Static Optimality and Dynamic Search-Optimality in Lists and Trees

Avrim Blum* Shuchi Chawla† Adam Kalai‡

Abstract

Adaptive data structures form a central topic of online algorithms research, beginning with the results of Sleator and Tarjan showing that splay trees achieve static optimality for search trees, and that Move-to-Front is constant competitive for the list update problem [ST85a, ST85b]. This paper is inspired by the observation that one can in fact achieve a $1+\epsilon$ ratio against the best *static* object in hindsight for a wide range of data structure problems via "weighted experts" techniques from Machine Learning, if computational decision-making costs are not considered.

In this paper, we give two results. First, we show that for the case of *lists*, we can achieve a $1+\epsilon$ ratio with respect to the best static list in hindsight, by a simple *efficient* algorithm. This algorithm can then be combined with existing results to simultaneously achieve good static and dynamic bounds. Second, for trees, we show a (computationally *in*efficient) algorithm that achieves what we call "dynamic search optimality": dynamic optimality if we allow the *online* algorithm to make free rotations after each request. We hope this to be a step towards solving the longstanding open problem of achieving true dynamic optimality for trees.

1 Introduction

Adaptive data structures form one of the central topics of online algorithms research, beginning with the results of Sleator and Tarjan that splay trees perform within a constant factor of the best *static* search tree for any request sequence, and that Move-to-Front is constant competitive with respect to the best off-line (adaptive) algorithm for lists [ST85a, ST85b]. Online algorithms for lists have subsequently been well-studied, with upper and lower bounds of 1.6 and 1.5 respectively, on the competitive ratio of randomized algorithms for this problem [AvW95, Tei93]. The case of trees appears much harder: it is not known whether splay trees or any other online algorithm performs within a constant factor of the optimal off-line adaptive tree algorithm.

An interesting point is that in terms of competing against the best *static* object in hindsight, one can in principle achieve a ratio of $1+\epsilon$ for a wide variety of data structure problems, if we ignore time spent computing which update to perform. In particular, Blum and Burch [BB00] show the following general result.

THEOREM 1.1. (THEOREM 4 OF [BB00]) *Given N on-line algorithms for a Metrical Task System Problem of diameter D, and given $\epsilon > 0$, one can use the Randomized Weighted Majority algorithm to achieve on any request sequence σ an expected cost at most $(1+\epsilon)L_\sigma + O(\frac{1}{\epsilon}D\log N)$, where L_σ is the cost of the best of the N algorithms on σ in hindsight.*[1]

For example, the list-update problem is a Metrical Task System problem with one state for each ordering of the n elements. There are $N = n!$ different lists of n elements, and the diameter of the space (the maximum cost to move between one list and another) is $O(n^2)$. Therefore, viewing each static list as an online algorithm, applying Randomized Weighted Majority achieves a $(1+\epsilon)$ static ratio with additive constant $O(\frac{1}{\epsilon}n^3\log n)$. For trees, there are "only" $2^{O(n)}$ different search trees on n elements, and the diameter of the space is $O(n)$. So, this algorithm achieves a $1+\epsilon$ static ratio with additive constant $O(\frac{1}{\epsilon}n^2)$. We call this type of bound (namely, a $1+\epsilon$ ratio with additive constant polynomial in n and $1/\epsilon$) *strong static optimality*. Unfortunately, the algorithm of Theorem 1.1 is horribly computationally inefficient, because it must explicitly maintain a probability distribution over all the component algorithms. Some progress on efficient MCMC simulations for these type of algorithms has been made [CLS01, KV00] but not for the situations of interest to us.

*email: avrim@cs.cmu.edu, Computer Science Department, Carnegie Mellon University, Pittsburgh, PA15213

†email: shuchi@cs.cmu.edu, Computer Science Department, Carnegie Mellon University, Pittsburgh, PA15213

‡email: akalai@cs.cmu.edu, Computer Science Department, Carnegie Mellon University, Pittsburgh, PA15213

[1] The Randomized Weighted Majority algorithm [LW94] maintains a probability distribution over the N component algorithms ("experts"). At each time step, the experts are penalized based on their cost at that step, multiplying the weight on an expert incurring cost c by $(1-\epsilon')^c$, and then renormalizing, where we will use $\epsilon' = O(\epsilon/D)$. The name of the algorithm comes from a setting in which each expert is making a prediction, in which case this procedure can be viewed as taking a weighted vote and then choosing a prediction probabilistically based on the vote totals. It is also called the Hedge algorithm [FS96].

In many situations the best static algorithm may be much worse than an optimal dynamic algorithm. In the case of list update, for instance, the best static algorithm would be $O(n)$ times worse than the best dynamic algorithm if the adversary repeatedly accesses every element in succession. One might ask then, why should we care for static optimality. The reason is that when accesses come from a fixed distribution, which is the case with many real world applications, static algorithms perform the best.

In this paper, we present two results. First, for the case of lists, we show how to achieve a $1+\epsilon$ ratio with respect to the best static list in hindsight by a simple *efficient* algorithm. The algorithm is randomized: the standard lower bound of 2 for deterministic algorithms holds for static ratio as well. We combine this algorithm with the currently best known dynamic algorithm (COMB, see [BEY98] for details) to obtain one which is dynamically optimal and at the same time has strong static optimality. Second, for trees, we show a (computationally *in*efficient) algorithm motivated by online learning algorithms that achieves what we call "dynamic search optimality": dynamic optimality if we allow the *online* algorithm to make free rotations after each request. We hope this to be a step towards addressing the longstanding open problem of achieving true dynamic optimality for trees. At the very least, this shows that dynamic optimality cannot be proven *impossible* by any argument that neglects the rotation cost of the online algorithm.[2]

2 Strong Static Optimality for List Update

The List Update problem is to maintain a list of n elements while serving requests of the form ACCESS(x), where x is an element in the list. The cost of an access for element x is the depth of x in the list i.e. the i-th element has a cost of $i-1$. This is known as the *Partial Cost* model (as opposed to the *Full Cost* model in which the cost of access is i for the i-th element). Following an access, the element x may be moved anywhere higher in the list without additional costs. Then other elements may be moved at a cost of one per transposition.

In this section, we describe two different algorithms for list update, both based on the same idea of reducing the problem to a 2-expert "combining expert advice" problem, and both achieving a $1+\epsilon$ competitive ratio with respect to the best static list. We prove the bound for the first algorithm by reducing to the algorithm of [LW94], and prove the bound for the second algorithm from first principles. We refer to the optimal static list as OPT.

2.1 Algorithm A

Let L be a list of n elements and σ be a sequence of accesses presented to the algorithm. Algorithm A proceeds as follows:

1. Pick n numbers $r_1, \cdots, r_n$ from $[0,1]$ uniformly at random such that $\sum_{i=1}^{n} r_i = 1$. (This is easily done by throwing $n-1$ darts into $[0,1]$ and letting r_i be the length of the ith interval.) Assign r_i to the i-th element in L.
   Let $\beta < 1$ be a performance parameter for the algorithm.
2. Assign a weight w_i to each element in the list, with initial value 1. Order elements in the list in increasing order of $r_i \times w_i$.
3. When an element x at position i is accessed, update $w_i = \beta \times w_i$.
4. Move element x up in the list, if necessary, to maintain ordering of elements by weight.
5. Go to step 3 till last request is served.

THEOREM 2.1. *Algorithm A with $\beta = e^{-\epsilon}$ is ($1+\epsilon$)-competitive with the best static list.*

In the following analysis, we use a_{xy} to denote the sublist of a list a that contains all occurrences of only the elements x and y.

LEMMA 2.1. **Pairwise Property Lemma [BEY98]** *An algorithm ALG satisfies the pairwise property if and only if for every request sequence σ and every pair $\{x, y\} \in L$, the probability that x precedes y in L when ALG serves σ, is the same as the probability that x precedes y in L_{xy} when ALG serves σ_{xy}.*
Let ALG be an algorithm that satisfies the pairwise property. Suppose that for every pair $\{x, y\} \subseteq L$, and for every request sequence σ, $ALG(\sigma_{xy}) \leq c \cdot OPT(\sigma_{xy})$ in the Partial Cost model, then ALG is strictly c competitive in both the Partial Cost and Full Cost models.

LEMMA 2.2. *Algorithm A satisfies the Pairwise property.*

Proof. Consider two elements x and y in L. The relative order of x and y in L depends only on which of $w_x r_x$ and $w_y r_y$ is greater. Let $R = 1 - \sum_{i \neq x,y} r_i$. Then, conditioned on R, r_x and r_y are random in $[0, R]$ subject to having a sum of R. So, $Pr[x$ precedes y in $L] =$

[2] In contrast, the standard $\Omega(\log k)$ lower bound for randomized paging, based on considering a sequence in which pages in $\{1, \ldots, k+1\}$ are requested completely at random, holds even if the online algorithm is allowed to move items into and out of its cache between requests for free.

$Pr[w_x r_x < w_y r_y] = P[w_x r_x < w_y(R - r_x)] = P[r_x < \frac{Rw_y}{w_x+w_y}] = \frac{w_y}{w_x+w_y}$ as in this case, $r_x \in_R [0, R]$.

Now let σ_{xy} be a subsequence of σ which contains all occurrences of x and y and no others. Let L_{xy} be a list containing only x and y but in the same initial order as in L. If we run algorithm A on L_{xy} with sequence of accesses σ_{xy}, the weights of x and y will be the same as in the original case. However, the random numbers associated with the two elements are now different and satisfy the property $r'_x + r'_y = 1$.

Now, $Pr[x$ precedes y in $L_{xy}] = Pr[w_x r'_x < w_y r'_y] = P[w_x r'_x < w_y(1 - r'_x)] = P[r'_x < \frac{w_y}{w_x+w_y}] = \frac{w_y}{w_x+w_y}$. This is the same as the probability obtained before.

Therefore, the order of the two elements in L_{xy} would reflect their relative order in the list L and the lemma follows.

Lemma 2.3. *For a list of two elements, algorithm A has a performance ratio of* $\frac{\log\frac{1}{\beta}}{1-\beta}$ *compared to the best static list.*

Proof. Consider a list L of two elements x and y. If a request accesses the element which is second in the list at the time of access, the algorithm incurs a cost of 1; otherwise, it incurs no cost. There are only two static lists on two elements – (x, y) and (y, x). We consider these as experts. When element y is accessed, the list (x, y) incurs a cost of 1 (or makes a "mistake"), while (y, x) incurs no cost (so it is "correct" on this request). The reverse happens when element x is accessed.

Recall that, $r_x + r_y = 1$. Algorithm A will move x to the front at any point of time when $w_x r_x < w_y r_y$. This happens with probability $P[w_x r_x < w_y r_y] = \frac{w_y}{w_x+w_y}$ as calculated in proof of Lemma 2.2.

Notice that if we associate weight w_y with the list (x, y) and w_x with the list (y, x), then the weight of each expert is $\beta^{(\#\text{ of mistakes made by that expert})}$. Algorithm A uses one of the two experts with probability in proportion to their weights. This is the same as the Randomized Weighted Majority algorithm of [LW94] for the case of 2 experts.

Therefore, following [LW94], we get, $E[A(\sigma_{xy})] \leq \frac{OPT(\sigma_{xy}) \log\frac{1}{\beta} + \log 2}{1-\beta}$.

Proof of Thm 2.1 First, notice that algorithm A does not incur any movement costs other than ordering the list initially after choosing the weights. This is because the relative order of elements changes only when one of them is accessed, and at that time, the list can be reordered by putting just the accessed element in the right place. The initial movement cost is at most $n(n-1)/2$.

Using the *List Factoring lemma* [BEY98] with Lemma 2.2 and Lemma 2.3, we obtain the competitive ratio of algorithm A with respect to the cost incurred by OPT as $E[A(\sigma)] \leq \frac{OPT(\sigma) \log\frac{1}{\beta} + O(n(n-1))}{1-\beta}$.

Putting $\beta = e^{-\epsilon}$, we get a competitive ratio of $(1+\epsilon)$ for A with an additive cost of at most $O(n(n-1)(1 + 1/\epsilon))$. ∎

2.2 Algorithm B Algorithm B is a simpler variant of algorithm A. The basic difference between the two is that here, we pick the initial random numbers iid, and we update additively instead of multiplicatively. The algorithm proceeds as follows:

1. Pick n numbers $r_1, \cdots, r_n$ from $[0, 1/\epsilon]$ independently uniformly at random. Assign r_i to the i-th element in L.
2. Assign a weight w_i to each element in the list, which tracks the number of accesses to that element. Order elements in the list in decreasing order of $r_i + w_i$.
3. When an element x at position i is accessed, increase w_i by 1.
4. Move element x up if necessary to maintain ordering by weights.
5. Go to step 3 till last request is served.

Theorem 2.2. *Algorithm B is (1+$O(\epsilon)$)-competitive with the optimal static list.*

The proof of Theorem 2.2 closely follows the proof of 2.1. We first note that algorithm B also satisfies the Pairwise property. The proof is left to the reader. The following lemma analyzes performance of algorithm B on a two element list.

Lemma 2.4. *For a list of two elements, algorithm B has a competitive ratio of* $1 + O(\epsilon)$ *with the optimal static list.*

Proof. Consider a list L of two elements x and y. The probability that x precedes y after w_x accesses to x and w_y to y, is given by $Pr[r_x + w_x > r_y + w_y] = Pr[r_x - r_y > w_y - w_x]$. Let $\Delta = w_y - w_x$. Since both r_x and r_y are chosen uniformly at random from $[0, 1/\epsilon]$, this probability $p(\Delta) = \frac{(1-\epsilon\Delta)^2}{2}$ if $\Delta > 0$, and $p(\Delta) = 1 - \frac{(1+\epsilon\Delta)^2}{2}$ otherwise. When $\Delta > 1/\epsilon$ or $\Delta < -1/\epsilon$, $p(\Delta) = 0$ or 1 respectively.

As before, we consider the two lists (x, y) and (y, x) as experts. The probability of picking expert (x, y) then becomes $p(\Delta)$. $p(\Delta)$ takes on $2/\epsilon + 1$ distinct values corresponding to $\Delta \in \{-1/\epsilon, .., 1/\epsilon\}$. As the algorithm proceeds, this probability either moves one step up or one step down depending on whether (x, y) or (y, x) makes a mistake respectively.

The algorithm starts at $\Delta = 0$. As it moves through various values of Δ, we can couple every movement from $\Delta = x$ to $\Delta = x + 1$ with a movement from $\Delta = x + 1$ to $\Delta = x$, with at most $2/\epsilon$ movements left uncoupled. Notice that this coupling has the property that both experts make exactly one mistake among the two moves ((x, y) while moving up and (y, x) while moving down).

Algorithm B on the other hand, makes an expected number of mistakes equal to $p(x) + 1 - p(x + 1)$. For $x \geq 0$, $p(x)+1-p(x+1) = 1+\epsilon-\epsilon^2(x+1/2) < 1+\epsilon$. For $x < 0$, $p(x) + 1 - p(x + 1) = 1 + \epsilon + \epsilon^2(x + 1/2) < 1 + 2\epsilon$.

The total number of mistakes made by algorithm B on uncoupled steps are at most $O(1/\epsilon)$. Putting these together, we get a competitive ratio of $1+O(\epsilon)$ with the best static list with an additive term of $O(1/\epsilon)$.

Proof of Thm 2.2 The result follows, as before, from *List Factoring lemma* and Lemma 2.4. ■

Notice that both algorithms A and B can be thought of as randomized versions of the frequency count algorithm.

3 Combining Statically and Dynamically Optimal Algorithms

In the previous section we described two algorithms which are variants of the Experts algorithm and are $(1 + \epsilon)$ optimal with respect to the best static list. In this section we reapply the experts technique, this time using algorithm A (or B) and algorithm COMB [BEY98] as the two experts. Through this we can achieve both $(1+\epsilon)$ static optimality and $(1.6+\epsilon)$ dynamic optimality simultaneously.

The key idea is to use one of the experts for some number of accesses, and then probabilistically decide to switch the expert or stay with the same algorithm. However, in making the probabilistic decision, we do not want to explicitly calculate the weights of the two experts, as that would require running the two algorithms simultaneously, defeating the very purpose of this algorithm. In order to get around this difficulty, we use a variant of the Exp3 algorithm proposed by [ACBFS95]. This algorithm was proposed for the multi-armed bandit problem in which a gambler must decide to play on one of K slot machines, but gets to see the profit of only the machine that he is playing on.

[Bur00] extend the Exp3 algorithm to the case where there is a cost d for switching between the different experts, and so, we may want to switch after every s steps, rather than after every step. We will now describe the two algorithms Hedge and Hedge-Bandit as given in [Bur00].

3.1 Algorithm Hedge Algorithm Hedge uses K algorithms (experts), choosing one for each time step and obtaining a gain equal to the gain of the chosen expert in that time period. Expert i obtains a gain $x_i(t)$ in time step t with $x_i(t) \in [0, 1]$. The algorithm proceeds as follows:
$\eta > 0$ is a performance parameter. Gain of expert i at time step t is the total gain obtained by that expert upto time step t. That is, $G_i(t) = \sum_{j<t} x_i(j)\ \forall i$. Gain of the algorithm $G_H(t) = \sum_{j<t} x_{i_j}(j)$, where i_j is the expert chosen in step j.
In every time step, the algorithm chooses an expert i_t according to the distribution $p_i = \frac{e^{\eta G_i(t-1)}}{\sum_j e^{\eta G_j(t-1)}}$, and updates the gains according to the obtained vector $x(t)$.

3.2 Algorithm Hedge-Bandit Algorithm Hedge-Bandit is similar to Hedge except that it does not get to see the entire gain vector $x(t)$, but only gets to see the gain of the expert that it chooses at a particular time step. In each time step, the algorithm simulates Hedge to obtain a distribution using which it selects the expert for that time step. It then returns a "fake" gain vector $\hat{x}(t)$ to Hedge based on the gain $x(t)$ that it actually observed, in order to get the next distribution.

To avoid spending too much in switching between experts, this algorithm runs each expert for s steps before making the next decision. It obtains a gain of at most s in a single time segment, and looses at most d in switching from one expert to another. To use this algorithm for list update, in every time step, we return a gain of $1 - \frac{i}{n}$, if the access cost for an element is i. We assume that the algorithm runs for T time steps. The following describes the behavior of the algorithm in each time segment:
For time segments $t = 1, 2, ..., T/s$ do:

- Get $\hat{p}_i(t)$ from Hedge. Let $p_i(t) = (1-\gamma)\hat{p}_i(t) + \frac{\gamma}{K}$
- Select i_t according to $p_i(t)$.
- Use expert i_t for s steps observing a total gain of $x_{i_t}(t)$. Send gains $\hat{x}_j(t)$ to Hedge, where $\hat{x}_j(t) = \frac{x_{i_t}(t)}{p_{i_t}(t)}$ if $j = i_t$, and 0 otherwise.

THEOREM 3.1. (THEOREM 5.3 OF [BUR00]) *The expected gain of Hedge-Bandit is at least*

$$G^* - (1-\gamma)\frac{T}{s}d - \frac{1-\gamma}{\gamma}sKlnK - (e-1)\gamma G^*$$

where G^* is the largest total actual gain acquired by any single expert, and where $\eta = e^{\gamma/sn}$

For the List Update problem, we have $K = 2$, and $d = O(n^2)$. Reverting back to our loss model and choosing appropriate values for s and γ, we obtain the following result:

THEOREM 3.2. *Algorithm Hedge-Bandit using algorithm A and algorithm COMB as experts is* $(1 + 2\epsilon)$ *statically optimal and* $1.6(1 + \epsilon)$ *dynamically optimal in the Full Cost model, with an additive term of* $O(n^5/\epsilon^2)$ *for the List Update problem, if it makes a switching decision every* en^3/ϵ *steps and uses* $\gamma = \epsilon/en$.

Proof. From theorem 3.1,

$$G - (1-\gamma)\frac{T}{s}d - \frac{1-\gamma}{\gamma}sKlnK - (e-1)\gamma G$$

Putting $L_{HB} = n(T - G_{HB})$, $L^* = n(T - G^*)$ and using $G^* < T$, we obtain the following:

$$\begin{aligned} L_{HB} &\leq n\{T - G^* + (1-\gamma)\tfrac{T}{s}d \\ &\quad + \tfrac{1-\gamma}{\gamma}sKlnK + (e-1)\gamma T\} \\ &= L^* + (1-\gamma)\tfrac{T}{s}dn \\ &\quad + \tfrac{1-\gamma}{\gamma}nsKlnK + (e-1)\gamma nT \end{aligned}$$

But, $L^* \geq T$ in the full cost model (we always incur a cost of at least 1). So we have

$$L \leq L^*\{1 + (1-\gamma)\frac{dn}{s} + (e-1)\gamma n\} + \frac{1-\gamma}{\gamma}nsKlnK$$

Using the values $\gamma = \epsilon/en$, $s = \frac{en^3}{\epsilon}$, $d = n^2$ and $K = 2$ gives us the desired result:

$$L_{HB} \leq (1+\epsilon)L^* + 8.9\frac{n^5}{\epsilon^2}$$

We omitted a few details in the above description of the algorithm. When we switch from one algorithm to the other at the end of a phase, we need to know the state of the other algorithm at the end of that phase. It is not obvious that this is calculable, as we want to avoid running the other algorithm explicitly. However, it can be argued that extra overhead incurred in starting the algorithms in a "wrong" state is at most $O(n^2)$. Moreover, the state can be easily determined without running the algorithms explicitly, by keeping access counts for each element in the case of algorithm A, and keeping timestamps for each element in the case of COMB. We wish to clarify here, that though using these extra variables would help us determine the next state of the list, it does not help us calculate the losses made by the experts, which necessitates the use of the Hedge-Bandit algorithm.

4 Search trees

An on-line binary search tree algorithm is *dynamically optimal* if its total cost (sum of search cost and number of rotations) is never more than a constant times the total cost of the best off-line algorithm, for any sequence of accesses. Much work has gone into an attempt to prove the dynamic optimality of splay trees since Sleator and Tarjan made their conjecture in [ST85b]. We are unable to prove the dynamic optimality of splay trees or any other on-line search tree algorithm.

Instead, we show that it is possible to have *dynamic search-optimality*, a property much weaker than dynamic optimality. In general the total cost of an algorithm is its search cost, the sum of the depths of the accesses, plus its rotation cost, the number of rotations made. We say that our algorithm has dynamic search-optimality because its search cost is at most a constant times the *total cost* of any off-line algorithm. While our algorithm is exponentially slow, it is based on simple principles.

There are two main difficulties in achieving dynamic optimality. First of all, the algorithm has to decide which nodes to keep near the root[3]. Secondly, for full dynamic optimality it has to be able to get these nodes near the root without using too many rotations. We show that the first difficulty is not insurmountable. Equivalently, dynamic optimality cannot be proven impossible by any argument that neglects the rotation cost of the on-line algorithm.

There is no especially strong evidence suggesting that any BST algorithm is dynamically optimal. A natural approach to disproving this would be to present a set of sequences, and argue that no on-line algorithm can handle all these sequences in a dynamically optimal manner. The simplest form of this argument would be that the on-line algorithm cannot "guess" which node comes next and therefore has too many nodes to keep near the root. However, the existence of an algorithm with dynamic search optimality implies that this type of argument is not possible.

Wilber [Wil89] made some progress by proving lower bounds for off-line algorithms. In particular, he has shown that a random sequence of accesses in $\{1, 2, \ldots, n\}$ costs an expected $\Omega(\log n)$ per access for off-line algorithms. This is a necessary condition for dynamic optimality to be possible, because any on-line

[3] Since rotations are free, a natural idea is at each step, to choose the tree of the optimal off-line algorithm so far. This is not as simple as it sounds, because off-line optimality is ambiguous. For example, suppose you start with a 7 node "line" tree of depth 7. After several accesses to node 1, the deepest node, any optimal off-line algorithm must have brought it to the root. There are 132 equally optimal ways to do this, all using 6 rotations.

algorithm pays an expected $\theta(\log n)$ just in search cost (not counting rotations) since the average depth of a node is $\theta(\log n)$.

We go slightly farther in analyzing the cost of random sequences. We show that the number of sequences with off-line cost k is less than 2^{12k} for any k. This is also a necessary condition for dynamic optimality, for information-theoretic reasons[4]. Essentially, we give a way to describe off-line rotations in 12 bits per rotation, even though there are, in general, $n-1$ possible rotations one can perform.

4.1 Structural preliminaries We assume that the nodes in the tree are simply the numbers $1, 2, \ldots, n$, and let m be the length of the access sequence. We further assume that all algorithms, on- and off-line, begin with the same fixed tree, say, rooted at n and having depth n. In this section, we prove the following. The constant 12 is not really important: we are mostly interested in the fact that it is $2^{O(k)}$.

THEOREM 4.1. *The number of access sequences having optimal off-line cost k is at most 2^{12k}, for all $k \geq 0$, regardless of n or m.*

Proof. We use an information theoretic argument based on the fact that one can concisely describe any sequence having optimal off-line cost k. We argue that it is possible to describe any access sequence via the trees used in the optimal off-line algorithm. First, as several people have observed, we may assume that the off-line algorithm rotates the next node to be accessed to the root before each access, and we only lose a factor of two in the optimal cost. The reason is that given any off-line algorithm, we can modify it by making it rotate a node to the root immediately before accessing it, and then reverse these rotations to move the node back to where it was. If the node was at depth d, we have paid $2(d-1)$ in rotations (and incurred no additional search cost), whereas the former algorithm paid only $d-1$.

Now it suffices to describe the sequence of trees, because the accesses will just be their roots. To do this, we describe the set of rotations performed from each tree to the next, which we show how to do in at most $6r$ bits if there are r rotations. This implies that there are at most 2^{6k} possible sequences of cost k because there are at most 2^{6k} descriptions of length $6k$. We then get 2^{12k} because of the factor of 2 lost in our initial assumption.

What remains is to describe a set of r rotations using $6r$ bits. Like Lucas [Luc88], we think of a rotation as an edge rotation which changes a single edge from either left to right or right to left. Of course, the nodes adjacent to an edge may change. But based on our assumption that the next access is rotated to the root, it is not difficult to see that all the edges on its path to the root must be rotated at least once.

Lucas argues that without further loss of generality we may assume that the set of edges rotated by an optimal off-line algorithm form a single connected component that includes the root and the next node to be accessed. Briefly, this is because any rotations of edges not in this connected component could easily be delayed (using lazy programming) until they are in such a connected component. Their delay will not affect the search cost, since these rotations cannot affect the depth of the next request, nor does their delay affect the rotation costs.

Next, observe that regardless of the order of the rotations, we can completely describe the result of the rotations in $6w$ bits if there are w edges. First, we describe the subset of edges that were rotated one or more times. Since this is a rooted subtree, we can describe this using the symbols left (00), right (01) and up (1), to form a cycle that traverses each edge twice, using a total of $3w$ bits. Next we describe their position in the resulting tree. In the resulting tree, these edges will still be a rooted subtree, so we can describe them also with $3w$ bits. First note that the set of nodes in this subtree doesn't change even as the positions of the edges do. Secondly, notice that the shape of this subtree completely determines the positions of all the nodes, because this is a binary search tree. Finally, note that off-line algorithm has to perform at least w rotations.

Thus, we can describe the optimal sequence of trees (and thus the access sequence) in $6r$ bits if it performs r rotations. Since we lose a factor of two due to our first assumption, this proves the theorem.

4.2 Dynamic Search Optimality In this section, we will consider probability distributions. A tree can be thought of predicting the next access, where it predicts nodes closer to the root with higher probability. From our structural result, we see:

COROLLARY 4.1. *There is a probability distribution over arbitrary sequences of accesses, that assigns probability at least 2^{-13k} to an access sequence of optimal off-line cost k.*

Proof. Choose a cost k according to the distribution

[4]Given an on-line algorithm, such as splaying, any access sequence can be described by the location of each node in its corresponding tree. This can be described using 3 symbols (left, right, and stop) and has length proportional to the search cost. There is no need to describe the rotations performed by the on-line algorithm, since those can be determined from the access sequence. So, for any on-line algorithm, there are at most 3^k sequences costing less than k.

$1/2^k$. By Theorem 4.1, there are at most 2^{12k} sequences of that cost.

It is easy to convert a binary search tree into a probability distribution p such that $p(j) \geq 3^{-\mathrm{depth}(j)}$. Simply choose j by beginning at the root, going left, right, or stopping, each with probability 1/3 (when possible). It is also possible to convert a probability distribution into a tree.

OBSERVATION 4.1. *For a probability distribution p over individual accesses, we can create a binary search tree such that $depth(a) \leq 1 - \log p(a)$ for any node a.*

Proof. For the root, choose the first i such that $\sum_1^{i-1} p(i) \leq 1/2$ and $\sum_{i+1}^{n} p(i) \leq 1/2$. Recurse on the numbers less than i (normalizing p) to create the left subtree, and the numbers greater than i for the right subtree. It is easy to see that the total probability of any subtree rooted at depth d is at most $1/2^{d-1}$ so a node of probability $p(i)$ cannot be deeper than $-\log p(i)$.

We can combine these two ideas to make an on-line algorithm.

THEOREM 4.2. *For any probability distribution p over access sequences, we can create an on-line algorithm with search cost at most $m - \log p(a_1 a_2 \ldots a_m)$ for every access sequence $a_1 a_2 \ldots a_m$.*

Proof. The distribution p can be thought of as predicting the next access from the previous accesses. In particular, the conditional probability of the next access given the previous accesses is:

$$\begin{aligned} p_i(a_i) &= p(a_i | a_1 a_2 \ldots a_{i-1}) \\ &= \frac{\sum_{b_{i+1},\ldots,b_m} p(a_1 \ldots a_{i-1} a_i b_{i+1} \ldots b_m)}{\sum_{b_i,\ldots,b_m} p(a_1 \ldots a_{i-1} b_i b_{i+1} \ldots b_m)} \end{aligned}$$

We can write p as a product of the conditional distributions of access i, i.e.,

$$p(a_1 a_2 \ldots a_m) = \prod_1^m p_i(a_i).$$

Our on-line algorithm works as follows. For the ith access, we have enough information to compute p_i. We then convert p_i into a tree by the method of Observation 4.1. Thus, the depth of access a_i in this tree will be no more than $1 - \log p_i(a_i)$. Our total search over m accesses is at most

$$\sum_1^m 1 - \log p_i(a_i) = m - \log p(a_1 a_2 \ldots a_m).$$

COROLLARY 4.2. *There is an on-line algorithm that has dynamic search optimality. In particular, on any access sequence, its search cost is at most 14 times the optimal off-line total cost.*

Proof. We use the probability distribution of Corollary 4.1 in combination with Theorem 4.2 to get a total cost of m plus 13 times the optimal off-line cost. But m is no larger than the optimal off-line cost.

5 Conclusions and open problems

In this paper we have presented two results: an efficient algorithm achieving strong static optimality for the list-update problem, and an inefficient algorithm achieving dynamic search-optimality for trees. Several natural open problems are: Can strong static optimality be achieved efficiently for the case of trees? Can dynamic search-optimality be achieved efficiently for trees? And, of course, can we achieve true dynamic optimality (efficiently or inefficiently) for search trees?

Acknowledgements: We would like to thank Santosh Vempala for a number of helpful discussions. This material is based upon work supported under NSF grants CCR-9732705 and CCR-0105488, an NSF Mathematical Sciences Postdoctoral Research Fellowship, and an IBM Graduate Fellowship.

References

[ACBFS95] P. Auer, N. Cesa-Bianchi, Y. Freund, and R. Schapire Gambling in a rigged casino: The adversarial multiarmed bandit problem In *Proceedings of the 36th Annual Symposium on Foundations of Computer Science*, pages 322-331, November 1995.

[AvW95] Susanne Albers, Bernhard von Stengel, and Ralph Werchner. A combined BIT and TIMESTAMP algorithm for the list update problem. *Information Processing Letters*, 56:135–139, 1995.

[Bur00] C. Burch. Machine learning in metrical task systems and other on-line problems. *CMU Tech Report* CMU-CS-00-135, May 2000.

[BB00] A. Blum and C. Burch. On-line learning and the metrical task system problem. *Machine Learning*, 39(1):35–58, April 2000.

[BEY98] Allan Borodin and Ran El-Yaniv. *On-line Computation and Competitive Analysis.* Cambridge University Press, 1998. `http://www.cup.org/Titles/56/0521563925.html`.

[CLS01] Deepak Chawla, Lin Li, and Stephen Scott. Efficiently approximating weighted sums with exponentially many terms. In *Proceedings of the Fourteenth Annual Conference on Computational Learning Theory*, July 2001.

[FS96] Yoav Freund and Robert Schapire. Game theory, on-line prediction and boosting. In *Proc. 9th Conf. on Computational Learning Theory*, pages 325–332, 1996.

[KV00] Adam Kalai and Santosh Vempala. Efficient algorithms for universal portfolios. In *Proceedings of the 41st Annual Symposium on the Foundations of Computer Science*, November 2000.

[Luc88] Joan M. Lucas. Canonical forms for competitive binary search tree algorithms. Technical Report DCS-TR-250, Rutgers University, 1988.

[LW94] N. Littlestone and M. K. Warmuth. The weighted majority algorithm. *Information and Computation*, 108:212–261, 1994.

[ST85a] Daniel Sleator and Robert E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28:202–208, 1985.

[ST85b] Daniel D. Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32:652–686, 1985.

[Tei93] Boris Teia. A lower bound for randomized list update algorithms. *Information Processing Letters*, 47:5–9, 1993.

[Wil89] Robert Wilber. Lower bounds for accessing binary search trees with rotations. *SIAM Journal on Computing*, 18:56–67, 1989.