# Memory Versus Randomization in On-line Algorithms

(Extended Abstract)

Prabhakar Raghavan*
Marc Snir*

Those who do not remember the past are condemned to relive it ...
*Santayana*
...unless they act randomly.

## 1. Introduction

The amortized analysis of on-line algorithms is of interest in a variety of situations [1, 6, 7]. An application of great practical importance is the caching problem [4, 7] – the replacement algorithm for lines in a cache, or pages in a virtual memory. The standard algorithm suggested in the literature is *Least Recently Used* (LRU). The analysis of LRU [7] shows that it achieves optimal worst-case performance (this will be made precise in section 2).

From a practical standpoint, an important requirement of an on-line caching algorithm is that it maintain very little state information (memory) from the past. Such memory is expensive and slow to update in hardware, as pointed out by [8]. Previous theoretical studies have not touched on this issue of the memory resources required by an on-line caching algorithm. For instance, the well-known LRU algorithm has a substantial memory requirement (this will be made precise in section 2). An alternative to large memory is randomization. We refer here to algorithms that make probabilistic choices during execution, with their performance studied under worst-case inputs. On-line randomized algorithms have received scant attention in the past.

It section 2 we show that the worst-case performance that is achieved by LRU is also achieved by a very simple randomized algorithm, namely random replacement. This algorithm is memoryless - it uses no information from the past. However, it uses $\log m$ random bits, where $m$ is the cache size, at each eviction. We show that there is a direct tradeoff between the number of memory bits and the number of random bits used by optimal on-line cache replacement algorithms. We also show that no memoryless algorithm performs better than random replacement.

*IBM Research Division, T.J.Watson Research Center, Yorktown Heights, NY 10598, USA.

In section 3 we extend the random replacement algorithm to give a solution to the *generalized cache problem*, a problem of practical interest for which no provably good algorithm was known before.

On-line algorithms typically use memory to maintain statistics on the cost of past events. This is replaced in randomized, memoryless algorithms by probabilistic processes whose distributions implicitly reflect these statistics. In section 4 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 is used to derive simple memoryless algorithms for two types of on-line problems. We derive a natural algorithm for the *k-server problem* [6] that is memoryless and randomized. We give a bound on the performance of this algorithm for the cases $k = 2$ and $k = n - 1$, and present a tantalizing conjecture which if true would yield a bound on our algorithm's performance for arbitrary $k$. This would be significant, for there is no known provably good algorithm for arbitrary $k$. We also derive memoryless algorithms for *metrical task systems* [1].

# 2. Caching Algorithms

## 2.1. Conventions

We begin our study of caching algorithms using a simple two-level store as the model for caches; the model is essentially that of Sleator and Tarjan, with added provisions for studying randomized algorithms and the memory required by on-line algorithms. There is a *main memory* consisting of a (potentially infinite) number of locations, each of which contains an *item*. The *cache* consists of $m$ locations, each capable of storing one item. The caching algorithm is given a sequence $v_1, v_2, \ldots, v_n$ of *references* to items. A *hit* occurs on the $i$th reference if $v_i$ is one of the items in the cache at the end of reference $i - 1$; otherwise, a *miss* is said to occur. When a miss occurs on the reference to $v_i$, an item is evicted from one of the cache locations, and item $v_i$ is loaded in its place. The cache is initially "empty", i.e. contains none of the referenced items.

We assume that the cache has a finite-state control, with a set $S$ of states. Formally, an on-line cache algorithm is defined by two functions:

$$Hit : [1..m] \times S \to S$$

If a hit occurs at location $i$ when in state $s$, the state is updated to $Hit(i, s)$.

$$Miss : S \to [1..m] \times S$$

If a miss occurs in state $s$ then item at location $i$ is evicted (and the missing item is loaded instead); the state is updated to $s'$, where $(i, s') = Miss(s)$. We adopt a natural measure of the state information maintained by the algorithm: we define a quantity we call its *memory* to be $\log_2 |S|$. An algorithm whose memory is 0 is deemed *memoryless*.

In a randomized algorithm, the state transitions may be probabilistic. Thus, $Hit(i, s)$ is a probability distribution on $S$, and $Miss(s)$ is a probability distribution on $[1..m] \times S$.

EXAMPLES:

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

2. 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.

3. FIFO: 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.

4. FWF: (Flush-When-Full, [5]) Whenever a miss occurs an invalid entry is evicted, and the newly loaded entry is marked as valid; if there are no invalid entries then all entries are invalidated. The scheme can be implemented by storing valid entries contiguously, and using a mod $m$ counter ($\log m$ bits of memory) to point to the last valid entry. The first invalid entry is evicted on a miss.

5. RFWF: (Random-Flush-When-Full, [4]) Same as above, except that a random invalid entry is selected for eviction. The algorithm uses $m$ memory bits, and up to $\log m$ random bits per miss.

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

We compare on-line algorithms (where the state at the end of the $i$th reference depends only on the first $i$ references) to off-line algorithms, where the state after the $i$th reference may depend on the entire sequence (including references $v_j$ for $j > i$). In particular, we use as a yardstick the *optimal off-line algorithm*, which produces the minimum number of misses on every sequence of references: whenever a miss occurs, the item in cache whose next reference is furthest into the future is evicted. It is instructive to compare an on-line algorithm $A$ working with a cache containing $M$ locations with an off-line algorithm working with a cache containing $m$ locations, $m \leq M$, on any reference sequence. We denote by $C_m(v_1, \ldots, v_n)$ the number of misses of the optimal algorithm.

Following [5], we define a deterministic algorithm $A$ to be *c(M,m)-competitive* if for any infinite sequence of references $v_1, v_2 \ldots$

$$\limsup_{n \to \infty} C_M^A(v_1, \ldots, v_n) - c(M, m) \cdot C_m(v_1, \ldots, v_n) < \infty \tag{1}$$

It is easy to show that $A$ is $c(M, m)$−competitive if and only if

$$C_M^A(v_1, \ldots, v_i) \leq c(M, m) \cdot C_m(v_1, \ldots, v_i) \tag{2}$$

for any finite sequence $v_1, \ldots, v_i$ of references (starting with an empty cache). We define the *competitiveness coefficient* of an algorithm $A$ to be the least upper bound on $c(M, m)$ such that (1) or (2) holds.

For a randomized algorithm one can give two different definitions of competitiveness. The first one corresponds to a situation where an adversary chooses a "worst-possible" sequence of references $v_1, v_2, \ldots$ for algorithm $A$; however, this sequence is chosen *a priori*, and does not depend on the actual random choices made during the execution of $A$. We think of these sequences as being generated by a "weak adversary". In other words, we assume that the reference sequence is not affected by the decisions of the caching algorithm.

A randomized algorithm $A$ is *weakly c(M,m)-competitive* if for any sequence of references $v_1, v_2, \ldots$ that has been fixed a priori,

$$\limsup_{n\to\infty} C_M^A(v_1, \ldots, v_n) - c(M,m) \cdot C_m(v_1, \ldots, v_n) < \infty \text{ a.s.}$$

(almost surely, i.e. with probability one). The weak competitiveness coefficient is defined accordingly.

The second definition corresponds to the situation where the adversary can choose each reference $v_i$ depending on any random choices made by the algorithm in serving the first $i-1$ references. This corresponds to a situation where the pattern of (future) memory references may be affected by the (past) behavior of the caching algorithm. (If the caching algorithm is implemented in software, then this is true of the memory references of the caching algorithm itself). Algorithm $A$ is *strongly c(M,m)-competitive* if for any sequence of references generated by such an adversary (henceforth the "strong adversary"),

$$\limsup_{n\to\infty} C_M^A(v_1, \ldots, v_n) - c(M,m) \cdot C_m(v_1, \ldots, v_n) < \infty \text{ a.s.}$$

Thus strong $c$−competitiveness implies weak $c$−competitiveness; the strong competitiveness coefficient is defined accordingly.

It is known [5, 7] that LRU, FIFO and FWF are $M/(M-m+1)$−competitive, and that no deterministic algorithm has a lower competitiveness coefficient. We show later that this lower bound holds for the strong competitiveness coefficient of randomized algorithms.

## 2.2. The Performance of RANDOM

**Lemma 1:** Let $X_1, X_2, \ldots$ be a sequence of random variables such that

$$\mathrm{E}[X_i \mid X_{i-1}, \ldots, X_1] \leq \beta < 0, \text{ a.s. } \forall i$$

and $\sigma^2 X_i \leq \gamma < \infty, \forall i$. Then

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

**Theorem 2:** The strong competitiveness coefficient of RANDOM is $\leq M/(M-m+1)$.

Theorem 2 asserts that RANDOM does as well as any deterministic algorithm, even in the face of a malicious adversary that adapts the sequence of references to the random choices of the algorithm.

**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 on a line that has a negative drift. Let $S_i^R$ be the set of items that RANDOM has in the cache (of size $M$) after the $i$th reference; let $S_i^O$ be the set of items kept in the cache (of size $m$) by the optimal algorithm after the $i$th reference. Let $t_i^R$ be an indicator variable that is 1 if RANDOM misses at reference $i$, and 0 otherwise; let $t_i^O$ be similarly defined for the optimal algorithm. Let

$$\Phi_i = |S_i^R \bigcap S_i^O|$$

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

$$X_i = t_i^R - c \cdot t_i^O - c \cdot \Delta\Phi_i.$$

Then

$$\sum_{i=1}^{n} X_i = C_M^R(v_1, \ldots, v_n) - c \cdot C_m(v_1, \ldots, v_n) - c \cdot \Phi_n + c \cdot \Phi_0.$$

But $\Phi_n \leq m$. Thus, RANDOM is $c$–competitive if $\limsup \sum_{i=1}^{n} X_i < \infty$ almost surely; we complete the proof by showing that this is indeed the case. We have four cases to consider:

1. $v_i \in S_{i-1}^R, v_i \in S_{i-1}^O$ (no misses).
Then $t_I^R = t_i^O = \Delta\Phi_i = 0$, and $X_i = 0$.

2. $v_i \in S_{i-1}^R, v_i \notin S_{i-1}^O$ (miss of optimal).
Then $t_i^R = 0, t_i^O = 1$, and $\Delta\Phi_i \geq 0$. Thus $X_i \leq -c$.

3. $v_i \notin S_{i-1}^R, v_i \in S_{i-1}^O$ (miss of RANDOM).
Then $t_i^R = 1$ and $t_i^O = 0$. RANDOM evicts an item in $S_{i-1}^R \bigcap S_{i-1}^O$ with probability $\Phi_{i-1}/M$, resulting in $\Delta\Phi_i = 0$; otherwise, it evicts an item not in $S_{i-1}^O$, so that $\Delta\Phi_i = 1$. Thus

$$\mathrm{E}[X_i \mid S_{i-1}^R, S_{i-1}^O, v_i] = 1 - c(M - \Phi_{i-1})/M.$$

Since $S_{i-1}^O \not\subset S_{i-1}^R$, $\Phi_{i-1} = |S_{i-1}^O \bigcap S_{i-1}^R| \leq m-1$, and so

$$\mathrm{E}[X_i \mid S_{i-1}^R, S_{i-1}^O, v_i] \leq 1 - c(M - m + 1)/M < 0.$$

4. $v_i \notin S_{i-1}^R, v_i \notin S_{i-1}^O$ (miss of both).
Here $t_i^R = t_i^O = 1$. With probability $(\Phi_{i-1} - 1)/M$, RANDOM evicts one of the $\Phi_{i-1} - 1$ items in $S_{i-1}^R \bigcap S_{i-1}^O$ that is not evicted by the optimal algorithm, in which case $\Delta\Phi_i = 1$; otherwise, $\Delta\Phi_i = 0$. It is easy to verify that

$$\mathrm{E}[X_i \mid S_{i-1}^R, S_{i-1}^O, v_i] \leq 1 - c(M - m + 1)/M < 0.$$

It is also easy to verify that

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

so that the variance of the r.v. $X_i$ is uniformly bounded. Thus, the sequence of random variables $X_1, X_2, \ldots$ fulfills the conditions of Lemma 1 (ignoring the $X_i$ of case 1), and the theorem follows. □

We now show that the bound of Theorem 2 cannot be improved, even under a weak adversary.

**Lemma 3:** Let $W$ be the waiting time for success in a sequence of Bernoulli trials, with probability of success $p$. Let $W_k$ be the truncated variable defined by

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

Then

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

**Theorem 4:** The weak competitiveness coefficient of RANDOM is at least $M/(M - m + 1)$.

**Proof:** Consider a sequence of references of the form

$$a_1 a_2 \ldots a_m \, (b_1 a_2 \ldots a_m)^2 \, (b_2 a_2 \ldots a_m)^3 \ldots$$

where the $a_i$ and the $b_i$ are all distinct items; here $(s)^k$ denotes $k$ repetitions of the sequence $s$. The optimal algorithm misses once on each segment $(b_i a_2 \ldots a_m)^k$. At the beginning of any such segment, 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 $b_i a_2 \ldots a_m$ 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, the expected number of near-misses is at least $(M/(M - m + 1))(1 - ((m-1)/M)^k)$. The claim follows. □

The last theorem can be strengthened to hold even if there are only $M + 1$ distinct items. Also, no memoryless algorithm achieves a better weak competitiveness coefficient; intuitively, when there is no information on which to base the choice of the evicted entry, random, equiprobable choice of an entry to evict is best. We formally prove the claim below, for the particular case $M = m$.

**Theorem 5:** Any memoryless on-line caching algorithm has weak competitiveness coefficient $\geq m$, when $m = M$.

**Proof outline:** A memoryless algorithm is a probability distribution $\{p_1, p_2 \cdots p_m\}$ on the cache locations, where $p_i$ is the probability of evicting the item in location $i$. Consider a randomly chosen sequence of references, consisting of a sequence of rounds. The $k$th round is of the form $(a_1, \ldots, a_m)^k$; the set of $m$ items occurring at round $k$ is obtained by choosing an item from round $k-1$ uniformly at random and replacing it by a new item.

During each round, the adversary has one miss. Let $a_1, \ldots, a_m$ be the items accessed at round $k-1$, and let $a_1, \ldots, a_{i-1}, \hat{a}_i, a_{i+1}, \ldots, a_m$ be the items accessed at round $k$. With probability going to one as $k \to \infty$, the on-line algorithm starts round $k$ with $a_1, \ldots, a_m$ in cache. Assume, w.l.o.g., that item $a_i$ occupies location $i$ in cache. Given this, the algorithm misses during round $k$ until it evicts the item in location $i$. Therefore, the expected number of misses is $\geq \frac{1}{p_i}(1-(1-p_i)^k)$. The expected number of misses for $i$ chosen uniformly at random (given that $a_1, \ldots, a_m$ are in cache at the start of the round), is $\geq 1/m \sum_{i=1}^{m} \frac{1}{p_i}(1-(1-p_i)^k)$. This is minimized at $m(1-(\frac{m-1}{m})^k)$ (when all the $p_i$s are equal). Thus, the ratio between expected on-line cost and off-line cost for such randomly chosen sequence of references has limit $\geq m$. □

The result of the last theorem does not hold true for algorithms with memory: Fiat *et al.*[4] have shown that RFWF has a weak competitiveness coefficient which is $O(\log m)$, when $m = M$.

## 2.3. Trading Memory for Randomness

Theorem 2 demonstrates that the use of randomness obviates the need for any memory, while performing as competitively as the FIFO algorithm. Recall that such a reduction/removal of state information was our original motivation stemming from [8]. We now study algorithms such as those suggested by [8], and show that they form a family of schemes that are as competitive as FIFO and trade memory for randomness in a quantifiable manner.

For the remainder of this section, we will focus on the case $m = M = 2^k$ for some $k$. Then, FIFO uses $k$ bits of memory and no randomness, whereas RANDOM uses no state information and $k$ bits of randomness per reference. Let $S_A(m)$ be the number of bits of memory maintained by an algorithm $A$, and $R_A(m)$ be the number of bits of randomness used by $A$ per reference (we regard $S_A(m)$ and $R_A(m)$ as invariants of $A$ that do not change with $i$). Clearly, for any algorithm $A$, $S(m) + R(m) \geq k$ (else $A$ will not be able to address all its $m$ cache locations, and the reference string could continually request items in $S^O - S^A$).

**Theorem 6:** Let $i$ be an integer, $0 \leq i \leq k$. There is an $m$–competitive on-line algorithm $A^i$ such that $S_{A^i}(m) = i$ and $R_{A^i}(m) = k - i$.

The family of algorithms $A^i$ bridges the gap between FIFO and RANDOM, trading randomness for space, and remaining $m$–competitive.

**Proof:** Algorithm $A^i$ is defined as follows. Let $I = 2^i$ and $J = m/I$. Conceptually, the cache locations of $A^i$ are organized as an $I \times J$ matrix. We use an $i$–bit counter that points to a row of this matrix. On a miss, we do the following: we randomly evict one of the $J$ items in the row pointed to by the counter ($k - i$ bits of randomness suffice for this), and increment the counter mod $I$.

To prove that $A^i$ is $m$–competitive for all $i$, we use essentially the same argument as in Theorem 2 (using the same potential function and random walk). The only difference

is that we look at the change in the position $X$ of the random walk over a sequence of $2^i$ misses instead of a single step at a time. □

Using an argument similar to that in Theorem 4, it can be shown that the weak competitiveness coefficient of $A^i$ is at least $m$ for all $0 \leq i \leq k$. The algorithms $A^i$ use the minimum possible total of space and randomness, in the spirit of [8].

# 3. The Generalized 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 *generalized cache problem,* an item $x$ has a positive real *weight* $w(x)$ associated with it. The significance of the weight is the following: in loading item $x$ into the cache, an algorithm incurs a cost $w(x)$. 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 $v_1, v_2, \ldots, v_n$ by $C_M^A(v_1, v_2, \ldots, v_n)$, and the cost of the OPTIMAL algorithm working with a cache containing $m$ locations on this sequence by $C_m(v_1, v_2, \ldots, v_n)$. The competitiveness coefficient of an algorithm $A$ is defined accordingly. Thus, the previous section dealt with the special unit cost case, where $w(x) = 1\,, \forall x$.

The generalized cache problem has applications to caching fonts in printers. The number of fonts that can be cached at a time in the printer is subject to a maximum, but fonts stored in larger files take longer to bring into the printer's cache. There are two noteworthy aspects of the generalized cache problem that distinguish it from the simple cache problem considered in the previous section: (1) it is non-trivial to find the optimal off-line schedule; (2) we know of no deterministic algorithm for the problem that is $M/(M-m+1)$-competitive.

We now present a randomized on-line algorithm for this problem, which we call the HARMONIC algorithm. The algorithm behavior depends only on the weights of the items in cache. Let $x_1, \ldots, x_m$ be the items in cache when a miss occurs. The HARMONIC algorithm uses the following simple, probabilistic eviction rule: evict $x_i$ with probability $p_i$ where

$$p_i = \frac{1/w(x_i)}{\sum_{j=1}^{m} 1/w(x_j)}.$$

**Theorem 7:** The strong competitiveness coefficient of the HARMONIC algorithm is $\leq M/(M-m+1)$.

**Proof:** As in the proof of Theorem 2, we use a potential function to create a random walk on the real line that has a negative drift. Let $S_i^H$ be the set of items kept in the cache by HARMONIC after the $i$th reference, and $S_i^O$ be the set of items kept by the optimal algorithm. Let

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

and $\Delta\Phi_i = \Phi_i - \Phi_{i-1}$. Letting $t_i^H$ denote the cost incurred by the HARMONIC algorithm in servicing the $i$th reference and $t_i^O$ the corresponding cost of the OPTIMAL algorithm, we define

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

where $\alpha > \beta > 1$. We now proceed on the lines of the proof of Theorem 2, breaking the analysis up into two parts instead of considering four cases:
1. The OPTIMAL algorithm evicts an item. We can assume that the OPTIMAL algorithm loads a new item only immediately before a reference to that item. Also, without affecting the analysis, we can charge the OPTIMAL algorithm for the item it evicts rather than for the item it loads; thus $t_i^H = w(x_i)$, if the OPTIMAL algorithm evicts $x_i$ on reference $i$ (and 0 otherwise).
2. The HARMONIC algorithm evicts an item on a miss, and is charged for the weight of the item it loads.

We examine the effect of the two kinds of actions described above on the random walk $\sum_{j=1}^{i} X_j$. In particular, we examine the effect of either action on $\mathrm{E}[X_i \mid S_{i-1}^O, S_{i-1}^H, v_i]$.
1. The OPTIMAL algorithm loads $x$ and evicts $x'$. Then $t_i^O = w(x')$, and $\Delta\Phi_i \leq w(x')M/(M-m+1)$. (The equality is realized when $x' \in S_{i-1}^H \bigcap S_{i-1}^O$ and $x \notin S_{i-1}^H$). Thus the contribution of the OPTIMAL algorithm's action to $\mathrm{E}[X_i | S_{i-1}^O, S_{i-1}^H, v_i]$ is $< 0$.
2. The HARMONIC algorithm misses on a reference to item $x$, so that $t_i^H = w(x)$. Then $|S_{i-1}^H \bigcap S_{i-1}^O| \leq m-1$ and $|S_{i-1}^H - S_{i-1}^O| \geq 1$. Thus

$$\mathrm{E}[\Delta\Phi_i \mid S_{i-1}^O, S_{i-1}^H, v_i] \;=\; w(x) - \frac{|S_{i-1}^H \bigcap S_{i-1}^O|}{\sum_{y\in S_{i-1}^H} 1/w(y)} + \frac{m-1}{M-m+1}\cdot\frac{|S_{i-1}^H - S_{i-1}^O|}{\sum_{y\in S_{i-1}^H} 1/w(y)} \;>\; w(x).$$

Thus, the contribution of the HARMONIC algorithm's action to $\mathrm{E}[X_i \mid S_{i-1}^O, S_{i-1}^H, v_i]$ is also $< 0$.

Applying Lemma 1 to the sequence of random variables $X_i$, we conclude that almost surely

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

so that almost surely

$$\limsup_{n\to\infty} C_M^H(v_1,\ldots,v_n) - \frac{\alpha M}{M-m+1}\cdot C_m(v_1,\ldots,v_n) < \infty$$

and this yields the result. □

The last result is valid even if there are infinitely many distinct weights; it is only required that all weights are in a bounded range $0 < a < weight < b < \infty$. Note that RANDOM is exactly the HARMONIC algorithm restricted to the special case $w(x) = 1, \forall x$. Thus, by Theorem 4, the weak competitiveness coefficient of the HARMONIC algorithm is $\geq M/(M-m+1)$. In fact, this lower bound holds true for any set of at least $M+1$ distinct items, when $M = m$. We require the following lemma from probability theory:

**Lemma 8:** Let $X_0, X_1, \ldots$ be a sequence of positive random variables with uniformly bounded expectations. Let $t$ be a stopping time for the sequence; $t$ ranges over the natural numbers and the event $[t = i]$ depends only on $X_0, \ldots, X_i$. Assume that

$$\Pr(t = i \mid X_0, \ldots, X_{i-1}) = \mathrm{E}(X_i \mid X_0, \ldots, X_{i-1}).$$

Then

$$\mathrm{E}(\sum_{i=0}^{t} X_i) = 1.$$

**Theorem 9:** Let $M = m$. The weak competitiveness coefficient of the HARMONIC algorithm $C(M, M) \geq M$, even when restricted to any set of $M + 1$ items.

**Proof:** Let $x_0, \ldots, x_M$ be any $M + 1$ items. Let $w_i = w(x_i)$. Consider a sequence that consists of successive rounds of references to $x_0, \ldots, x_{M-1}$; assume the HARMONIC algorithm starts with $x_M$ in cache, and $x_0$ out of cache. Let $t$ be the number of misses on the sequence. Let $i(j)$ be the index of the $j$-th item evicted; $i(t) = M$.

Without loss of generality, we are charging the HARMONIC algorithm for the items it evicts. The cost of HARMONIC on this sequence is

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

When the $j$-th eviction occurs the cache contains all items, with the exception of $x_{i(j-1)}$. Therefore, for $j \leq t$,

$$\Pr(i(j) = r \mid i(j-1)) = \frac{1/w_r}{\sum_{k \neq i(j-1)} 1/w_k}$$

if $r \neq i(j-1)$, zero otherwise. Accordingly,

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

and

$$\Pr(t = j \mid i(j-1)) = \frac{1/w_M}{\sum_{k \neq i(j-1)} 1/w_k}$$

Thus,

$$\mathrm{E}[w_i(j) \mid i(j-1)] = M \cdot w_M \cdot \Pr(t = j \mid i(j-1)).$$

It follows, by the previous lemma, that

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

Consider now a sequence of accesses consisting of successive rounds $(S_0)^{n_0}, (S_1)^{n_1}, \ldots, (S_{(i \bmod (M+1))})^{n_i}$, where $S_j = (x_0, \ldots, x_{j-1}, x_{j+1}, \ldots, x_M)$. Let $c < 1$

be a positive constant. Let $A_i$ be the event "$x_{(i \bmod (M+1))}$ is not in cache at the end of the $i$-th round". Let $C_i$ be the cost of HARMONIC for the $i$-th round. By taking the sequence $n_0, n_1, \ldots$ to grow sufficiently fast we can obtain that almost surely $\neg A_i$ occurs only finitely many times, and almost surely the cost of HARMONIC at the $i$-th round is larger than $cMw_{(i \bmod (M+1))}$, with finitely many exceptions. Thus, almost surely

$$\limsup_{n \to \infty} \sum_{i=0}^{n} C_i^H - cM \sum_{i=0}^{n} w_{(i \bmod (M+1))} \geq 0.$$

On the other hand, the cost of OPTIMAL on the $i$-th round is $w_{(i \bmod (M+1))}$.

□

# 4. Random Walks and Probabilistic Counters

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 section 4.1. This abstraction proves useful when we subsequently analyze algorithms for *server systems* (an abstraction due to Manasse *et al.* [6] that captures caching as a special case) and for the *metrical task systems* of Borodin *et al.* [1].

## 4.1. Traversals and Random Walks

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

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

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

The other definitions of section 2.1 are extended in a similar manner. Formally, a traversal algorithm with a set $S$ of states is a function

$$S \times \text{ Current Node} \rightarrow S \times \text{New Node}.$$

As before, we define an algorithm to be *memoryless* if $\log|S|$ is zero. Thus 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. Note that there is no difference between a weak and a strong adversary for randomized memoryless algorithms.

Let $i_0 i_1, \ldots i_{r-1}$ be a sequence of nodes in which each node of $G$ occurs at least once. A deterministic traversal algorithm is defined by visiting the nodes of $G$ in the order defined by the sequence. The state of the algorithm is the index $s$, and if the algorithm is at node $i_s$ in state $s$, then it next moves to node $i_{(s+1) \bmod r}$, and state $(s+1) \bmod r$. We call such algorithm a *cyclic traversal algorithm.* The following result is proven in [1].

**Theorem 10:** For any graph on $n$ nodes there is a cyclic traversal algorithm which is $4(n-1)$-competitive.

Cyclic traversal algorithms are not memoryless because they remember the index $s$. 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 moves to node $j$, when at node $i$. Thus, $\sum_{j \neq i} p(i,j) = 1$. 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 of a random walk that starts at node $i$ and ends when node $j$ is first reached. Define the *expansion* of the random walk to be $\max_{i,j} H(i,j)/d(i,j)$.

**Lemma 11:** A probabilistic traversal algorithm based on a random walk with expansion $c$ is $c$-competitive.

The HARMONIC random walk has transition probabilities

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

The probability of using a particular outgoing edge from a node is inversely proportional to its cost. This process has been studied in [2], where the following result is proved about a HARMONIC walk on any graph with $m$ edges.

**Theorem 12:** The HARMONIC random walk has expansion $\leq 2m$.

The last result is tight; equality is achieved for certain graphs. When the distance matrix $(d(i,j))$ is metric, $m = n(n-1)/2$. The HARMONIC random walk does not yield, in general, the least possible expansion. We conjecture the following holds true.

**Conjecture 13:** For any $n$-node graph there is a random walk with expansion $O(n)$.

We can prove this conjecture for a few special cases – in particular, for the case where where the distance matrix is Euclidean: $x_1, \ldots, x_n$ are $n$ points in $R^d$ for fixed $d$, and $d(x_i, x_j)$ is the Euclidean distance between $x_i$ and $x_j$. This follows from the fact that for any set of points in $R^d$, there exists a linear-sized graph $G$ spanning them such that the shortest path in $G$ between $x_i$ and $x_j$ is $\leq c_d \cdot d(x_i, x_j)$ $\forall i, j$ for a constant $c_d$ depending on the dimension $d$ of the space (Chew [3] provides an introduction to such graphs). A HARMONIC walk restricted to this sparse graph $G$ will, by Theorem 12, satisfy Conjecture 13.

## 4.2. Server Systems

The server problem is a generalization [6] of the caching problem. The problem is specified by a complete graph on $n$ nodes $1, \ldots, n$, and a metrical distance matrix $(d(i, j))$. There are $M$ servers that occupy $M$ of the nodes of the graph. A request specifies a node; in response, a server must be moved to 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 after each request, not knowing about future requests.

The simple 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 generalized 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 obtains an equivalent problem by using the distance matrix $d(i, j) = (w_i + w_j)/2$; this matrix represents a generalized cache problem where the caching algorithm is charged 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

$$\text{Algorithm} : [1..n]^M \times S \times [1..n] \rightarrow [1..n]^M \times S$$

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 $\log_2 |S|$. All the definitions of Section 2.1 carry through. The definition of memory given here is different from the definition used in Section 2: here we allow the transition to depend on the server locations, in addition to the algorithm state; in Section 2 we did not allow the transition to depend on the cache content. With this more lenient definition, it is possible to construct a memoryless deterministic caching algorithm such that $c(M, M) = M$ (Marek Chrobak, personal communication), and there is no trade-off between memory and randomness.

In [6] it is shown that the competitiveness coefficient of any deterministic on-line 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 proof extends to randomized algorithms, against strong adversaries.

**Theorem 14:** The strong competitiveness coefficient of any randomized $M$ server algorithm $C(M,m) \geq M/(M-m+1)$, even for a system with $M+1$ nodes.

**Proof:** The proof uses an adversary similar to that in [6] and is omitted here. □

The last result implies that $M/(M-m+1)$ is the best possible strong competitiveness coefficient for the cache problem and for the generalized cache problem. For the rest of this section, we deal with the case $M = m$.

### 4.2.1. Random Walk Algorithms for the Server Problem

We now present a simple and natural memoryless algorithm for the server problem. Let $(p(i,j))$ be a matrix of transition probabilities defined on the graph. Suppose we have a request at a node $r$, and we currently have no server at $r$. Let $i_1, \cdots i_M$ be the current positions of our servers. We choose one of our servers at random to service the request at $r$, according to the probability distribution induced by $(p(i,j))$: Server $i_j$, $1 \leq j \leq M$ is chosen with probability

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

**Theorem 15:** Let $M = n-1$. Assume that the random walk defined by the transition probabilities $p(i,j)$ is $c$-competitive. Then the strong competitiveness coefficient of the corresponding server algorithm $C(M,M) \leq c$.

**Proof:** When $M = n-1$, there is exactly one node of the graph that contains none of our servers. We call this node $r(t)$ at time $t$. We incur a cost only when the request at time $t$ falls at $r(t)$. Likewise, for the adversary generating the requests, we can assume that there is exactly one node not occupied by any of his servers; we denote this by $a(t)$. If $r(t) = a(t)$, the adversary must make a move and incur a cost in order to make us incur any further cost.

Assume that initially, at $t = 0$, $r(0) = a(0)$. We consider the behavior of our algorithm in a sequence of *phases*. A new phase begins when the current phase ends. A phase ends when $r(t) = a(t)$ or when the adversary makes a move. We can assume that a phase ends only when $r(t) = a(t)$ (i.e., the adversary does not make a move unless he has no other choice). Then, a strong adversary begins a phase by moving one of his servers at $t = t_0$. At this point $r(t_0)$ is at distance $D = d(r(t_0), a(t_0))$ from $a(t_0)$. The adversary incurred a cost of $D$. On every subsequent request until the end of the phase, the adversary presents us with a request at $r(t)$.

It is easy to see that $r(t)$ executes a random walk on the graph, choosing at each step an edge $(i,j)$ with probability $p(i,j)$. The walk terminates when it reaches $a(t_0)$. The expected length of the walk from $r(t_0)$ to $a(t_0)$ is the expected cost the on-line algorithm incurs in this phase. Summing over all phases yields the result. □

**Corollary 16:** The strong competitiveness coefficient of the $M = n-1$ server algorithm induced by the HARMONIC random walk is $C(M,M) \leq M(M+1)$.

Manasse *et al.* [6] give an algorithm for the $(n-1)$-server problem that is $(n-1)$-competitive, and thus superior in its amortized performance. However, it requires substantial computational resources (time and space), whereas our algorithm is simple and natural. Furthermore, when Conjecture 13 holds for the random walk derived from $(p(i,j))$, the competitiveness of the resulting memoryless randomized algorithm is within a multiplicative constant of that of the algorithm of Manasse *et al.*

The proof of the last theorem suggests 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 our servers. Thus the adversary first makes a move in order to be able to give us a request for which we incur a cost.

**Conjecture 17:** (Lazy Adversary Conjecture): The following (strong) 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 we have none, the adversary presents the next request at that point (instead of making a move and incurring a cost).

The conjecture suggests that the ratio of our expected cost to that of the adversary is maximized under this adversary policy. (The conjecture is *not* true for every algorithm; we only suggest it is true for a class of algorithms that includes memoryless algorithms). If this conjecture were proved, we can reduce the analysis of the algorithm to a phase analysis and random walk similar to that in Theorem 15 (details omitted here). The result would be an upper bound of $c$ on the strong competitiveness coefficient of our algorithm, where $c$ is the expansion factor for a random walk on a graph with $M+1$ nodes; $c = M(M+1)$ for the HARMONIC random walk. This would be significant because for $M > 2$, there is no known algorithm that is $c$-competitive where $c$ is bounded by *any* function of $M$ alone, even on finite graphs (other than $M = n-1$).

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 18:** The strong competitiveness coefficient of the HARMONIC algorithm for the 2-server problem is in the interval $[3, 6]$.

The proof is omitted in this version. Manasse *et al.* [6] 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.

## 4.3. Metrical Task Systems

A metrical task system consists of a a graph $G$ with $n$ nodes $\{1, \ldots, n\}$ and a metrical cost matrix $(d(i,j))$. An algorithm resides at one node of $G$ at any given time. A task $T$

is a vector of length $n$ whose $i$th component is the cost of processing $T$ while at node $i$; we assume that these costs are uniformly bounded. Given a task sequence $T_1, T_2 \cdots T_k$, 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. We refer the reader to Borodin *et al.*[1] for details; 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.)

An algorithm is now defined by a function:

$$\text{Algorithm} \ : \ [1..n] \times S \times \text{Task} \rightarrow [1..n] \times S.$$

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

Borodin *et al.* give a general 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. A metrical task system algorithm $\hat{A}$ is derived from $A$, as follows: Let $i$ be the current location of the algorithm, and let $j$ be the next node visited by the traversal algorithm. Then $\hat{A}$ moves to $j$ when the processing cost since entering $i$ reaches the move 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.* is to view the process as occurring in continuous time - details may be found in their paper and are omitted here). The cost incurred by algorithm $\hat{A}$ is thus twice the cost of its moves. Using an argument similar to that used in the proof of Theorem 9, one can obtain the following result.

**Theorem 19:** Assume that the traversal algorithm $A$ is $c$-competitive. Then the derived metrical task system algorithm $\hat{A}$ is $2c$-competitive.

Using a deterministic traversal algorithm with competitiveness coefficient $4(n-1)$ (Theorem 10), Borodin *et al.* obtain a deterministic on-line algorithm for any metrical task system which is $8(n-1)$-competitive.

Using the HARMONIC random walk traversal, we obtain a probabilistic on-line algorithm which is $2n(n-1)$ competitive. When conjecture 13 holds true, we obtain an algorithm wich is $O(n)$-competitive. Such random walk algorithms are not memoryless, as one has to maintain a counter that accumulates the total processing cost at the current node. However, one can replace this counter by a probabilistic counter.

Assume that the algorithm is at node $i$, and let $\bar{d}$ be the cost of the next move; if the algorithm is probabilistic, we take $\bar{d}$ to be the expected cost of the next move (i.e., for a random walk algorithm, $\bar{d} = \sum_{j \neq i} p(i,j)d(i,j)$). Given a task $T$, let $T(i)$ be the cost of processing $T$ at node $i$. Assume $T(i) \leq d$ (to justify this, we use the same continuous-time ideas of Borodin *et al.* without further elaboration in this version). Independent of

previous tasks and processing costs, our algorithm *Gambler* does the following: given $T$, flip a coin with $\Pr[Heads] = T(i)/\bar{d}$; if it comes up Heads, *Gambler* moves to the next node in the traversal (this next node may be chosen probabilistically), else it remains at the current node. Note that *Gambler* is memoryless. Lemma 8 can now be invoked to show that the expected processing cost incurred by *Gambler* algorithm at node $i$ is $\bar{d}$. This yields the following theorem, whose proof we omit here.

**Theorem 20:** Let $c$ be the competitiveness coefficient of a traversal algorithm $A$. The competitiveness coefficient of *Gambler* using the traversal $A$ is $\leq 2c$.

Using this construction, together with the HARMONIC random walk, we get a memoryless on-line algorithm for metrical task systems which is $2n(n-1)$-competitive. When conjecture 13 is correct, we get an $O(n)$-competitive memoryless algorithm.

# Acknowledgements

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

# References

[1] A. Borodin, N. Linial, and M. Saks. An optimal online algorithm for metrical task systems. In *Nineteenth Annual ACM Symposium on Theory of Computing*, pages 373–382, 1987.

[2] 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. In *Proceedings of the 21st Annual ACM Symposium on Theory of Computing*, Seattle, May 1989.

[3] P. Chew. There exist planar graphs almost as good as the complete graph. 1988. To appear, JCSS.

[4] A. Fiat, R. Karp, M. Luby, L. McGeoch, D. Sleator, and N. Young. Randomized algorithms for paging problems. 1988. In preparation.

[5] A. R. Karlin, M. S. Manasse, L. Rudolph, and D.D. Sleator. Competitive snoopy caching. *Algorithmica*, 3(1):70–119, 1988.

[6] M.S. Manasse, L.A McGeooch, and D.D. Sleator. Competitive algorithms for on-line problems. In *Twentieth ACM Annual Symposium on Theory of Computing*, pages 322–333, 1988.

[7] D.D. Sleator and R.E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28:202–208, February 1985.

[8] K. So and R.N. Rechtschaffen. Cache operations by MRU change. *IEEE Trans. Computers*, 37(6):700–709, June 1988.