}[X_i|X_1,\cdots,X_{i-1}] \le \beta < 0, \text{ a.s.} \qquad i > 1$$

and

$$var(X_i) \le \gamma < \infty, \ \forall\, i,$$

where $var(Y)$ denotes the variance of random variable Y. Then

$$\sum_{i=1}^{\infty} X_i = -\infty, \text{ a.s.}$$

Proof We have

$$\sum_{i=1}^{\infty} \frac{var(X_i)}{i^2} < \infty.$$

This implies that

$$\lim_{n\to\infty} \frac{1}{n}\cdot\left\{(X_1 - \mathrm{E}[X_1]) + \sum_{i=2}^{n}(X_i - \mathrm{E}[X_i|X_1,\cdots,X_{i-1}])\right\} = 0, \text{ a.s.}$$

(see [16], 32.1.E). Thus,

$$\limsup_{n\to\infty} \frac{1}{n}\cdot\sum_{i=1}^{n} X_i \le \beta, \text{ a.s.}$$

This implies that

$$\sum_{i=1}^{\infty} X_i = -\infty, \text{ a.s.} \quad \square$$

3. Performance of the Random algorithm

In this section, we study the competitiveness of the Random algorithm for cacheing, against each of the adversaries we have defined.

It is known [3, 4] that the LRU, FIFO, and FWF algorithms are $M/(M-m+1)$-competitive, and that no deterministic algorithm has a lower competitiveness coefficient. $M/(M-m+1)$ is also the best competitiveness coefficient for randomized algorithms against adaptive on-line adversaries; this follows from the lower bound of Theorem 5.6, for the more general server problem, to be defined in Section 5. RFWF has the same performance as FWF against adaptive on-line or off-line adversaries. The RFWF algorithm has a competitiveness coefficient $\mathcal{C}_o^{A}(m,m) = 2H_m$ against oblivious adversaries, and no algorithm has a competitiveness coefficient that is smaller than H_m [12, 15]. Thus, RFWF is optimal against any type of adversary.

We now analyze the performance of the simple Random algorithm, which is memoryless. We show that Random

has a competitiveness coefficient of $M/(M - m + 1)$ and, consequently, optimal performance against adaptive on-line adversaries. Random is not optimal, however, against oblivious adversaries or against adaptive off-line adversaries: We show that Random has a competitiveness coefficient $M/(M - m + 1)$ against oblivious adversaries, whereas the RFWF algorithm achieves $O(\log m)$ when $m = M$. We also show that Random has a competitiveness coefficient that is greater than or equal to $m \ln m$ against adaptive off-line adversaries when $m = M$, while LRU, FIFO, and FWF achieve m in this case.

Theorem 3.1
Random is $c(M, m)$-competitive against adaptive on-line adversaries, where $c(M, m) \leq M/(M - m + 1)$.

Proof We use a potential function to analyze the performance of Random amortized over a long sequence of references. This is seen to correspond to a random walk with negative drift on a line. (Random is a "lazy algorithm," which does not change state on hits. Thus, by Lemma A.1 (see Appendix A), we can restrict our attention to "cruel" adversaries, which cause Random to miss at each reference.) Let ω_i be the random choice of item to evict made by Random at step i, should a miss occur. Let S_i^R be the set of items that Random has in the cache (of size M) after the ith reference; let S_i^B be the set of items kept in the cache (of size m) by the adversary B after the ith reference. Let t_i^R be an indicator variable that is 1 if Random misses at reference i and 0 otherwise; let t_i^B be similarly defined for the adversary. Let

$$\Phi_i \equiv |S_i^R \cap S_i^B| ,$$

and let $\Delta\Phi_i = \Phi_i - \Phi_{i-1}$. For any $c > M/(M - m + 1)$, consider the sequence of random variables

$$X_i \equiv t_i^R - c \cdot t_i^B - c \cdot \Delta\Phi_i .$$

Then

$$\sum_{i=1}^{n} X_i$$

$$= C_M^R(v_1, \cdots, v_n) - c \cdot C_m^B(v_1, \cdots, v_n) - c \cdot \Phi_n + c \cdot \Phi_0 .$$

But $\Phi_0 \leq m$ and $\Phi_n \leq m$. Thus, Random is c-competitive if $\limsup_{n\to\infty} \sum_{i=1}^{n} X_i < \infty$, a.s. We complete the proof by showing that this is indeed so. We consider two mutually exclusive cases at step i:

- A_i: $v_i \notin S_{i-1}^R$, $v_i \in S_{i-1}^B$ (Random misses and adversary B hits).
  Then $t_i^R = 1$ and $t_i^B = 0$. With probability Φ_{i-1}/M, Random evicts an item in $S_{i-1}^R \cap S_{i-1}^B$, resulting in $\Delta\Phi_i = 1$. Thus,

$$E[X_i | A_i, \omega_1, \cdots, \omega_{i-1}] = 1 - c(M - \Phi_{i-1})/M.$$

  Since $S_{i-1}^B \not\subset S_{i-1}^R$, we have $\Phi_{i-1} = |S_{i-1}^B \cap S_{i-1}^R| \leq m - 1$, so

$$E[X_i | A_i, \omega_1, \cdots, \omega_{i-1}] \leq 1 - c(M - m + 1)/M.$$

- $\neg A_i$: $v_i \notin S_{i-1}^R$, $v_i \notin S_{i-1}^B$ (both Random and adversary B miss).
  Here $t_i^R = t_i^B = 1$. With probability $(\Phi_{i-1} - 1)/M$, Random evicts one of the $\Phi_{i-1} - 1$ items in $S_{i-1}^R \cap S_{i-1}^B$ that is not evicted by the adversary, in which case $\Delta\Phi_i = 1$; otherwise, $\Delta\Phi_i = 0$. It is easy to verify that

$$E[X_i | \neg A_i, \omega_1, \cdots, \omega_{i-1}] \leq 1 - c(M - m + 1)/M.$$

  It follows that

$$E[X_i | X_1, \cdots, X_{i-1}] \leq 1 - c(M - m + 1)/M.$$

  It is also easy to verify that

$$|X_i| \leq 2c,$$

  so that the variance of the random variable X_i is uniformly bounded.

Let $\beta = \min[-c, 1 - c(M - m + 1)/M]$. By our choice of c, we have $\beta < 0$. The sequence of random variables $X_1, X_2, \cdots$ fulfills the conditions of Lemma 2.3, so

$$\sum_{i=1}^{\infty} X_i = -\infty, \text{ a.s.} \qquad \square$$

The proof of the previous theorem also implies that

$$\lim_{n\to\infty} E[C_M^R(v_1, \cdots, v_n) - c \cdot C_m^B(v_1, \cdots, v_n)] = -\infty,$$

for any adaptive on-line adversary B and any $c > M/(M - m + 1)$. Thus, Random also satisfies the criteria for competitiveness of Definition 3 of Appendix A, with competitiveness coefficient $M/(M - m + 1)$.

A crucial observation here is that the kind of one-step martingale analysis used in the above proof *does not* apply to adaptive off-line adversaries (this was brought to our attention by Anna Karlin). Here, the quantity X_i at the ith reference v_i depends on the adversary's response to v_i, which is a function of future references (and thus the algorithm's future random choices). Thus, the behavior of X_i is not determined by Φ_i alone. Indeed, the following theorem, proved by A. Karlin, shows that a statement such as that of Theorem 3.1 does not hold for adaptive off-line adversaries.

Theorem 3.2
The competitiveness coefficient of Random against adaptive off-line adversaries for $M = m$ is $c(m, m) \geq m \ln m$.

Proof Consider a sequence of references generated from a set of $m + 1$ distinct items $a_0, \cdots, a_m$; repeatedly, a reference is generated to the unique item not in the cache (as would be done by a cruel adversary). Divide the sequence of references into segments $S_1, S_2, \cdots$ according to this rule: S_i is the shortest sequence of references following S_{i-1} in which each of the items $a_0, \cdots, a_m$ occurs at least once. Let b_i be the last item referenced in segment S_i. Then b_i is referenced exactly once in segment S_i. An off-line cacheing algorithm (using a cache of size m) can satisfy the sequence of references with exactly one miss in each segment: When the reference to b_{i-1} occurs at the end of S_{i-1}, the algorithm evicts b_i and loads b_{i-1}. Thus, when segment S_i starts, the cache contains all items referenced in this segment, with the exception of the last reference to b_i. The Random algorithm, on the other hand, has one miss at each step. Let L be the expected length of a segment S_i. Then L is the expected number of steps before Random evicts each of the $m + 1$ items at least once. This can be formulated as a slight variation of the *coupon collector problem*: There are $m + 1$ coupons to collect, and at each step one is equally likely to collect one of the m coupons that were not collected at the previous step. The expected waiting time until all coupons have been collected is $mH_m + 1$ [16]. □

Lemma 3.3
Let W be the waiting time for success in a sequence of Bernoulli trials, with probability of success p, and let W_k be the truncated variable defined by

$$W_k = \begin{cases} W & \text{if } W \leq k, \\ k & \text{if } W > k, \end{cases}$$

where k is an integer and W a positive real number. Then

$$\mathrm{E}[W_k] = \frac{1}{p}[1 - (1 - p)^k].$$

Proof We have

$$\begin{aligned} \mathrm{E}[W_k] &= \mathrm{E}[W] - \mathrm{E}[W - k | W > k] \cdot \Pr[W > k] \\ &= \mathrm{E}[W] \cdot (1 - \Pr[W > k]) \\ &= \frac{1}{p}[1 - (1 - p)^k]. \end{aligned}$$

□

Theorem 3.4
The oblivious competitiveness coefficient of Random is $M/(M - m + 1)$.

Proof Consider the following sequence of references that suffice to prove the lower bound:

$$a_1 a_2, \cdots, a_m (b_1 a_2, \cdots, a_m)^2 (b_2 a_2, \cdots, a_m)^3, \cdots, (b_j a_2, \cdots, a_m)^j, \cdots,$$

where the a_j and the b_j are all distinct items. Here, $(s)^k$ denotes k repetitions of the pattern s. The adversary (with a cache of size m) misses once on each segment $(b_i a_2, \cdots, a_m)^k$. At the beginning of any such segment, the cache (of size M) maintained by Random contains at most $m - 1$ of the items appearing in that segment. Let us define a *near miss* to be a miss that occurs when Random has exactly $m - 1$ of these items in its cache. Random *succeeds* on a near miss if it does not evict any of these $m - 1$ items. Random has at least one miss on each repetition of the pattern $b_j a_2, \cdots, a_m$, until it succeeds on a near miss. The probability of a success on a near miss is $(M - m + 1)/M$. Hence, by Lemma 3.3, the expected number of near misses is at least $[M/(M - m + 1)]\{1 - [(m - 1)/M]^k\}$. The claim follows. □

Theorem 3.4 can be strengthened to hold even if there are only $M + 1$ distinct items. Also, no memoryless algorithm achieves a better oblivious (or adaptive on-line) competitiveness coefficient. Intuition suggests that when there is no information on which to base the choice of the evicted item, random, equiprobable choice of an item to evict is at least as good as any other rule. We formally prove the claim below, for the case $M = m$. We defer the proof of the following theorem to Appendix A.

Theorem 3.5
Any memoryless on-line cacheing algorithm has an oblivious competitiveness coefficient that is greater than or equal to m when $m = M$.

This theorem does not hold true for algorithms with memory: Fiat et al. [12] have shown that RFWF has an oblivious competitiveness coefficient of $O(\log m)$ when $m = M$.

4. The weighted cache problem

We now consider a generalization of the problem studied in the previous section. As before, we consider a two-level store with a cache capable of holding m items at a time. In the *weighted cache problem*, an item x has a positive real *weight* $w(x)$, representing the cost of loading the item into the cache. In measuring the competitiveness of an algorithm, we compare the cost it incurs over a sequence of references (rather than the number of misses) with the

cost incurred by the optimal off-line algorithm. We denote the cost of an algorithm A working with a cache containing M locations on the reference sequence $v_1, v_2, \cdots, v_n$ by $C_M^A(v_1, v_2, \cdots, v_n)$. The competitiveness coefficients of an algorithm A are defined accordingly. Thus, the previous section dealt with the special unit cost case, in which $w(x) = 1, \forall x$.

The weighted cache problem has applications to cacheing fonts in printers. The number of fonts that can be cached at a time in the printer is subject to a maximum, but fonts requiring larger files take longer to bring into the printer's memory.

There are two noteworthy aspects of the weighted cache problem that distinguish it from the simple cache problem considered in the previous section. First, finding the optimal off-line schedule is nontrivial—indeed, the only technique we know for this is the general reduction to the assignment problem, due to Chrobak et al. [8]. Second, there were no simple, good deterministic algorithms for this problem before this work. A deterministic algorithm for the weighted cache problem that is m-competitive for the special case $M = m$ has been obtained by Chrobak et al. [8].

We present in this section a simple generalization of the Random algorithm for this problem, which we call the Reciprocal algorithm. It is memoryless and has a competitiveness coefficient $\leq M/(M - m + 1)$ against adaptive on-line adversaries. The lower bound of Theorem 5.6 for the server problem (presented in Section 5) implies that no algorithm can do better than Reciprocal against adaptive on-line adversaries. No deterministic $M/(M - m + 1)$-competitive algorithm for the case $m \neq M$ is known. Also, no deterministic memoryless algorithm is competitive against adaptive on-line adversaries, even if the more lenient definition of memory deduced from the formulation implicit in (1) is used[3].

The behavior of the Reciprocal algorithm depends only on the weights of the items in the cache. Let $x_1, \cdots, x_m$ be the items in cache when a miss occurs. The Reciprocal algorithm uses the following simple probabilistic eviction rule: Evict x_j with probability p_j, where

$$p_j \equiv \frac{1/w(x_j)}{\sum_{k=1}^{m} 1/w(x_k)}.$$

Theorem 4.1
The adaptive on-line competitiveness coefficient of the Reciprocal algorithm is less than or equal to $M/(M - m + 1)$.

Proof As in the proof of Theorem 3.1, we use a potential function to create a random walk with a negative drift on the real line. Let S_i^H be the set of items kept in the cache by Reciprocal after the ith reference, and S_i^B be the set of items kept by the adversary. Let

$$\Phi_i \equiv \sum_{x \in \{S_i^H \cap S_i^B\}} w(x) - \frac{m-1}{M-m+1} \cdot \sum_{x \in \{S_i^H - S_i^B\}} w(x),$$

and $\Delta\Phi_i \equiv \Phi_i - \Phi_{i-1}$. Letting t_i^H denote the cost incurred by the Reciprocal algorithm in servicing the ith reference and t_i^B the corresponding cost of the adversary, we define

$$X_i \equiv t_i^H - \frac{\alpha M}{M-m+1} \cdot t_i^B - \beta \Delta\Phi_i,$$

where $\alpha > \beta > 1$. We now proceed along the lines of the proof of Theorem 3.1, breaking the analysis into two parts, or actions, Y and Z.

Y: The adversary evicts an item. We can assume that the adversary loads a new item only immediately before a reference to that item. Also, without affecting the analysis, we can assume that the adversary incurs the cost of the item it evicts rather than for the item it loads; thus, $t_i^B = w(x_i)$ if the adversary evicts x_i on reference i.

Z: The Reciprocal algorithm evicts an item on a miss and incurs a cost equal to the weight of the item it loads.

We examine the effect of the two kinds of action on the random walk $\sum_{j=1}^{i} X_j$. In particular, we examine the effect of either action on $\mathrm{E}[X_i | X_1, \cdots, X_{i-1}]$.

Y: The adversary evicts x' and loads x. Then $t_i^B = w(x')$, and $-\Delta\Phi_i \leq w(x')M/(M - m + 1)$. (The equality is realized when $x' \in \{S_{i-1}^H \cap S_{i-1}^B\}$ and $x \notin S_{i-1}^H$). Thus, the contribution of the adversary's action to $\mathrm{E}[X_i | X_1, \cdots, X_{i-1}]$ is <0.

Z: The Reciprocal algorithm misses on a reference to item x, so $t_i^H = w(x)$. Then $|S_{i-1}^H \cap S_{i-1}^B| \leq m - 1$, and $|S_{i-1}^H - S_{i-1}^B| \geq 1$. Thus,

$$\begin{aligned}\mathrm{E}[\Delta\Phi_i | Z, \omega_1, \cdots, \omega_{i-1}] &= w(x) - \frac{|S_{i-1}^H \cap S_{i-1}^B|}{\sum_{y \in S_{i-1}^H} 1/w(y)} + \frac{m-1}{M-m+1} \cdot \frac{|S_{i-1}^H - S_{i-1}^B|}{\sum_{y \in S_{i-1}^H} 1/w(y)} \\ &> w(x).\end{aligned}$$

(ω_i is the random choice made by Reciprocal at step i.) Thus, the contribution of the Reciprocal algorithm's action to $\mathrm{E}[X_i | X_1, \cdots, X_{i-1}]$ is also less than zero.

[3] Marek Chrobak (University of California, Riverside), personal communication, 1990.

Note that $|X_i - X_{i-1}|$ is bounded by a constant times the largest weight of any of the items. Applying Lemma 2.3 to the sequence of random variables X_i, we conclude that

$$\lim_{n\to\infty} \sum_{i=1}^{n} X_i = -\infty, \text{ a.s.},$$

so that

$$\limsup_{n\to\infty} \left[C_M^H(v_1, \cdots, v_n) - \frac{\alpha M}{M - m + 1} \cdot C_m^B(v_1, \cdots, v_n) \right] < \infty, \text{ a.s.}$$

This yields the result. □

We also have, in this theorem, $\lim_{n\to\infty} \mathrm{E}[\sum_{i=1}^{n} X_i] = -\infty$, so that the theorem also holds if Definition 3 of Appendix A is used. The theorem is valid even if there are infinitely many distinct weights; all that is required is that all weights be in a bounded range $0 < a < \textit{weight} < b < \infty$, where a and b are arbitrary constants.

Coppersmith et al. [17] have derived a general approach to the server problem described in Section 5 that yields the above algorithm and analysis as special cases.

Note that Random is exactly the Reciprocal algorithm restricted to the special case $w(x) = 1$, $\forall x$. Thus, by Theorem 3.4, the oblivious competitiveness coefficient of the Reciprocal algorithm is greater than or equal to $M/(M - m + 1)$. In fact, when $M = m$, this lower bound holds true for any set of at least $M + 1$ distinct items.

We require the following lemma from probability theory.

Lemma 4.2
Let $X_1, X_2, \cdots$ be a sequence of positive random variables with uniformly bounded expectations. Let τ be a stopping time for the sequence. Suppose further that $\mathrm{E}[X_1] = \Pr[\tau = 1]$ *and for* $i > 1$,

$$\mathrm{E}[X_i | X_1, \cdots, X_{i-1}] = \Pr[\tau = i | X_1, \cdots, X_{i-1}].$$

Then,

$$\mathrm{E}\left[\sum_{i=1}^{\tau} X_i\right] = 1.$$

Proof It is known that τ ranges over the natural numbers, the events $[\tau = i]$ depend only on $X_1, \cdots, X_i$, and τ is a.s. finite. We have $\mathrm{E}[X_i | X_1, \cdots, X_{i-1}] = \Pr[\tau = i | X_1, \cdots, X_{i-1}]$, so that

$$\mathrm{E}[X_1] + \sum_{i=2}^{\infty} \mathrm{E}[X_i | X_1, \cdots, X_{i-1}] = 1.$$

Let

$$Y_n = \begin{cases} 1 + \sum_{i=1}^{n} X_i & \text{if } \tau < n, \\ \sum_{i=1}^{\tau} X_i & \text{otherwise.} \end{cases}$$

Then the Y_n are nonnegative random variables, $Y_1 = 1$, and

$$\mathrm{E}[Y_n | X_1, \cdots, X_{n-1}] = \mathrm{E}[Y_{n-1} + X_n - I_{[\tau=n]} | X_1, \cdots, X_{n-1}] = Y_{n-1}.$$

Here $I_{[\tau=n]}$ is an indicator variable that is 1 when $\tau = n$, and 0 otherwise. Also,

$$\lim_{n\to\infty} Y_n = \sum_{i=1}^{\tau} X_i, \text{ a.s.}$$

The martingale convergence theorem [16] implies that

$$1 = \lim_{n\to\infty} \mathrm{E}[Y_n] = \mathrm{E}\left[\lim_{n\to\infty} Y_n\right] = \mathrm{E}\left[\sum_{i=1}^{\tau} X_i\right]. \quad \square$$

Theorem 4.3
Let $M = m$. The oblivious competitiveness coefficient of the Reciprocal algorithm $C(M, M)$ is greater than or equal to M, for any *set of $M + 1$ items (independent of the weights).*

Proof Let $a_0, \cdots, a_M$ be any $M + 1$ items. Let $w_i \equiv w(a_i)$. Consider a sequence that consists of successive *rounds* of references each of the form $a_0, \cdots, a_{M-1}$; assume that the Reciprocal algorithm starts with a_M in the cache and a_0 out of the cache. Let t be the number of misses on the sequence. Let $I(j)$ be the index of the jth item evicted, where $I(0) \equiv m$. Once a_m is evicted, no further misses occur; therefore, $I(t) = M$.

Without loss of generality, the Reciprocal algorithm incurs the costs of the items it evicts. The cost of Reciprocal on this sequence is

$$\sum_{j=1}^{t} w_{I(j)}.$$

When the jth eviction occurs, the cache contains all items except $a_{I(j-1)}$. Therefore, for $j \le t$,

$$\Pr[I(j) = r | I(j-1)] = \begin{cases} \dfrac{1/w_r}{\sum_{k \neq I(j-1)} 1/w_k} & \text{if } r \neq I(j-1), \\ 0 & \text{otherwise.} \end{cases}$$

Accordingly,

$$\mathrm{E}[w_{I(j)}|I(j-1)] = \frac{M}{\sum_{k \neq I(j-1)} 1/w_k}$$

and

$$\Pr[t = j|I(j-1)] = \frac{1/w_M}{\sum_{k \neq I(j-1)} 1/w_k}.$$

Thus,

$$\mathrm{E}[w_I(j)|I(j-1)] = M \cdot w_M \cdot \Pr[t = j|I(j-1)].$$

It follows, by the previous lemma, that

$$\mathrm{E}\left[\sum_{j=0}^{t} w_{I(j)}\right] = M \cdot w_M.$$

Consider now a sequence of references consisting of rounds $i = 0, 1, \cdots$. Round i is of the form $[S_{i \bmod (M+1)}]^{n_i}$, where $S_0 \equiv (a_1, \cdots, a_M)$, and $S_j \equiv (a_0, \cdots, a_{j-1}, a_{j+1}, \cdots, a_M)$ for $0 < j < M$, and $S_M \equiv (a_0, \cdots, a_{M-1})$. Let $c < 1$ be a positive constant. Let A_i be the event "$a_{i \bmod (M+1)}$ is in the cache at the end of the ith round." Let C_i^H be the cost of Reciprocal for the ith round. By taking the sequence $n_0, n_1, \cdots$ to grow sufficiently fast, we can establish that almost surely A_i occurs only a bounded number of times, and almost surely the cost of Reciprocal at the ith round is larger than $cMw_{i \bmod (M+1)}$, with a bounded number of exceptions. Thus,

$$\limsup_{n \to \infty} \left[\sum_{i=0}^{n} C_i^H - cM \sum_{i=0}^{n} w_{i \bmod (M+1)}\right] \geq 0, \text{ a.s.}$$

On the other hand, the cost of the adversary on the ith round is $w_{i \bmod (M+1)}$. □

5. Random walks and probabilistic counters

In this section, we study the interplay between random walks and the competitive analysis of on-line algorithms.

The idea underlying the randomized algorithms of the previous sections is that a deterministic process that explicitly remembers statistics from the past can be replaced by a probabilistic process whose distribution implicitly remembers such statistics. For example, FIFO ensures that once an item is brought into the cache, it is not evicted before m further misses occur. Random does the same in a probabilistic sense: An item once brought into the cache remains there for a number of misses whose expectation is m. The deterministic counter of FIFO is replaced by a "probabilistic counter" in Random. We provide a second example below in the setting of *on-line graph traversal*, an abstract problem defined in the following subsection. This abstraction proves useful when we subsequently analyze algorithms for *server systems* (an abstraction due to Manasse et al. [5], which includes cacheing as a special case), and for the *metrical task systems* of Borodin et al. [6].

• *Traversals and random walks*

Consider a complete graph G with n nodes $\{1, \cdots, n\}$. A finite *cost* or distance $d(i, j) > 0$ is associated with each edge (i, j). We assume that the distance matrix, **D**, is *metrical* (i.e., is symmetric and satisfies the triangle inequality). An instance of the *traversal problem* is defined by a specified sequence $i_1, i_2, \cdots, i_r$ of nodes in G. An algorithm A starts at some initial node i_0 and moves along the edges of the graph, not knowing the identity of the next node in the specified sequence, until it eventually reaches i_1; then it moves until it reaches i_2, and so on. The next move of the algorithm may depend on its current state and location, but not on the next node in the specified sequence.

We denote by $C^A(i_1, \cdots, i_r)$ the cost of the path traversed by algorithm A when visiting nodes $i_1, \cdots, i_r$. We compare this cost to the length $C(i_1, \cdots, i_r) = \sum_{s=1}^{r} d(i_{s-1}, i_s)$ of the optimal path $(i_0, i_1), (i_1, i_2), \cdots, (i_{r-1}, i_r)$. [Edge (i, j) is the optimal path between nodes i and j because of the triangle inequality.] A deterministic algorithm A is *c-competitive* on graph G if for any infinite sequence of nodes $i_1, i_2, \cdots$,

$$\limsup_{r \to \infty} [C^A(i_1, \cdots, i_r) - c \cdot C(i_1, \cdots, i_r)] < \infty.$$

The other definitions of Section 2 are extended in a similar manner: An oblivious adversary chooses the sequence $i_0, i_1, \cdots$ in advance; an adaptive on-line adversary chooses i_{k+1} when on-line algorithm A reaches i_k; and an adaptive off-line adversary chooses i_{k+1} only when on-line algorithm A reaches i_{k+1}.

Formally, a traversal algorithm with a set of states S is a function

$$F: S \times [1, \cdots, n] \to S \times [1, \cdots, n].$$

As before, we define an algorithm to be *memoryless* if $\log |S|$ is zero. The next move of a memoryless algorithm does not depend on the past in any way—it does not depend on previously visited nodes or on the number of times it has previously been at the current node.

We now define a deterministic traversal algorithm with memory called the *cyclic traversal algorithm*. Let $i_0, i_1, \cdots, i_{r-1}, i_0$ be a cycle in which each node of G occurs at least once (the cycle is not necessarily Hamiltonian). A cyclic traversal algorithm visits the nodes of G in the order defined by the cycle. The state s of the algorithm is an index that ranges from 0 to r, where $r \geq n$. If the

algorithm is in state s (and at node i_s), it next moves to state $(s + 1) \bmod r$ (and node $i_{(s+1) \bmod r}$). Cyclic traversal algorithms are not memoryless, because they remember the index s. The following result is proved in [6].

Theorem 5.1
For any graph on n nodes and any distance matrix, there is a deterministic cyclic traversal algorithm that is $4(n - 1)$*-competitive.* □

A *probabilistic traversal algorithm* is obtained by executing a *random walk* on the graph. We associate a *transition probability* $p(i, j)$ with each edge (i, j); $p(i, j)$ is the probability that the algorithm, when at node i, moves to node j. Thus, $\sum_j p(i, j) = 1$. The algorithm executes a random walk on the graph according to these transition probabilities. Notice that a probabilistic traversal algorithm is memoryless. Let $h(i, j)$ be the expected cost, or distance, of a random walk that starts at node i and ends when node j is first reached. Define the *edge expansion* of the random walk to be $\max_{i,j}[h(i, j)/d(i, j)]$.

Lemma 5.2 follows immediately from the definition of competitiveness for a traversal algorithm.

Lemma 5.2
A probabilistic traversal algorithm based on a random walk with edge expansion c is c-competitive against adaptive on-line adversaries. □

In the *Harmonic random walk*, the probability of using a particular outgoing edge from a node is inversely proportional to its cost:

$$p(i, j) \equiv \frac{1/d(i, j)}{\sum_{k \neq i} 1/d(i, k)} .$$

(We do not permit transitions from a node to itself.) This process has been studied in [18], where the following result is proved about a Harmonic walk on any graph (not necessarily complete) with E edges.

Theorem 5.3
The Harmonic random walk has an edge expansion that is less than or equal to $2E$. □

The last result is tight; i.e., equality is achieved for certain graphs.

The Harmonic random walk does not yield, in general, the smallest possible edge expansion. In fact, the following weaker expansion condition is sufficient, for our purposes. For any path $p = i_1, i_2, \cdots, i_k$ in the graph, define

$$d(p) = d(i_1, i_2) + d(i_2, i_3) + \cdots + d(i_{k-1}, i_k)$$

and

$$h(p) = h(i_1, i_2) + h(i_2, i_3) + \cdots + h(i_{k-1}, i_k) .$$

We define the *cycle expansion* of a random walk to be $\max[h(p)/d(p)]$, where the maximum is taken over all simple cycles (closed paths) $p = i_1, i_2, \cdots, i_k, i_1$ in the graph.

Assume a random walk with cycle expansion η for a graph G. Let $d_{\max}$ be the maximum length of an edge of G. If p is a simple path from node i to node j,

$$h(p) \leq h(p) + h(j, i) \leq \eta \cdot [d(p) + d(j, i)]$$

$$\leq \eta \cdot [d(p) + d_{\max}] .$$

If p is an arbitrary path, it can be decomposed into the union of disjoint simple cycles and one simple path. Thus, for any path p in the graph (not necessarily simple),

$$h(p) \leq \eta \cdot [d(p) + d_{\max}] .$$

Consequently, we have the following.

Lemma 5.4
A probabilistic traversal algorithm based on a random walk with cycle expansion c is c-competitive against adaptive on-line adversaries. □

Coppersmith et al. [17] have recently proved the following general result.

Theorem 5.5
For any n-node graph and any distance matrix, there is a random walk with cycle expansion $n - 1$. *This is the best possible expansion.* □

Thus, the competitiveness coefficient of the traversal problem on n-node graphs is less than or equal to $n - 1$, for adaptive on-line adversaries.

- *Server systems*

The server problem is a generalization [5] of the cacheing problem. The problem is specified by a complete graph on n nodes, and a metrical distance matrix **D**. There are M mobile servers that occupy M of the nodes of the graph at any time. A request specifies a node; in response, a server must be moved to that node if no server is currently at that node. An algorithm chooses which server to move in order to satisfy successive requests in a sequence; an on-line algorithm has to decide on a move (when necessary) after each request, not knowing about future requests.

The cache problem corresponds to a server problem with a unit distance matrix. The nodes of the graph are the memory items, and the servers are the cache locations. The weighted cache problem corresponds to a server problem with a distance matrix of the form $d(i, j) = w_j$. Such a distance matrix is not metrical; however, one can obtain an equivalent problem by using the distance matrix $d(i, j) = (w_i + w_j)/2$. This matrix represents a weighted cache problem in which the cacheing algorithm incurs half

of the cost of an item when the item is loaded, and half when the item is evicted.

Formally, an on-line algorithm for the server problem is defined by a function

$$F: S \times [1, \cdots, n]^M \times [1, \cdots, n] \rightarrow S \times [1, \cdots, n]^M. \quad (9)$$

This transition function specifies the next state and the set of nodes occupied by the M servers after the request has been serviced, given the current state, the current locations of the M servers, and the request node. The request node must be occupied after the transition. The memory of the algorithm is $\log |S|$. All of the definitions and results of Section 2 hold.

Equation (9) is a generalization of the formulation for the cacheing problem given by (1). Thus, the measure of memory we use here is more lenient than the one we used to analyze cacheing algorithms, where we used the formulation given by (2) and (3).

In [5] it is shown that the competitiveness coefficient of any deterministic on-line server algorithm is at least $M/(M - m + 1)$ (one compares an on-line algorithm with M servers to an off-line algorithm with m servers). The argument actually implies the same bound for randomized algorithms with adaptive on-line adversaries.

Theorem 5.6 [5]
Let A be a (possibly randomized) on-line server algorithm for M servers on a graph with $M + 1$ nodes. Let B be an adaptive on-line adversary defined as follows. B starts with its m servers at m randomly chosen locations. At step i, B generates a request at the unique node v_i not occupied by one of A's servers (B is a cruel adversary). If B has no server at node v_i, it moves the server currently at node v_{i-1} to v_i. Then, for any sequence of random choices of A and any $i > 1$, the expected cost of step i for adversary B is $(M - m + 1)/M$ times the cost of step i for algorithm A.

For the rest of this section, we consider the case $M = m$ only.

• *Random walk algorithms for the server problem*
We now present a simple and natural memoryless on-line algorithm for the server problem. Let **P** be a matrix of transition probabilities defined on the graph. Suppose we have a request at a node r, and the on-line algorithm currently has no server at r. Let $i_1, \cdots, i_M$ be the current positions of the algorithm's servers. We choose one of the servers at random to service the request at r, according to the probability distribution induced by **P**: Server i_j is chosen with probability

$$p_j = \frac{p(r, i_j)}{\sum_{k=1}^{M} p(r, i_k)} \qquad 1 \leq j \leq M.$$

Theorem 5.7
Let $M = n - 1$. Assume that the random walk defined by the transition probabilities $p(i, j)$ has cycle expansion c. Then the adaptive on-line competitiveness coefficient of the corresponding server algorithm $C(M, M)$ is less than or equal to c.

Proof When $M = n - 1$, at all times there is exactly one node of the graph that contains none of the on-line algorithm's servers. We call this node $a(t)$ at time t. The algorithm incurs a cost only when the request at time t is at $a(t)$. We can assume, without loss of generality, that the adversary is cruel, so that the request at time t is, in fact, at node $a(t)$. Similarly, there is exactly one node not occupied by any of the servers of the adversary generating the requests; we denote this by $b(t)$. If $a(t) = b(t)$, the adversary must make a move and incur a cost in order to serve the tth request. The adversary moves a server from node $b(t + 1)$ to node $b(t)$ and incurs a cost of $d[b(t + 1), b(t)] = d[b(t), b(t + 1)]$ (**D** is symmetric).

We consider the behavior of our algorithm in a sequence of *phases*. A phase starts when the adversary must make a move because the request is at its unoccupied node. Assume that a phase starts at step t_0, when the adversary moves its server from $b(t_0 + 1)$ to $b(t_0)$. At this time, we have $a(t_0) = b(t_0)$. It is easy to see that $a(t)$ executes a random walk on the graph, choosing at each step an edge (i, j) with probability $p(i, j)$. The walk at the current phase starts at $b(t_0)$. The phase terminates when the walk reaches $b(t_0 + 1)$. The expected length of the walk from $b(t_0)$ to $b(t_0 + 1)$ is $h[b(t_0), b(t_0) + 1]$, the expected cost the on-line algorithm incurs in this phase. Summing over all phases of the sequence yields the result. □

Corollary 5.8
1. *The adaptive on-line competitiveness coefficient of the server algorithm induced by the Harmonic random walk is $C(M, M) \leq M(M + 1)$, for $M = n - 1$.*
2. *The adaptive on-line competitiveness coefficient of the server algorithm induced by the random walk described in the proof of Theorem 5.5 is $C(M, M) \leq M$, where $M = n - 1$.*

Theorem 5.9
For every $M \geq 2$, there exists a distance matrix on $M + 1$ points for which the competitiveness of the Harmonic algorithm is greater than or equal to $M(M + 1)/2$ against an oblivious adversary.

Proof Let the nodes be numbered $1, 2, \cdots, M + 1$. Let $d(1, 2) = 1$, and let all other distances be $B \gg M$. The request sequence is an infinite repetition of $(1, 3, 4, \cdots, M + 1)^L$, $(2, 3, 4, \cdots, M + 1)^L$, for a large integer L. Call the above subsequences [each of the

form $(1, 3, 4, \cdots, M + 1)^L$ or $(2, 3, 4, \cdots, M + 1)^L$] *rounds*. The adversary places one server at each of the nodes $3, \cdots, M + 1$, and never moves these $M - 1$ servers. It uses its last server to alternate between nodes 1 and 2 upon demand, to service requests at those nodes. Thus, it pays a cost of 2 per round.

How well does the Harmonic algorithm perform? Note that there is always exactly one node of the graph that is not occupied by one of Harmonic's servers; let us call this node the "hole" $a(t)$. The hole executes a random walk in the graph, always going from a node to a neighbor that is chosen in inverse proportion to its distance. We therefore ask the question, What is the expected cost incurred by the hole in this random walk in a "round trip" from node 1 to node 2 and back to node 1? (Such a round trip is a random walk from node 1 that terminates on first reaching node 1 after having visited node 2 at least once.) The ratio of this quantity to the cost of the adversary per pair of rounds (which is 2) is a lower bound on the competitiveness for Harmonic on this graph.

The answer comes from an electrical analogy studied by Chandra et al. [18], who show that the expected cost of this round trip equals $M(M + 1)$ times the effective resistance that would exist between nodes 1 and 2 if each edge in the graph were replaced by an electrical resistor whose value equaled the cost, or the distance, of that edge. A simple calculation shows that in our case, this effective resistance is $2B/(2B + M - 1)$; by our choice $B \gg M$, this is arbitrarily close to 1. □

The proofs of Theorems 5.7 and 5.9 suggest an approach to analyzing the algorithm for any value of M (regardless of its relation to n). For the remainder of this section, we study the server problem in a slightly more general setting: The requests are points in an arbitrary metric space (rather than the nodes of a finite graph with a distance matrix). We begin with M points in the space, each of which is occupied by one adversary server and one of the algorithm's servers. Thus, the adversary first makes a move and issues a request for which the algorithm incurs a cost.

Conjecture 5.10 (Lazy Adversary Conjecture)
The following (adaptive) adversary strategy results in the poorest performance for memoryless algorithms:

> *Whenever there is a point in the space at which the adversary has a server but the algorithm none, the adversary issues the next request at that point (instead of making a move and incurring a cost).*

The conjecture suggests that the ratio of the expected cost of the algorithm to that of the adversary is maximized under this adversary policy. (The conjecture is *not* true for every algorithm; we suggest only that it is true for a class of algorithms that includes memoryless algorithms). If this conjecture were proved, we could reduce the analysis of the algorithm to a phase analysis and random walk similar to that in Theorem 5.7. The result would be an upper bound of c on the adaptive competitiveness coefficient of the algorithm, where c is the expansion factor for a random walk on a graph with $M + 1$ nodes; $c = M$ for the random walk described in the proof of Theorem 5.5.

Even without the Lazy Adversary Conjecture, we can bound the performance of the Harmonic algorithm in an arbitrary metric space for the case $M = 2$.

Theorem 5.11
The adaptive on-line competitiveness coefficient of the Harmonic algorithm for the two-server problem is in the interval [7, 18].

Proof The lower bound follows from Theorem 5.9. The proof of the upper bound again uses a potential-based argument. Here we define the potential we use to prove the result; the actual methodology of the proof is similar to that in Theorems 3.1 and 4.1.

At every step, the two servers managed by Harmonic and the two managed by the adversary occupy (up to) four points in the metric space. Let m_1 and m_2 be the costs of the two perfect matchings between the two points occupied by Harmonic's servers and the two occupied by the adversary's. The potential of this configuration is defined to be $m_1m_2/(m_1 + m_2)$ (thus, it is zero when the points occupied by Harmonic's servers are exactly those occupied by the adversary).

After a request has been served, there is at least one point in the metric space where both Harmonic and the adversary have a server. We now consider what happens on the next request: We assume that the adversary first moves one of its servers (possibly by a distance zero) and then requests the point to which it has just moved. In response, Harmonic moves a server and incurs a cost (which is a random variable) and changes the potential (by an amount that is also a random variable). A detailed calculation considering three possible cases yields the result. Details are given in Appendix C. □

Manasse et al. [5] give a 2-competitive, deterministic algorithm for this problem. Their algorithm has a better competitiveness coefficient; ours is randomized but simpler, memoryless, and computationally efficient. Subsequent to our work, Berman et al. [7] have proved that our Harmonic algorithm achieves a bounded competitiveness for $M = 3$ in any metric space. Later, Grove [19] proved that Harmonic achieves a competitive ratio that is $O(M2^M)$ in any metric space and for all M. By Theorem 2.2, it follows that there is a deterministic algorithm achieving a competitiveness $2^{O(M)}$.

• *Metrical task systems*

A metrical task system (MTS) consists of a graph G with n nodes $\{1, \cdots, n\}$ and a metrical cost matrix **D**. An algorithm occupies one node of G at any given time. A task T is a vector of length n whose ith component is the cost of processing T while occupying node i; we assume that these costs are uniformly bounded. Given a sequence of tasks $T_1, T_2, \cdots, T_k$, an algorithm must choose a *schedule* $i_1, i_2, \cdots, i_k$ of nodes, where i_j is the node occupied by the algorithm at step j, while processing T_j. An on-line algorithm must choose i_j knowing only $T_1, \cdots, T_j$. The cost of a schedule is the sum of all task-processing costs and all transition costs incurred. Metrical task systems can be viewed as a generalization of server systems. [A node in the metrical task system encodes the locations of the M servers of the server problem; an M-server problem on a graph with n nodes is represented by a metrical task system with $\binom{n}{M}$ nodes.] We refer the reader to Borodin et al. [6] for details.

An algorithm (controlled by an automaton) is now defined by a function with the following form:

$$\text{Algorithm}: [1, \cdots, n] \times S \times T \rightarrow [1, \cdots, n] \times S.$$

As in the previous sections, we can define memory and the competitiveness of deterministic and randomized algorithms.

Borodin et al. give a deterministic algorithm for metrical task systems, which can be generalized as follows. Let A be a traversal algorithm for the graph of the task system. An MTS algorithm $\hat{A}$ is derived from A, as follows. Let i be the node currently occupied by $\hat{A}$, and let j be the next node visited by the traversal algorithm. $\hat{A}$ moves to j when the cumulative processing cost since entering i equals or exceeds the transition cost $d(i, j)$. [This introduces a technicality: The total cost since entering i could jump substantially above $d(i, j)$ in the course of processing a single task, thus necessitating several state changes before processing the next task. The solution given by Borodin et al. views the process as occurring in continuous time. Details omitted here may be found in [6]. Thus, the cost incurred by algorithm $\hat{A}$ approaches twice the total cost of all its moves. If A is probabilistic, $\hat{A}$ moves out of node i when the processing cost since entering i reaches the expected cost of the move out of i. If the traversal algorithm is c-competitive, the derived MTS algorithm is $2c$-competitive.

One technical problem must be addressed in order for the random-walk approach to work. The adversary has the choice of remaining at node v, not incurring any transition cost. The cost the adversary incurs between two returns of the on-line algorithm $\hat{A}$ to node v is the cost of the first move of the traversal algorithm A out of v (this is the cost of the tasks before the on-line algorithm leaves v). The cost the on-line algorithm incurs is at most twice the cost of the path up to the first return to v. We need to make sure that the ratio between these two costs is bounded. This motivates the following definition.

A traversal algorithm has *loop ratio* ℓ if, for any node v and any visit of the traversal algorithm to v, the expected cost of the loop from v back to v is at most ℓ times the expected cost of the first move out of v.

Theorem 5.12

Let A be a traversal algorithm that is c-competitive against adaptive on-line adversaries and that has loop ratio ℓ. Consider the MTS algorithm $\hat{A}$ derived from the traversal algorithm, as follows. Let i be the node currently occupied by $\hat{A}$, and let $\bar{d}$ be the expected cost of the next move by $\hat{A}$. Then $\hat{A}$ moves to the next node in the traversal A when the processing cost since entering i reaches $(\ell/c)\bar{d}$. Algorithm $\hat{A}$ is $(c + \ell)$-competitive against adaptive on-line adversaries.

Borodin et al. use a weaker condition in their traversal algorithm: Their algorithm does not necessarily have a bounded loop ratio. Theorem 5.12 should thus be viewed as a sufficient, rather than necessary, condition for $\hat{A}$ to be $(c + \ell)$-competitive.

Proof We assume, for simplicity, that the on-line algorithm incurs a cost exactly equal to $(\ell/c)\bar{d}$ before leaving node i. We can arrange this by using the continuous-time method of Borodin et al. We can also assume, without loss of generality, that the adversary moves from node i only if the on-line algorithm $\hat{A}$ reaches node i. The adversary can then decide either to move to a new node j or to remain at node i until the on-line algorithm reaches node i again. Suppose the adversary decides to stay at node i. Let $\bar{d}(i)$ be the expected cost of the first move out of i for the on-line algorithm, and let $h(i, i)$ be the expected cost of the return trip to i. Then $h(i, i) \leq \ell\bar{d}(i)$. The on-line algorithm incurs an expected cost of $c_a \leq (1 + \ell/c)h(i, i) \leq (1 + \ell/c)\ell\bar{d}(i)$, whereas the adversary incurs a cost of $c_b = (\ell/c)\bar{d}(i)$. Thus,

$$c_a - (\ell + c)c_b \leq \left(1 + \frac{\ell}{c}\right)\ell\bar{d}(i) - (\ell + c)\frac{\ell}{c}\bar{d}(i) = 0.$$

Suppose that the adversary moves to node $j \neq i$. Then the adversary incurs a cost of $c_b = d(i, j)$, whereas the on-line algorithm incurs an expected cost of $c_a = (1 + \ell/c)h(i, j)$, where $h(i, j)$ is the expected cost of the traversal from i to j. Thus,

$$\begin{aligned} c_a - (\ell + c)c_b &\leq \left(1 + \frac{\ell}{c}\right)h(i, j) - (\ell + c)d(i, j) \\ &= \left(1 + \frac{\ell}{c}\right)[h(i, j) - c \cdot d(i, j)]\,. \end{aligned}$$

□

Lemma 5.13
The Harmonic random walk has a loop ratio of n.

Proof Since $p(i, j) = [1/d(i, j)]/[\sum_k 1/d(i, k)] > 0$ for all $i \neq j$, the Markov chain defined by the transition probabilities **P** is aperiodic, and there is a unique, stationary probability distribution Φ on the nodes, defined by the equations

$$\phi_i = \sum_j \phi_j p(j, i),$$

$$\sum_i \phi_i = 1.$$

One can check, by substitution, that

$$\phi_i = \frac{\sum_j 1/d(i, j)}{\sum_{r,s} 1/d(r, s)},$$

where r and s range over all nodes. The expected cost of a move out of node i is

$$d_i = \sum_j p(i, j)d(i, j) = \frac{n - 1}{\sum_j 1/d(i, j)}.$$

Thus, $\bar{d}$, the average cost of a move in the random walk, is equal to

$$\bar{d} = \sum_i \phi_i d_i = \frac{n(n - 1)}{\sum_{r,s} 1/d(r, s)}$$

(the harmonic mean of the distances), and

$$h(i, i) = \bar{d}/\phi_i = \frac{n(n - 1)}{\sum_j 1/d(i, j)}. \quad \square$$

Thus, we obtain from the Harmonic random walk an algorithm that is $n(n + 1)/2$-competitive. The random walk traversal of Coppersmith et al. [17] (Theorem 5.5) has a loop ratio of $2(n - 1)$. Using it, they obtain an algorithm for metrical task systems that is $3(n - 1)$-competitive. By a further refinement of this construction, a $(2n - 1)$-competitive algorithm is derived in [17].

We have yet to show that such random walk algorithms are memoryless, as one must maintain a counter that accumulates the total processing cost at the current node. One can, however, replace this counter with a probabilistic counter.

Assume that the algorithm occupies node i, and let λ be the threshold for the next move (the algorithm moves to the next node when the processing costs since entering i exceed λ). For an algorithm based on our previous construction (which translates a traversal algorithm A to an algorithm $\hat{A}$ for metrical task systems), $\lambda = (c + \ell) \sum_{j \neq i} p(i, j)d(i, j)$. For a task T, let $T(i)$ be the cost of processing T at node i. Assume $T(i) \leq \lambda$; to justify this, we use the continuous-time ideas of Borodin et al. The main idea is to view the processing of tasks as occurring in continuous time. By this device, if $T(i)$ exceeds λ for the present node i, we move on to its successor node in the traversal. Let the new node have a threshold of λ'; we compare $T(i) - \lambda$ with λ' now, moving on to the successor again if $T(i) - \lambda > \lambda'$, and so on. A formal description of the process is given in [6].

We now describe our algorithm, which we call *Gambler*. Independent of previous tasks and processing costs, Gambler does the following: Given T, flip a coin with Pr[heads] $= T(i)/\lambda$; if the coin comes up heads, Gambler moves to the next node in the traversal (this next node may be chosen probabilistically); otherwise, it remains at the current node. Note that Gambler is memoryless. Lemma 4.2 can now be invoked to show that the expected processing cost incurred by the Gambler algorithm at node i is λ. This yields the following theorem.

Theorem 5.14
Let A be a c-competitive MTS algorithm of the form given in Theorem 5.12. The memoryless traversal algorithm Gambler derived from it is also c-competitive against adaptive on-line adversaries.

6. Further work

This paper leaves many open problems. We do not completely understand the relation between the two definitions of competitiveness, one using a limit at infinity, and the second using finite sequences. In particular, we have not shown that the two definitions yield the same competitiveness coefficient for cacheing against oblivious adversaries. Also, we conjecture that both definitions are equivalent for randomized algorithms with finite control.

While adaptive off-line adversaries may be more powerful than adaptive on-line adversaries against specific algorithms, they yield the same competitiveness coefficient for cacheing and for the more general server problem. It would be interesting to have a direct proof of the equivalence of these two adversaries.

It would be most interesting to find generalizations of the RFWF algorithm for the weighted cacheing problem and the general server problem, thus showing that one can do better against an oblivious adversary than against an adaptive one for these generalizations.

Finally, it has been conjectured by Manasse et al. [5] that the competitiveness coefficient of the m-server problem is m. Following an upper bound of $2^{O(m \log m)}$ due to Fiat et al. [20], Grove [19] has obtained an upper bound of $2^{O(m)}$ based on the Harmonic algorithm. Recently, Koutsoupias and Papadimitriou [21] have given an upper

bound of $2m - 1$. Much work has been directed recently at proving particular cases of this conjecture [8, 9]; it would be interesting to extend these results to the case $M \neq m$. The existence of a memoryless m-competitive algorithm against adaptive on-line adversaries for a large class of graphs has recently been proved [17]. The proof expands on the use of random walks outlined in this paper.

Appendix A: Cacheing and games

• *Lazy algorithms and cruel adversaries*

In this appendix, we prove some simplifying results on cacheing algorithms. We restrict ourselves here to adaptive adversaries (either on-line or off-line). We show that one can assume, without loss of generality, that on-line cacheing algorithms are *lazy* (changing state only when a miss occurs). This allows us to restrict our attention to *cruel* adversaries, which force a miss at each step (the name "cruel" is taken from [6]). We also show that one can assume, without loss of generality, that there are exactly $M + 1$ distinct items.

Lemma A.1

Let A be a lazy cacheing algorithm. Then A is $c(M, m)$-competitive against adaptive off-line (on-line) adversaries if and only if it is $c(M, m)$-competitive against cruel adaptive off-line (on-line) adversaries.

Proof Let B be an adaptive off-line adversary. A cruel adversary $\hat{B}$ simulates B as follows. At the end of step j, the simulating adversary $\hat{B}$ assumes the state of B at the end of step i_j, when the jth miss of A occurred. Since A does not change states at hits, $\hat{B}$ can simulate the next $i_{j+1} - i_j$ steps of the computation, until A next misses, and compute the state of B after step i_{j+1}. $\hat{B}$ then issues the next reference v_{j+1}, which is the reference B would issue at step i_{j+1}.

Consider a fixed sequence of random choices by A. If B issues the references $v_1, v_2, \cdots$, and A misses at steps $i_1, i_2, \cdots$, then $\hat{B}$ will issue the references $v_{i_1}, v_{i_2}, \cdots$, and A will miss at each step. The number of misses of A on the subsequence of references $v_{i_1}, v_{i_2}, \cdots, v_{i_j}$ equals its number of misses on the sequence of references $v_1, v_2, \cdots, v_{i_j}$; whereas the number of misses of the optimal algorithm on the subsequence is clearly less than or equal to its number of misses on the sequence. Thus A is $c(M, m)$-competitive against B only if it is $c(M, m)$-competitive against $\hat{B}$. This concludes the argument for adaptive off-line adversaries.

Suppose now that B is an adaptive on-line adversary. Then $\hat{B}$ can also simulate the on-line cacheing management of B as follows: Assume that $\hat{B}$ misses at step j. If B also misses at step i_j and $\hat{B}$ has in its cache the item evicted by B at step i_j, then $\hat{B}$ evicts that item. Otherwise, $\hat{B}$ evicts an item that is not in the cache of B at step i_j. It is easy to check that the number of misses of $\hat{B}$ on the subsequence of references $v_{i_1}, v_{i_2}, \cdots$ is less than or equal to the number of misses of B on the sequence of references $v_1, v_2, \cdots$. □

Corollary A.2

Let A be a cacheing algorithm that is $c(M, m)$-competitive against adaptive off-line (on-line) adversaries. Let $\hat{A}$ be the algorithm obtained from A by preserving the Miss *function and modifying the* Hit *function to be the identity ($\hat{A}$ does not change states on hits and behaves as A on misses). Then $\hat{A}$ is $c(M, m)$-competitive against adaptive off-line (on-line) adversaries.*

Proof By the previous lemma, it is sufficient to consider cruel adversaries; however, A and $\hat{A}$ behave identically against cruel adversaries. □

This corollary implies, for example, that the FIFO algorithm is as competitive as the LRU algorithm. Indeed, if we modify the LRU algorithm so that it modifies its state only on misses, LRU "remembers" only references that caused misses, and orders the elements in the cache according to the order in which they were loaded. This is exactly the behavior of the FIFO algorithm. Corollary A.2 does not hold for oblivious adversaries. The RFWF algorithm has a competitiveness coefficient $c(m, m) = O(\log m)$ against oblivious adversaries [12]. A lazy version of this algorithm, which does not mark entries unavailable on hits, has a competitiveness coefficient $c(m, m) \geq (m + 1)/2$. This is proved in Appendix B.

Lemma A.3

Let A be a randomized cacheing algorithm. The following two assertions are equivalent:

1. *A is $c(M, m)$-competitive against adaptive off-line (on-line) adversaries.*
2. *A is $c(M, m)$-competitive against adaptive off-line (on-line) adversaries that generate references to only $M + 1$ distinct items.*

Proof Clearly, Assertion 1 implies Assertion 2. Let A be a cacheing algorithm. Let $i_t(x)$, the *index* of item x at step t, be the location in the cache that contains x after t references; $i_t(x) = 0$ if x is not in the cache. Index $i_t(x)$ depends on the adversary and on the first t random choices of A. Let B be an adaptive adversary. We transform B into an adversary $\hat{B}$ that generates references from a set of $M + 1$ items $\{y_1, \cdots, y_{M+1}\}$. Let $\omega_1, \omega_2, \cdots, \omega_{t-1}$ be the first $t - 1$ random choices made by A. If B now generates a reference to an item x at step t, then $\hat{B}$ generates at step t a reference to the unique item y such that $i_{t-1}(y) = i_{t-1}(x)$; if B evicts x' at step t, then $\hat{B}$ evicts

the unique item y' such that $i_t(y') = i_t(x')$. Make the sequence of random choices made by A fixed. Then A misses on the sequence generated by B at the same steps it misses on the sequence generated by $\hat{B}$. The set of indices of the items in the cache of adversary B at step t is identical to the set of indices of the items in the cache of $\hat{B}$ at step t. Thus, the cache management of $\hat{B}$ is correct ($\hat{B}$ performs an eviction whenever it misses), and $\hat{B}$ misses at the same cycles at which B misses. Furthermore, if B manages its cache on-line, then so does $\hat{B}$. □

• *Alternative definitions*

We defined competitiveness in terms of the limit behavior of the cacheing algorithm on infinite request sequences. As shown below, this leads to a very natural game-theoretic formulation of competitiveness. It is often more convenient or more intuitive, however, to analyze cacheing algorithms in terms of their behavior on finite sequences of references. This leads to other definitions of competitiveness, which, fortunately, are not too different from the ones we use.

The following definition is often used for deterministic algorithms [4, 5].

Definition 2[4]

A deterministic cacheing algorithm A is $c(M, m)$-competitive if there exists a constant g such that, for any finite sequence of references $(v_1, \cdots, v_n)$,

$$C_M^A(v_1, \cdots, v_n) - c \cdot C_m^{Opt}(v_1, \cdots, v_n) \leq g. \qquad \text{(A1)}$$

In order to distinguish between the two definitions, we say that an algorithm is *c-competitive in the limit* if it fulfills (8), and *c-competitive on finite sequences* if it fulfills (A1). The following results show, however, that the need to draw the distinction seldom arises.

Clearly, (A1) implies (8); an algorithm that is c-competitive on finite sequences is also c-competitive in the limit. One can easily build a pathological algorithm that is c-competitive in the limit but not c-competitive on finite sequences, so that Definition 2 is strictly stronger. Using this stronger definition, however, does not change the competitiveness coefficient $\mathcal{C}_d(M, m)$ of cacheing, and does not change the competitiveness coefficient $\mathcal{C}_d^A(M, m)$ of deterministic cacheing algorithms A with *finite* control.

Theorem A.4

The following two assertions are equivalent:

1. *There exists a deterministic cacheing algorithm that is $c(M, m)$-competitive in the limit.*
2. *There exists a deterministic cacheing algorithm that is $c(M, m)$-competitive on finite sequences.*

[4] Definition 1 is in Section 2.

Proof Clearly, Assertion 2 implies Assertion 1. To prove the reverse, we find it convenient to relax our definitions and allow a cacheing algorithm to start with a non-empty cache. This may save at most M misses and hence affects neither definition of competitiveness. If A is an on-line cacheing algorithm, we denote by $A[s, x_1, \cdots, x_M]$ the cacheing algorithm obtained by starting A in state s, with cache contents $x_1, \cdots, x_M$.

Assume that no cacheing algorithm is $c(M, m)$-competitive on finite sequences. Then, for any cacheing algorithm A (A may start on a non-empty cache), there is a sequence of references $\xi(A) = v_1, \cdots, v_n$ such that

$$C_M^A(v_1, \cdots, v_n) - c(M, m) \cdot C_m^{Opt}(v_1, \cdots, v_n) > 1$$

(the cost of A exceeds the prescribed bound by at least one miss).

Let A be an on-line cacheing algorithm. Define inductively an infinite sequence of references $u_1, u_2, \cdots$ as follows. We initially determine $\xi(A)$ for the initial state and cache contents $[s, x_1, \cdots, x_M]$. After applying this $\xi(A)$, we determine the sequence ξ for the state that results, and so on.

We have

$$C_M^A(v_1, \cdots, v_{n_i}) - c(M, m) \cdot C_m^{Opt}(v_1, \cdots, v_{n_i}) > i,$$

so that A is not $c(M, m)$-competitive in the limit. Thus, the complement of Assertion 2 implies the complement of Assertion 1. □

The theorem implies that the value of the competitiveness coefficient $\mathcal{C}_d(M, m)$ is the same for both definitions of competitive cacheing algorithms. We also have the following result.

Theorem A.5

Let A be a cacheing algorithm with finite control. Then the following two assertions are equivalent:

1. *A is $c(M, m)$-competitive in the limit.*
2. *A is $c(M, m)$-competitive on finite sequences.*

Proof Clearly, Assertion 2 implies Assertion 1. Assume that Assertion 2 does not hold. Then, given any fixed k, there is a sequence of references $v_1, \cdots, v_n$ such that

$$C_M^A(v_1, \cdots, v_n) - c(M, m) \cdot C_m^{Opt}(v_1, \cdots, v_n) > k.$$

Let s_i be the state of A after references $v_1, \cdots, v_i$. Let

$$c_i \equiv C_M^A(v_1, \cdots, v_i) - c(M, m) \cdot C_m^{Opt}(v_1, \cdots, v_i).$$

We have $c_0 = 0$, $c_n > k$. Then, if k is sufficiently large with respect to $|S|$, the number of states, and with respect to the product $m \cdot c(M, m)$, there are two indices i, j, where $i < j$, such that $s_i = s_j$ and $c_j - c_i \geq m \cdot c(M, m) + 1$. Let c^A be the number of misses of algorithm A from

cycles $i+1$ to j, and let c^{Opt} be the number of misses of the adversary for these cycles. Consider now the sequence of references $v_1, \cdots, v_j, v_{i+1}, \cdots, v_j, v_{i+1}, \cdots$ ($v_1, \cdots, v_j$ followed by an indefinite repetition of $v_{i+1}, \cdots, v_j$). Let $\hat{c}_i$ be defined as above for the new sequence of references. Algorithm A goes through the same sequence of states on each repetition of the sequence of references $v_{i+1}, \cdots, v_j$ and has c^A misses on each such segment. The optimal algorithm has at most $c^{Opt} + m$ misses on each such segment. It follows that

$$\hat{c}_{j+(q+1)(j-i)} - \hat{c}_{j+q(j-i)} \geq c_j - c_i - c(M, m) \cdot m \geq 1,$$

for any positive integer q, so that

$$\limsup_{n \to \infty} \hat{c}_n = \infty. \qquad \square$$

This theorem implies that if A is a deterministic cacheing algorithm with finite control, the value of the competitiveness coefficient $\mathcal{C}_d^A(M, m)$ is the same for both definitions of competitive cacheing algorithms. All of the algorithms considered in this paper have finite control.

We wish to extend the previous definitions and results to randomized cacheing algorithms. What is the "correct" definition of a competitive randomized cacheing algorithm on finite sequences? One might be tempted to use a condition similar to that used for infinite sequences, defining A to be $c(M, m)$-competitive if there exists a constant d such that, for any n and an adversary B of the suitable type, $C_M^A(v_1, \cdots, v_n) - c(M, m) \cdot C_m^B(v_1, \cdots, v_n) < d$, a.s. This, however, is too strong a condition: It would rule out all randomized cacheing algorithms considered in this paper. Rather than forbidding bad worst-case behavior that occurs with small probability, we require good average behavior (where the average is over the random choices of the algorithm).

Definition 3

*A randomized cacheing algorithm A is $c(M, m)$-*competitive on finite sequences against adaptive off-line adversaries *if there exists a constant d such that*

$$\mathrm{E}[C_M^A(v_1, \cdots, v_n) - c(M, m) \cdot C_m^{Opt}(v_1, \cdots, v_n)] < d$$

for any finite sequence of references $v_1, v_2, \cdots, v_n$ generated adaptively by an adversary.

*A randomized cacheing algorithm A is $c(M, m)$-*competitive on finite sequences against adaptive on-line adversaries *if there exists a constant d such that*

$$\mathrm{E}[C_M^A(v_1, \cdots, v_n) - c(M, m) \cdot C_m^B(v_1, \cdots, v_n)] < d$$

for any finite sequence of references $v_1, v_2, \cdots, v_n$ generated adaptively by an adversary and any on-line cacheing algorithm B (the moves of B depend on previous random choices of A).

*A randomized cacheing algorithm A is $c(M, m)$-*competitive on finite sequences against oblivious adversaries *if there exists a constant d such that*

$$\mathrm{E}[C_M^A(v_1, \cdots, v_n) - c(M, m) \cdot C_m^{Opt}(v_1, \cdots, v_n)] < d$$

for any finite sequence of references $v_1, v_2, \cdots, v_n$.

All three definitions coincide when algorithm A is deterministic, coinciding then with (A1).

The definition for oblivious adversaries has been used by Fiat et al. [12] and Ben-David et al. [11]. The latter paper also defines competitiveness against adaptive adversaries, using seemingly more powerful adversaries. Rather than stopping after n references (n fixed), their adversaries can adaptively decide when to halt. A cacheing algorithm A is $c(M, m)$-competitive against such an adaptive off-line adversary if

$$\mathrm{E}[C_M^A(v_1, \cdots, v_s) - c(M, m) \cdot C_m^{Opt}(v_1, \cdots, v_s)] < d$$

for any adaptively generated sequence of references $v_1, v_2, \cdots$ and any adaptive "stopping time" s. A similar definition is used for adaptive on-line adversaries. The ability to stop adaptively, however, does not add power to the adversary. The reason is as follows. If an adversary always stops after finitely many references (s is always finite), there is a fixed bound n such that the adversary stops after at most n references ($s \leq n$). Rather than stopping after s steps, the adversary can continue repeating the last reference, up to step n. This causes no further misses to either the algorithm or the adversary.

While the new definitions of competitiveness may change the competitiveness coefficients of specific algorithms, they do not change the competitiveness coefficients of cacheing for adaptive adversaries.

Theorem A.6

The competitiveness coefficient $\mathcal{C}_{af}(M, m)$ of cacheing for adaptive off-line [$\mathcal{C}_{an}(M, m)$ for adaptive on-line] algorithms does not change if competitiveness on finite sequences is used, rather than competitiveness in the limit.

Proof The claim for adaptive off-line adversaries follows from the fact that the two definitions are equivalent for deterministic algorithms (Corollary A.4), and the fact that for each definition, $\mathcal{C}_{af}(M, m) = \mathcal{C}_d(M, m)$ (Theorem 2.1, and the similar theorem for competitiveness on finite sequences proved by Ben-David et al. [11]. (The claim can also be proved with the argument given below for adaptive on-line algorithms.)

Now consider adaptive on-line adversaries. By Corollary A.2, we can restrict ourselves to cruel adversaries, which cause a miss at each step. (It is easily seen that this corollary applies to either definition of competitiveness.)

Let $\lambda > 0$ be a positive real number and $n > 0$ be a positive integer. We consider two mutually exclusive (and exhaustive) cases:

Case 1: There exists an on-line cacheing algorithm A such that, for any cruel adversary B against A, the expected number of misses of B on the first n references is $\geq \lambda n$.

Let $\hat{A}$ be a cacheing algorithm that simulates A for n steps, then reinitializes to the initial state of A and starts again. Then, for any cruel adversary B against $\hat{A}$, any non-negative integer k, and any sequence of random choices of A in the first kn steps, B has an expected number of misses of at least $\lambda n - m$ at steps $kn + 1, kn + 2, \cdots, (k + 1)n$. Let X_k be the random variable defined by

$$X_k = C_M^A(v_1, \cdots, v_k) - c \cdot C_m^B(v_1, \cdots, v_k),$$

and let

$$Y_k = X_{kn} - X_{(k-1)n}.$$

Then

$$(1 - c)n \leq Y_k \leq n,$$

and

$$\mathrm{E}[Y_k | Y_1, \cdots, Y_{i-k}] \leq n - c(\lambda n - m).$$

Thus, if $c > n/(\lambda n - m)$, so that $n - c(\lambda n - m) < 0$, then

$$\lim_{n \to \infty} \mathrm{E}\left[\sum_{k=1}^{n} Y_k\right] = -\infty,$$

and, by Lemma 2.3,

$$\sum_{k=1}^{\infty} Y_k = -\infty, \text{ a.s.}$$

$\hat{A}$ is c-competitive against B according to both definitions. It follows that

$$\mathscr{C}_{\mathrm{an}}(M, m) \leq \frac{1}{\lambda - (m/n)} \tag{A2}$$

according to both definitions.

Case 2: For any on-line cacheing algorithm A, there exists a cruel adversary B against A that has an expected number of misses less than λn on the first n references.

Let A be an on-line cacheing algorithm. There exists a cruel adversary B against A such that, for any sequence of random choices of A in the first kn steps, B has an expected number of misses less than λn at steps $kn + 1, kn + 2, \cdots, (k + 1)n$. Define X_k and Y_k as before. Then

$$(1 - c)n \leq Y_k \leq n,$$

and

$$\mathrm{E}[Y_k | Y_{k-1}, \cdots, Y_1] > n - c\lambda n.$$

Thus, if $c < 1/\lambda$, A is not c-competitive against B according to either definition. It follows that

$$\mathscr{C}_{\mathrm{an}}(M, m) \geq 1/\lambda, \tag{A3}$$

according to both definitions. Since either (A2) or (A3) holds for any n and λ, it follows that both definitions yield the same value for $\mathscr{C}_{\mathrm{an}}(M, m)$. □

This proof also shows that

$$\mathscr{C}_{\mathrm{an}}(M, m) = \inf \mathscr{C}_{\mathrm{an}}^{A}(M, m),$$

where the infimum is taken over all cacheing algorithms *A with finite control*. In fact, it is sufficient to consider algorithms that restart from the initial state after n cycles, for some fixed n. The same result holds for adaptive off-line adversaries. The infimum is actually achieved by finite-control cacheing algorithms, both for adaptive on-line and adaptive off-line adversaries.

We conjecture that a result similar to Theorem A.5 can be shown for randomized algorithms with finite control: Namely, such an algorithm is $c(M, m)$-competitive in the limit against adaptive on-line (off-line) adversaries if and only if it is $c(M, m)$-competitive on finite sequences against such adversaries. All of the algorithms considered in this paper have finite control, and the two definitions of competitiveness coincide for all of them. However, we do not have a general proof of the conjecture.

- *The cacheing game*

The competitiveness of deterministic cacheing algorithms can be analyzed in terms of an infinite *cacheing game* played between the cache manager (player a) and the adversary (player b). This game-theoretic approach yields some interesting relations between the various competitiveness coefficients discussed above.

Formally, such a game is described by a tuple $\mathscr{G}$: $\langle X, s, f, X_a, X_b, \mathfrak{S}_a, \mathfrak{S}_b, \mathfrak{X}, \mathfrak{X}_a, \mathfrak{X}_b \rangle$. X is the infinite set of *positions* in the game. The game positions form a tree: $s \in X$ is the root of the tree, which is the *initial game position*; $f: X - \{s\} \to X$ is the predecessor function; $f^{-1}(x)$ is the set of *successors* to position x. We assume that the game never terminates; i.e., $f^{-1}(x) \neq \emptyset$, for any $x \in X$. (Each position $x \in X$, except for s, has a single predecessor and a nonempty set of successors.) The set of positions X is partitioned into two subsets, X_a and X_b ($X_a \cap X_b = \emptyset$, $X_a \cup X_b = X$). X_a is the set of positions to which player a can move, and X_b is the set of positions to which player b can move. (Every $x \in X_a$ is the predecessor of one or more positions $y \in X_b$, and vice versa. No $x \in X_a$ has a predecessor in X_a; the same holds for positions in X_b.) A *play* is an infinite path in the game tree, starting at the root, i.e., a sequence $x_0, x_1, \cdots$ of positions, alternately in X_a and X_b, such that $x_0 = s$

and $x_i = f(x_{i+1})$, for any i. $\mathfrak{X}$ is the set of all plays associated with game $\mathcal{G}$; it is partitioned into two subsets, $\mathfrak{X}_a$ and $\mathfrak{X}_b$ ($\mathfrak{X} = \mathfrak{X}_a \cup \mathfrak{X}_b$, $\mathfrak{X}_a \cap \mathfrak{X}_b = \emptyset$). $\mathfrak{X}_a$ is the set of *winning plays for player a*, and $\mathfrak{X}_b$ is the set of *winning plays for player b*. A *pure strategy* A for player a is a function that associates with each position $x \in X_a$ a move to a successor position: $A: X_a \to X_b$, and $A(x) \in f^{-1}(x)$, for any $x \in X_a$ ($f \circ A = \mathbf{id}$). Similarly, a pure strategy B for player b is a function $B: X_b \to X_a$ such that $f \circ B = \mathbf{id}$. $\mathfrak{S}_a$ is the set of pure strategies that player a can use in the game, and $\mathfrak{S}_b$ is the set of pure strategies available to player b. The game is a *game with perfect information* if $\mathfrak{S}_a$ ($\mathfrak{S}_b$) contains all pure strategies of player a (b); otherwise, it is a game with partial information.

Each pair of pure strategies A and B defines a play [denoted $play(A, B)$] $x_0, x_1, \cdots$, in which $x_0 = s$ and, for each i, $x_{i+1} = A(x_i)$ if $x_i \in X_a$, or $x_{i+1} = B(x_i)$ if $x_i \in X_b$. A is a *winning strategy for player a* if for each strategy $B \in \mathfrak{S}_b$, $play(A, B) \in \mathfrak{X}_a$. A winning strategy for player b is similarly defined.

We consider games in which costs are associated with positions by a real-valued function $Val: X \to \mathbf{R}$. The set of winning plays for player a is defined as

$$\mathfrak{X}_a = \{(x_0, x_1, \cdots) \in \mathfrak{X}: \limsup_{n\to\infty} Val(x_n) < \infty\}. \qquad \text{(A4)}$$

Thus, A is a *winning strategy* for player a if, for any strategy $B \in \mathfrak{S}_b$,

$$\limsup_{n\to\infty} Val(x_n) < \infty,$$

where $(x_0, x_1, \cdots) = play(A, B)$.

In the cacheing game, the two players are the cache manager and the adversary. Players alternate moves: The adversary selects the next reference; then the cache manager updates the cache contents, if necessary. A position is defined by the sequence of references leading to that position and the sequence of moves made by the cache manager. The value of a position x reached by a sequence of references $v_1, v_2, \cdots, v_n$ is equal to

$$Val(x) = C_M^a(v_1, \cdots, v_n) - c \cdot C_m^{Opt}(v_1, \cdots, v_n),$$

where $C_M^a(v_1, \cdots, v_n)$ is the cost paid by the cache manager, and $C_m^{Opt}(v_1, \cdots, v_n)$ is the (optimal) cost paid by the adversary for the sequence of references. The value c is a parameter of the game. This is a game of perfect information. Note that we impose no restriction on the computing power or memory of the cache manager: It has perfect recall of the past. Clearly, a $c(M, m)$-competitive algorithm for cache management defines a winning strategy for player a in the cacheing game, with parameter $c = c(M, m)$, and vice versa.

A game is *strictly determinate* if either player a or player b has a winning strategy. While finite-tree games with perfect information are always strictly determinate, infinite games need not be so, in general. However, various conditions on the topology of the set of winning plays are known to imply strict determinateness (see [22–25]).

Let $T = \langle X, f \rangle$ be an infinite tree, and let $\mathfrak{X}$ be the set of (infinite) paths of T that have one end point at the root. Given a position x of T, we define $\mathfrak{U}(x) \subset \mathfrak{X}$ to be the set of paths that traverse position x. A topology is induced on $\mathfrak{X}$ (the Hausdorff topology) by taking the family of sets $\mathfrak{U}(x_i)$, $i = 0, 1, \cdots$ to be a basis for the neighborhoods of a path $\xi = (x_0, x_1, \cdots)$. A set is *Borel* if it belongs to the σ-algebra generated by the open sets. Martin [25] has shown that an infinite game with perfect information is strictly determinate if the set of winning plays is Borel.

The set of winning plays defined by (A4) for a game with costs associated with positions is F_σ (countable union of closed sets) and, thus, is Borel. Indeed,

$$\mathfrak{X}_a = \bigcup_i \bigcup_j \bigcap_{k \geq j} \{(x_0, x_1, \cdots): Val(x_k) \leq i\} .$$

It follows from the result of Martin [25] that games with costs are strictly determinate. (The strict determinacy for F_σ is proved by Wolfe [24], using weaker set theoretical assumptions than needed for the more general result of Martin [25].) In particular, the cacheing game is strictly determinate.

Mixed strategies

We now define mixed strategies, which correspond to probabilistic cacheing algorithms in the cacheing game. Let $\mathcal{G} = \langle X, s, f, X_a, X_b, \mathfrak{S}_a, \mathfrak{S}_b, \mathfrak{X}_a, \mathfrak{X}_b \rangle$ be an infinite tree game. A *mixed strategy* for player a is a probability distribution on the set $\mathfrak{S}_a$ of pure strategies for that player. The distribution is defined over the σ-field generated by the family of sets $\{\{A: A(x) = y\}, x \in X_a, y \in f^{-1}(x)\}$ (we assume that $\mathfrak{S}_a$ is measurable). A mixed strategy for player b is defined in a similar manner. Given a pair of mixed strategies A, B for both players, one obtains a probability distribution on the set $\mathfrak{X}$ of plays, induced by the mapping $(A, B) \to play(A, B)$: the probability of the set $\mathfrak{U}(x)$ is the probability that $play(A, B)$ reaches position x.

The mixed strategy A is a *winning strategy for player a* if $\Pr[play(A, B) \in \mathfrak{X}_a] = 1$ (player a wins a.s.) for any (mixed) strategy $B \in \mathfrak{S}_b$. It is sufficient to consider, in the definitions above, only pure adversary strategies: A mixed strategy A wins almost surely against any mixed strategy B if and only if it wins almost surely against any pure strategy B. In a game with costs, a mixed strategy A for player a is a winning strategy if for any (mixed) strategy $B \in \mathfrak{S}_b$,

$$\Pr[\limsup_{n\to\infty} Val(x_n) < \infty] = 1, \quad \text{(A5)}$$

where $(x_0, x_1, \cdots) = play(A, B)$.

A randomized cacheing algorithm corresponds to a mixed strategy for the cache manager in the cacheing game. An adaptive off-line adversary corresponds to a strategy for the adversary in this game. By (A5) and (7), a randomized algorithm A is $c(M, m)$-competitive against an adaptive off-line adversary if and only if the corresponding strategy is a winning strategy in the cacheing game with parameter $c = c(M, m)$.

The same game-theoretical model also applies to the other types of adversaries. However, these are not games with perfect information. In the case of an oblivious adversary, the adversary has no knowledge of the moves of the cache manager; the strategy choice of the adversary at a position x is restricted to depend on only its own previous moves. In the case of an adaptive on-line algorithm, a position represents the sequence of references leading to that position and the sequence of moves on both caches (of both the on-line algorithm and the adversary). The cache manager, however, has no knowledge of the contents of the adversary cache; its strategy choices are restricted to depend on only the previous references and its own previous moves. The results on strict determinacy of games do not apply to games with partial information.

The following theorem shows that mixed strategies do not outperform pure strategies.

Theorem A.7
Let A be a mixed winning strategy for player a in a strictly determinate game. Then, there is a pure winning strategy $\hat{A}$ for player a in this game.

Proof Assume, by contradiction, that there is no such pure strategy. Then, since the game is strictly determinate, player b has a pure winning strategy B. For each pure strategy A' of the first player, $play(A', B) \in \mathfrak{X}_b$. But this implies that, for each mixed strategy A'' of player a, $\Pr[play(A'', B)] \in \mathfrak{X}_b = 1$, and A is not a winning strategy. □

This theorem, when applied to cacheing games, implies Theorem 2.1. We turn now to the proof of Theorem 3.5.

Proof of Theorem 3.5 (Any memoryless on-line cacheing algorithm has an oblivious competitiveness coefficient greater than or equal to m when $m = M$.) A memoryless cacheing algorithm is a probability distribution $\{p_1, \cdots, p_M\}$, where p_i is the probability that the item at location i is evicted on a miss. Consider a randomly chosen sequence of references consisting of a sequence of rounds. The kth round is of the form $(a_1, \cdots, a_m)^k$; the set of m items referred to at round k is obtained by choosing an item from round $k - 1$ uniformly at random and replacing it with any new item.

During each round following the first, the adversary has one miss. Let $a_1, \cdots, a_m$ be the items accessed at round $k - 1$, and let $a_1, \cdots, a_{i-1}, \hat{a}_i, a_{i+1}, \cdots, a_m$ be the items accessed at round k. With probability going to 1 as $k \to \infty$, the on-line algorithm starts round k with $a_1, \cdots, a_m$ in the cache. Assume, without loss of generality, that item a_i occupies location i in the cache. The algorithm misses during round k until it evicts the item in location i; the expected number of misses is $(1/p_i)[1 - (1 - p_i)^k]$. The expected number of misses for i chosen uniformly at random (given that $a_1, \cdots, a_m$ are in the cache at the start of the round) is at least $(1/m) \sum_{i=1}^{m} (1/p_i)[1 - (1 - p_i)^k]$. This is minimized at $m\{1 - [(m - 1)/m]^k\}$ (when all the p_is are equal). Thus, the ratio between the expected on-line cost and off-line cost for such a randomly chosen sequence of references has a limit greater than or equal to m. □

Appendix B: Lazy random flush when full

Let *lazy random flush when full* (LRFWF) be the algorithm that behaves like RFWF, except that it does not change state on hits. When a miss occurs, both algorithms select an available entry for eviction; if there are no available entries, all entries are marked available before the eviction. Then, the missing item is loaded and marked unavailable.

Theorem B.1
The competitiveness coefficient of LRFWF against oblivious adversaries is $c(m, m) \geq (m + 1)/2$.

(Recall that the competitiveness coefficient of RFWF against oblivious adversaries is $2H_m$.)

Proof Assume that the on-line algorithm and the adversary both contain in their caches items $1, 2, \cdots, m$, and that all entries in the cache of LRFWF are unavailable. Consider a sequence of references of the form $2, 3, \cdots, m, m + 1$, repeated indefinitely. The off-line algorithm incurs one miss on such a sequence. Algorithm LRFWF misses at least once on each repetition of $(2, 3, \cdots, m)$, until item 1 is evicted.

Assume that k of the entries ($k > 0$) in the cache of LRFWF are available and that the entry containing item 1 is unavailable. Thus, LRFWF will have k misses, until all available entries are made unavailable. Then all entries are marked available. Finally, at each miss, algorithm LRFWF draws one random sample from the set of available entries, in a process of sampling without replacement. The process terminates when item 1 is found. The expected number of samples drawn is $(m + 1)/2$, and the maximum number of samples drawn is m. Consequently, LRFWF incurs an

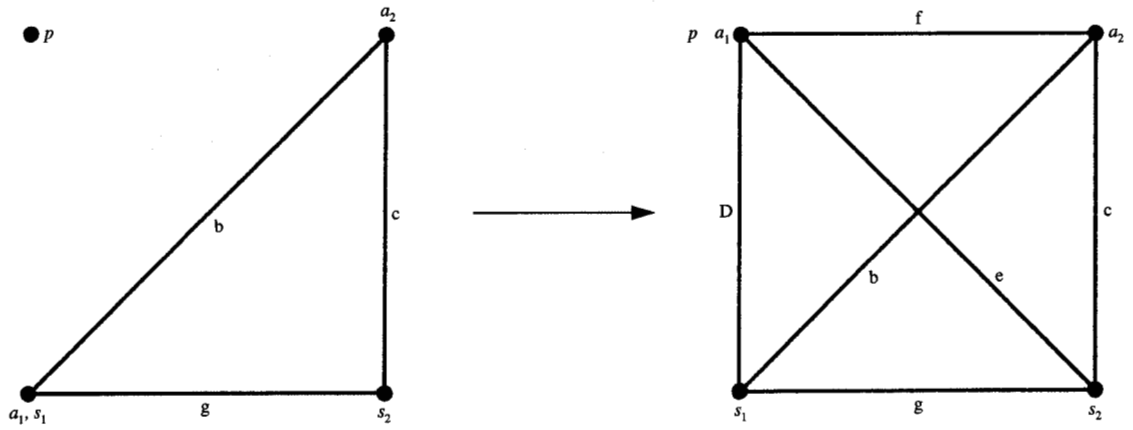

Figure 1

Case 1 in proof of Theorem 5.11, before and after the adversary's move. (Letter labels on edges represent cost of move along edge.)

average of $(m + 1)/2$ misses and a maximum of m extra misses.

Consider now an infinite sequence of references of the form $(1, 2, \cdots, m)^{2m}$, $(2, \cdots, m + 1)^{2m}$, repeated indefinitely. The adversary has one miss on each segment. LRFWF evicts item 1 from the cache when satisfying the sequence of references $(2, \cdots, m + 1)^{2m}$. Thus, a miss occurs on item 1 during the sequence of references $(1, 2, \cdots, m)^{2m}$, and item 1 is in the cache and marked unavailable at the end of the sequence. Similarly, item $m + 1$ is in the cache and marked unavailable when the sequence $(2, \cdots, m + 1)^{2m}$ ends. It follows that the expected number of misses of LRFWF on each segment is at least $(m + 1)/2$. □

Appendix C: Harmonic algorithm for two servers

In this appendix, we give the details of the proof of Theorem 5.11.

Let us denote the points of the metric space occupied by the Harmonic algorithm's servers by s_1 and s_2, and the points occupied by the two servers of the adaptive on-line adversary by a_1 and a_2. The following observations facilitate the proof:

- The adversary can defer moving a server to a new point until the adversary is about to place a request on that point.
- After Harmonic services a request, there is at least one point occupied by both a server of Harmonic and a server of the adversary. Without loss of generality, we assume that this point is currently occupied by a_1 and s_1.

Let $d(a_i, s_j)$ be the distance between adversary server a_i and Harmonic server s_j, where i and j assume the values 1 or 2. The potential function we use in the analysis is

$$\Phi = \frac{M_1 M_2}{M_1 + M_2},$$

where $M_1 = d(a_1, s_1) + d(a_2, s_2)$ and $M_2 = d(a_1, s_2) + d(a_2, s_1)$ are the costs of the two perfect matchings between Harmonic's servers and those of the adversary. We show that on each request the expectation of

$$\text{(Harmonic's cost)} - 6 \cdot \text{(adversary's cost)} + 4 \cdot \Delta\Phi \quad \text{(C1)}$$

is less than or equal to zero.

Before each request, the adversary first moves either a_1 or a_2 a distance D to a new position (if it moves neither server, the following analysis holds with D set to zero). We consider two cases, depending on whether a_1 or a_2 is moved by the adversary. We now analyze what happens during the next request. The following fact is useful: For positive reals x, y, z,

$$\frac{xy}{x+y} \leq \frac{x(y+z)}{x+y+z}. \quad \text{(C2)}$$

Case 1: The adversary moves a_1 to point p and places the next request on p. This case is illustrated in **Figure 1**.

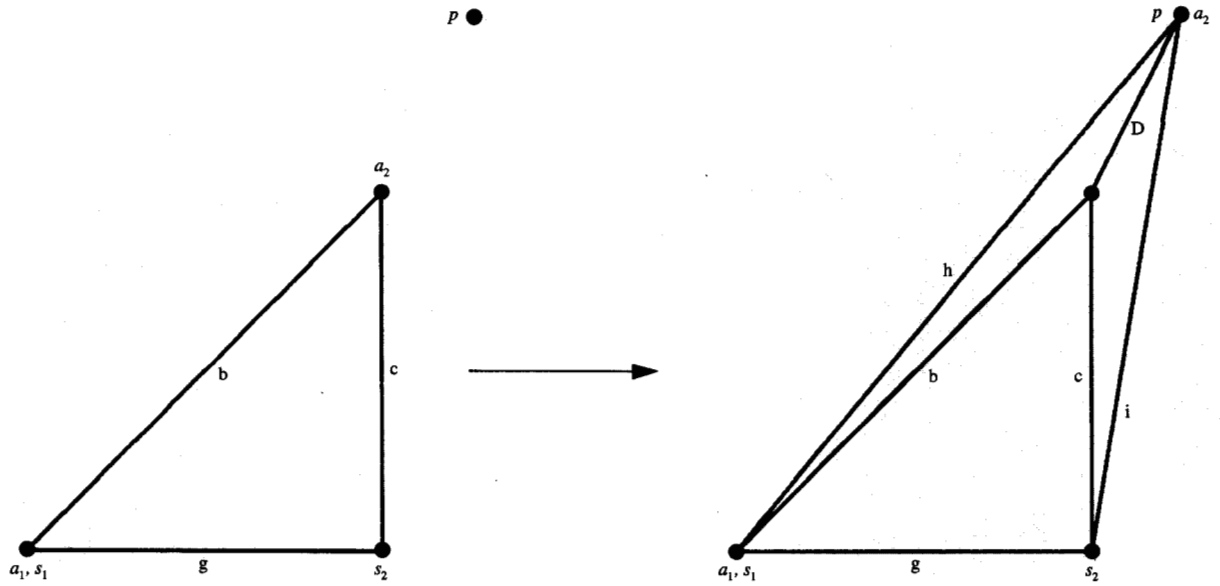

Figure 2

Case 2 in proof of Theorem 5.11, before and after the adversary's move.

The adversary pays a cost D on this move. With probability $e/(D + e)$, the Harmonic algorithm uses server s_1 to service this request (paying a cost D in the process), and with probability $D/(D + e)$, it uses s_2 (paying a cost e in the process). The expected cost incurred by Harmonic is thus $2De/(D + e)$. At the end of the previous request, $M_1 = c$ and $M_2 = g + b$, so that Φ is $c(g + b)/(c + g + b)$. If Harmonic were to use s_1 to serve the request, Φ would become $(e + f)c/(e + f + c)$. If it were to use s_2, the new value of Φ would be $(D + f)b/(D + f + b)$. Thus, the expectation of $\Delta\Phi$ is

$$\frac{D}{D+e}\cdot\frac{(D+f)b}{D+f+b}+\frac{e}{D+e}\cdot\frac{(e+f)c}{e+f+c}-\frac{c(g+b)}{c+g+b}. \tag{C3}$$

Using the triangle inequality in the denominators of the three terms of (C3) (respectively using $f \le c + e$, $f \le b + D$, and $g \le D + e$) together with (C2) results in

$$\mathrm{E}[\Delta\Phi] \le \frac{bD + ce - gc}{b + c + D + e}.$$

Finally, using the triangle inequality $e \le g + D$ verifies that (C1) holds.

Case 2: The adversary moves a_2 to point p and places the next request on p. This case is illustrated in **Figure 2.**

This time the expected cost incurred by Harmonic is $hi/(h + i)$. Once again, we can verify (C1), using the triangle inequality together with (C2). This completes the proof. □

Acknowledgments

We thank Allan Borodin, Marek Chrobak, Don Coppersmith, Anna Karlin, Howard Karloff, Nick Reingold, and Barbara Simons for their helpful comments.

References

1. J. R. Bitner, "Heuristics That Dynamically Organize Data Structures," *SIAM J. Computing* **8**, No. 1, 82–110 (1979).
2. G. Gonnet, J. I. Munro, and H. Suwanda, "Towards Self-Organizing Linear Search," *SIAM J. Computing* **10**, No. 3, 613–637 (1981).
3. D. D. Sleator and R. E. Tarjan, "Amortized Efficiency of List Update and Paging Rules," *Commun. ACM* **28**, 202–208 (February 1985).
4. A. R. Karlin, M. S. Manasse, L. Rudolph, and D. D. Sleator, "Competitive Snoopy Caching," *Algorithmica* **3**, No. 1, 70–119 (1988).

5. M. S. Manasse, L. A. McGeoch, and D. D. Sleator, "Competitive Algorithms for Server Problems," *J. Algorithms* **11**, 208–230 (1990).
6. A. Borodin, N. Linial, and M. Saks, "An Optimal Online Algorithm for Metrical Task Systems," *J. ACM* **39**, 745–763 (1992).
7. P. Berman, H. J. Karloff, and G. Tardos, "A Competitive 3-Server Algorithm," *Proceedings of the 1st ACM–SIAM Symposium on Discrete Algorithms*, San Francisco, 1990, pp. 280–290.
8. M. Chrobak, H. J. Karloff, T. Payne, and S. Vishwanathan, "New Results on Server Problems," *Proceedings of the 1st ACM–SIAM Symposium on Discrete Algorithms*, San Francisco, 1990, pp. 291–300.
9. M. Chrobak and L. L. Larmore, "An Optimal Online Algorithm for k Servers on Trees," *SIAM J. Computing* **20**, 144–148 (1991).
10. K. So and R. N. Rechtschaffen, "Cache Operations by MRU Change," *IEEE Trans. Computers* **37**, No. 6, 700–709 (June 1988).
11. S. Ben-David, A. Borodin, R. M. Karp, G. Tardos, and A. Wigderson, "On the Power of Randomization in On-Line Algorithms," *Algorithmica* **11**, No. 1, 2–14 (1994).
12. A. Fiat, R. M. Karp, M. Luby, L. A. McGeoch, D. D. Sleator, and N. Young, "Competitive Paging Algorithms," *J. Algorithms* **12**, 685–699 (1991).
13. L. A. Belady, "A Study of Replacement Algorithms for Virtual Storage Computers," *IBM Syst. J.* **5**, 78–101 (1966).
14. P. Raghavan and M. Snir, "Memory Versus Randomization in On-Line Algorithms," *16th International Colloquium on Automata, Languages, and Programming*, Vol. 372 of *Lecture Notes in Computer Science*, Springer-Verlag, New York, July 1989, pp. 687–703. Revised version available as *IBM Research Report RC-15840*, June 1990.
15. L. A. McGeoch and D. D. Sleator, "A Strongly Competitive Randomized Paging Algorithm," *Algorithmica* **6**, 816–825 (1991).
16. M. Loève, *Probability Theory*, Vol. 2, Springer-Verlag, New York, 1978.
17. D. Coppersmith, P. Doyle, P. Raghavan, and M. Snir, "Random Walks on Weighted Graphs, and Applications to On-Line Algorithms," *J. ACM* **40**, 454–476 (1993).
18. A. K. Chandra, P. Raghavan, W. L. Ruzzo, R. Smolensky, and P. Tiwari, "The Electrical Resistance of a Graph Captures Its Commute and Cover Times," *Proceedings of the 21st Annual ACM Symposium on Theory of Computing*, Seattle, May 1989, pp. 574–586.
19. E. Grove, "The Harmonic Online k-Server Algorithm Is Competitive," *Proceedings of the 23rd Annual ACM Symposium on Theory of Computing*, New Orleans, 1991, pp. 260–266.
20. A. Fiat, Y. Rabani, and Y. Ravid, "Competitive k-Server Algorithms," *Proceedings of the 31st Annual IEEE Symposium on Foundations of Computer Science*, St. Louis, 1990, pp. 454–463.
21. E. Koutsoupias and C. H. Papadimitriou, "On the k-Server Conjecture," *Proceedings of the 26th Annual ACM Symposium on Theory of Computing*, Montreal, 1994, pp. 507–511.
22. D. Gale and F. M. Stewart, "Infinite Games with Perfect Information," *Contributions to the Theory of Games, Vol. II, Annals of Mathematics Studies, 28*, W. H. Kuhn and A. W. Tucker, Eds., Princeton University Press, Princeton, NJ, 1953, pp. 245–266.
23. M. Davis, "Infinite Games of Perfect Information," *Advances In Game Theory, Annals of Mathematics Studies, 52*, M. Dresher, L. S. Shapley, and A. W. Tucker, Eds., Princeton University Press, Princeton, NJ, 1964, pp. 85–101.
24. P. Wolfe, "The Strict Determinateness of Certain Infinite Games," *Pacific J. Math.* **5**, Supplement I, 841–847 (1955).
25. D. A. Martin, "Borel Determinacy," *Ann. Math.* **102**, 363–371 (1975).

Received August 27, 1993; accepted for publication March 16, 1994

Prabhakar Raghavan *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (PRAGH at YKTVMV, pragh@watson.ibm.com).* Dr. Raghavan received his Ph.D. degree in electrical engineering and computer science from the University of California at Berkeley in 1986. Since then, he has been with the Theory of Computation group of the Mathematical Sciences Department at the IBM Thomas J. Watson Research Center. His interests are centered on combinatorial and randomized algorithms for problems in parallel computation, computational geometry, scheduling, and graph theory.

Marc Snir *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (SNIR at YKTVMV, snir@watson.ibm.com).* Dr. Snir received the Ph.D. degree in mathematics from the Hebrew University of Jerusalem in 1979. He is a senior manager at the IBM Thomas J. Watson Research Center, where he leads research on scalable parallel software. He worked at New York University on the NYU Ultracomputer project from 1980 to 1982 and worked at the Hebrew University of Jerusalem from 1982 to 1986. Dr. Snir was recently involved in the definition of high-performance FORTRAN; he is now working on the definition of the Message Passing Interface Standard in parallel computing. He has published papers in the areas of computational complexity, parallel algorithms, parallel architectures, interconnection networks, and parallel programming environments. Dr. Snir is a member of the IBM Academy of Technology, a senior member of the IEEE, and a member of ACM and SIAM.