Optimal binary trees in online algorithms[1]

D. Sleator and M. Talupur

September 2002

CMU-CS-02-148

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Some binary search tree algorithms, such as splay trees, structure the tree in a way that depends on the history of accesses. In this paper we consider what happens if at each point in time the optimal binary search tree (for the access frequencies seen so far) is maintained. We prove lower and upper bounds on the competitive ratio (with respect to the final optimal tree) of such an algorithm

[1]This research was sponsored by National Science Foundation (NSF) grant no. CCR-0122581. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the NSF or the US government.

Keywords: optimal binary trees, online algorithms, competitive analysis

1 Introduction

An important idea in data structure design is that of *self-adjustment*, in which the data structure changes in response to the requests it sees. Self-adjusting data structures are often analyzed using *competitive analysis*, in which the performance of the data structure is compared to that of a hypothetical adversary who has the advantage of seeing the future. In this approach the worst-case ratio of the cost of the self-adjusting data structure in question is compared to an optimal off-line algorithm. The goal is to find self-adjusting algorithms that minimize this *competitive factor*.

In the usual framework for competitive analysis, the off-line optimum (adversary) is allowed to adjust the structure according to the same rules as the on-line algorithm. Blum *et. al* [1] asked what happens if we restrict the adversary so that it is not allowed to adjust its structure over time? They showed (among other things) that using this modified form of competitiveness, there exists a randomized binary search tree algorithm that is $(1+\epsilon)$ competitive. Kalai and Vempala [2] then constructed a specific and practical algorithm that achieves this. In this paper, we analyze a specific algorithm for maintaining a binary search tree.

More specifically, problem we consider is that of using a binary search tree to store a static set of n items. A sequence of access requests (searches) is applied to the tree. The cost incured in an access is simply the depth of the accessed node in the tree. Given a set of access frequencies, a well-known $O(n^2)$ algorithm [3] can be used to construct the optimal static tree for that particular frequency distribution. So one very plausible algorithm to consider is the one which, at every step, constructs the optimal tree for the current frequency distribution, and uses that for the next search. This is the algorithm we analyze in this paper. Without loss of generality (as explained below) we can ignore the cost of computing the optimal trees and not charge the algorithm for restructuring the tree at each step.

Kalai and Vempala [2] showed that a slightly modified version of this algorithm achieves $(1+\epsilon)$ competitiveness. Each element has a pseudo-frequency. This is initialized randomly according to a specific exponential distribution (depending on ϵ). From then on the pseudo-frequency is updated just like the frequency (it's incremented when the element is accessed). At each step the structure is simply that of the optimal tree built according to the pseudo-frequencies.

In this report, we analyze the performance of the algorithm that maintains an optimal binary search tree at every stage and see how it measures up against the best possible static offline algorithm. We give upper and lower bounds on the competitive ratio of this algorithm.

2 The algorithm

As described before our online algorithm computes an optimal tree for the request sequence so far after each request. We ignore the cost of computing the optimal tree and restructuring the binary tree at each stage for now. We consider only the search costs.

First we give a lower bound on the competitive factor which holds for any deterministic algorithm.

Theorem 2.1. *For any deterministic algorithm the competitive factor must be at least 4/3.*

Proof. Consider any deterministic online algorithm D and an adversary A.The adversary A identifies two elements from the tree that D maintains(we can assume that the tree D maintains has at least 2 elements, the case with one element is not interesting). Now one these elements must be at a depth at least 2 (depth of the root is taken to be 1). Since the algorithm D is deterministic the adversary A knows which element is at depth 2 (or lower) and A accesses it. Thus the adversary keeps accessing the lower of the two elements it has identified and at every step algorithm D incurs a cost of at least 2.

In the optimal tree for the whole sequence of requests, the element which was accessed most often is at the root and the other element will be at depth 2. This means the average cost per access is at most 1.5 since the element with higher access frequency is at depth 1 and the other element is at depth 2. Elements other than the two identified elements can be ignored as they are not accessed at all. Hence, the ratio of the cost of algorithm D to that of the final optimal tree is at least $2/1.5 = 4/3$. □

This result holds for any deterministic algorithm and so holds for the algorithm which maintains an optimal tree at every step. We now prove an upper bound on the competitive ratio of our algorithm. We make use of a result from [4] which gives an upper bound on the depth of an element in an optimal binary tree. The result states that if the weight (i.e. the number of requests) of the subtree rooted at node i is W_i, the weight of the whole tree is W and the depth of the node i is d then

$$\frac{W_i}{W} \leq \frac{2}{F_{d+3}}$$

where F_{d+3} is the *d+3* number in the fibonacci series 1, 1, 2, 3, 5, 8.... From this it follows that if weight (i.e the number of requests) of a element is w_i and its depth is d then

$$\frac{w_i}{W} \leq \frac{2}{F_{d+3}}$$

because the weight of the subtree rooted at element i is atleast w_i. Since $\frac{2}{F_{d+3}}$ is less than $\frac{1}{\phi^d}$, the depth of a node i (in the optimal tree) with weight w_i is bounded from above by $\log_\phi \frac{W}{w_i}$, where ϕ is the *golden ratio*, W is the total weight of the tree.We also use a result by Knuth which states that the average path length, C_{opt}, for an optimal tree satisfies

$$H - \log(\frac{eH}{2}) \leq C_{opt} \leq H + 1$$

where H is the entropy of the normalized weight sequence $p_1 \ . \ . \ p_n$. This inequality holds for $H \geq \frac{2}{e}$. This condition is satisfied for almost all sequences, the exceptions being extremely skewed distributions. Exercise 6.2.35 in [3] shows that the expected entropy of a randomly generated sequence is $\frac{ln(n)}{2}$. There is weaker result by Mehlhorn [5] that is independent of the entropy of the sequence. The result states

$$H/\log(3) \leq C_{opt}$$

To bound the competitive ratio of our algorithm, we first get an upper bound on the cost of a sequence of requests to the binary tree in terms of the entropy of the request sequence, H, the number of requests, W and the number of elements in the tree,n . We assume that the number of requests, $W = \Omega(n^2)$. Once we have an upper bound on the cost for a sequence of requests we can get an upper bound on the competitive factor using the results by Knuth, [3] and Mehlhorn, [5], which give lower bounds on the cost incurred by an optimal tree.

Theorem 2.2. *On any sequence of requests the cost of the algorithm that maintains an optimal tree at every stage is upper bounded by*

$$(\log_\phi 2)(W-1)H + (n+1)\log_\phi(\frac{W-1}{e}) + n^2$$

where H is the entropy of the sequence of normalized weights of the elements and C is as defined above. We assume that W, the number of accesses is $\Omega(n^2)$ where n is the number of elements.

Proof. Let W_i denote the total number of accesses to element i at the end of the requests. Then $\sum_i W_i = W$. Consider the optimal tree just before some access, say an access to element j. Let w'_i be the weight of element i and let $\sum_i w'_i = W'$. Then the cost of the access to element j is upper bounded by:

$$\log_\phi(\frac{W'}{w'_j})$$
$$= \log_\phi(W') - \log_\phi(w'_j)$$

Consider the first term in the upper bound, $\log_\phi W'$ the weight of the tree. The first terms of all the upper bounds sum to $\log_\phi(W-1)!$, because the total weight of the tree increases from 0 to $W-1$ in steps of size 1. Note that for the first access to a element the above bound will not be valid as the weight of the element would be 0. For the first access n is an upper bound on the access depth. For n elements thsum of all the upper bounds is n^2. We will account for this at the end. For the second term, $\log_\phi(w'_i)$ where w'_i is the weight of the element i, consider the upper bounds on all accesses to a particular element i. For these accesses the second term sums to $\log_\phi(W_i-1)!$, because the weight of the element i increases from 1 to $W_i - 1$ in steps of size 1. So the sum of the second terms over all the accesses is $\sum_i \log_\phi(W_i-1)!$. So the sum of the upper bounds on the accesses is

$$\log_\phi(W-1)! - \sum_i \log_\phi(W_i-1)!$$

Using Sterling's formula, n! $= \sqrt{2\pi n}(\frac{n}{e})^n(1+\frac{1}{12n}+O(\frac{1}{n^2}))$ and ignoring the $(\frac{1}{12n}+O(\frac{1}{n^2}))$ part for now, the above sum is less than

$$(W-1)\log_\phi(\tfrac{W-1}{e}) + \tfrac{1}{2}\log_\phi(2\pi(W-1)) - \textstyle\sum_i (W_i-1)\log_\phi(\tfrac{(W_i-1)}{e})$$
$$-\textstyle\sum_i(\tfrac{1}{2}\log_\phi(2\pi(W_i-1)))$$

We will ignore the $\frac{1}{2}\log_\phi(2\pi(W-1))$, $\frac{1}{2}\log_\phi(2\pi(W_i-1))'s$ for now, because they are small compared to the other terms. The above expression then reduces to

$$(W-1)\log_\phi(\frac{W-1}{e}) - \sum_i (W_i-1)\log_\phi(\frac{(W_i-1)}{e})$$

$$= (W_i-1)\log_\phi(\frac{W-1}{W_i-1}) + (n-1)\log_\phi(\frac{W-1}{e})$$

$$= (W-1)(\frac{W_i-1}{W-1}\log_\phi(\frac{W-1}{W_i-1})) + (n-1)\log_\phi(\frac{W-1}{e})$$

$$\leq (W-1)(\frac{W_i}{W}\log_\phi(\frac{W}{W_i})) + (n-1)\log_\phi(\frac{W-1}{e})$$

$$= \log_\phi(2)(W-1)H + (n-1)\log_\phi(\frac{W-1}{e})$$

The terms we ignored previously are upper bounded by

$$(\frac{1}{12(W-1)} + O(\frac{1}{(W-1)^2}))(W-1)\log_\phi(\frac{W-1}{e}) + \frac{1}{2}\log_\phi(2\pi(W-1))$$

Taking this into account, the sum of the upper bounds on the access costs is itself upper bounded by

$$\log_\phi(2)(W-1)H + (n-1)\log_\phi(\tfrac{W-1}{e}) + (\tfrac{1}{12(W-1)} + O(\tfrac{1}{(W-1)^2}))(W-1)\log_\phi(\tfrac{W-1}{e})$$
$$+\tfrac{1}{2}\log_\phi(2\pi(W-1))$$

For W much greater than n, the number of elements in the tree, the above expression is upper bounded by

$$(\log_\phi 2)(W-1)H + (n+1)\log_\phi(\frac{W-1}{e})$$

Adding the upper bound n^2 on the cost of first access to each of the n elements to the expression above we have the required upper bound on the cost of the request sequence. $\square$

Since

$$C_{opt} \geq H - \log(\frac{eH}{2})$$

for entropy greater than 2/e the competitive factor of the algorithm is upper bounded by

$$\frac{(\log_\phi 2)(W-1)H + (n+1)\log_\phi(\frac{W-1}{e})}{W(H - \log(\frac{eH}{2}))}$$

which for W much greater than n is close to

$$\frac{(\log_\phi 2)H}{H - \log(\frac{eH}{2})}$$

Using Mehlhorn's result we can obtain a weaker bound that holds at all entropies. First note that

$$(\log_\phi 2)(W-1)H + (n+1)\log_\phi(\frac{W-1}{e}) \leq (\log_\phi 2)(W)H + (n+1)\log_\phi(\frac{W-1}{e})$$

Since $C_{opt} \geq H/\log(3)$ the competitive factor of our algorithm is upper bounded by

$$\frac{\log_\phi(2)(W)H + (n+1)\log_\phi(\frac{W-1}{e})}{WH/\log(3)}$$

which for $W = \Omega(\mathrm{n}^2)$ equals $\log_\phi(2)\log(3) = \log_\phi 3$

We have so far ignored the cost of finding an optimal tree at every stage. Optimal tree for a given sequence of weights can be found in $O(n^2)$ time. Instead of computing optimal tree at every stage we can compute an optimal tree once every $O(n^2)$ steps. This will ensure that cost of computing optimal trees is distributed over $O(n^2)$ steps and hence will be negligible. So we only need to prove that fixing an optimal tree for $O(n^2)$ steps does not cause much deterioration in the performance. The following theorem proves that there is no significant change in the cost of our algorithm over any sequence of requests.

Theorem 2.3. *The overhead for fixing the tree for $O(n^2)$ steps is a constant per step and it can be made as small as wanted by increasing the number of steps for which the tree is fixed. The number of requests, W, is assumed to be $\Omega(n^2)$, where* n *is the number of elements.*

Proof. Consider a block of $d \cdot n^2$ requests/steps, where d is some constant, with the optimal tree fixed just before the beginning of the block. Let the weight of element i be w_i and the total weight of the tree be W. During the $d \cdot n^2$ steps, let i be accessed f_i times. $\sum f_i = d \cdot n^2$. If the algorithm had continued using the updated optimal tree after each access then the cost for the $d \cdot n^2$ steps would have been

$$\log(\frac{(W + d \cdot n^2 - 1)!}{(W-1)!}) - \sum \log(\frac{(w_i + f_i - 1)!}{(w_i - 1)!}$$

using similar reasoning as in the previous theorem. The cost of for the $d \cdot n^2$ steps with tree fixed is $\sum f_i \log(\frac{W}{w_i})$. The difference of these two is

$$\sum f_i \log(\frac{W}{w_i}) - \log(\frac{(W + d \cdot n^2 - 1)!}{(W-1)!}) + \sum \log(\frac{(w_i + f_i - 1)!}{(w_i - 1)!}$$

Using Sterling's formula and using the same approximation as previously used, the expression is equal to

$$\sum f_i \log(\tfrac{W}{w_i}) - (W + d \cdot n^2 - 1)\log(\tfrac{W + d \cdot n^2 - 1}{e}) + (W-1)\log(\tfrac{W-1}{e})$$
$$+ \sum (w_i + f_i - 1)\log(\tfrac{w_i + f_i - 1}{e}) - \sum (w_i - 1)\log(\tfrac{w_i - 1}{e})$$

Noting that $\sum w_i = W$ and $\sum f_i = d \cdot n^2$, the above expression reduces to

$$\sum f_i \log(\tfrac{W}{w_i}) - (W + d \cdot n^2 - 1)\log(W + d \cdot n^2 - 1) + (W-1)\log(W-1)$$
$$+ \sum (w_i + f_i - 1)\log(w_i + f_i - 1) - \sum (w_i - 1)\log(w_i - 1)$$

$$\sum f_i \log(\tfrac{W}{w_i}) - (W + d \cdot n^2 - 1)\log(1 + \tfrac{d \cdot n^2 - 1}{W})W + (W-1)\log(W-1)$$
$$+ \sum (w_i + f_i - 1)\log[w_i(1 + \tfrac{f_i - 1}{w_i})] - \sum (w_i - 1)\log(w_i - 1)$$

$$\sum f_i \log(\tfrac{W}{w_i}) - (W + d\cdot n^2 - 1)\log(W) - (W + d\cdot n^2 - 1)\log(1 + \tfrac{n^2-1}{W}) + (W-1)\log(W-1)$$
$$+\sum(w_i + f_i - 1)\log(w_i) + \sum(w_i + f_i - 1)\log(1 + \tfrac{f_i - 1}{w_i}) - \sum(w_i - 1)\log(w_i - 1)$$

Since log(1+y) is at most ylog(e), the above expression is at most

$$\sum f_i \log(\tfrac{W}{w_i}) - (W + d.n^2 - 1)\log(W) - \log(e)(W + d.n^2 - 1)\tfrac{d\cdot n^2 - 1}{W} + (W-1)\log(W-1)$$
$$+\sum(w_i + f_i - 1)\log(w_i) + \sum\log(e)(w_i + f_i - 1)\tfrac{f_i - 1}{w_i} - \sum(w_i - 1)\log(w_i - 1)$$

Since $\sum f_i = d\cdot n^2$ the above expression is equal to

$$(W-1)\log(W-1) - (W-1)\log(W) + \sum(w_i - 1)\log(w_i) - \sum(w_i - 1)\log(w_i - 1)$$
$$-\log(e)(W + d\cdot n^2 - 1)\tfrac{d\cdot n^2 - 1}{W} + \sum\log(e)(w_i + f_i - 1)\tfrac{f_i - 1}{w_i}$$

$$\leq \sum(w_i - 1)\log(\frac{w_i}{w_i - 1}) - \log(e)(d\cdot n^2 - 1) - \log(e)\frac{(d\cdot n^2 - 1)^2}{W} + \sum\log(e)(f_i - 1 + \frac{(f_i - 1)^2}{w_i})$$
$$\leq \sum(w_i - 1)\log(1 + \frac{1}{w_i - 1}) + \log(e)(n-1) - \log(e)\frac{(d\cdot n^2 - 1)^2}{W} + \sum\log(e)(\frac{(f_i - 1)^2}{w_i})$$
$$\leq \sum\log(e)(w_i - 1)\frac{1}{w_i - 1} + \log(e)(n-1) + \sum\log(e)(\frac{(f_i - 1)^2}{w_i})$$
$$= \log(e)(2n-1) + \sum\log(e)(\frac{(f_i - 1)^2}{w_i})$$

The expression $\sum\log(e)(\frac{(f_i-1)^2}{w_i})$ is maximum when $f_k = n^2$, where k is such that w_k is the smallest among all weights and all other f_i's are 0. The maximum value of the above expression then is

$$= \log(e)(2n-1) + \log(e)(\frac{(d\cdot n^2 - 1)^2}{w_i})$$

If $w_i \geq d^2 \cdot n^2$, then the upper bound would be less than or equal to

$$= \log(e)(2n-1) + \log(e)(n^2)$$

Note that w_i need not be $O(n^2)$, but it can be so only the first few times the element i is accessed. So ignoring these first few times, the overhead of fixing the tree for a block is at most

$$\log(e)(2n-1) + \log(e)(n^2)$$

per block. This overhead can be distributed over the accesses in the block. Then the overhead per step is clearly a constant inversely dependent on the block size. So we can make the overhead to as small as we want by increasing the block size. Hence the theorem. □

3 Conclusion

We considered an online algorithm which maintains an optimal binary search tree at each step and we proved upper and lower bounds on the competitive factor of such an algorithm with respect to the static optimal offline algorithm (which is the final optimal tree for the whole sequence). The lower bound we proved holds for any deterministic algorithm even when the adversary is offline oblivious. We gave two upper bounds, the first one is a stronger result, but it depends on the entropy of the request sequence. The second upper bound is independent of the entropy of the request sequence. For both the upper bounds we assumed that the number of requests is Ωn^2, where n is the number of the elements in the tree. This result in conjunction with the result from Kalai and Vempala, [2], demonstrates the power of randomization techniques again.

References

[1] Shuchi Chawla Avrim Blum and Adam Kalai. Static optimality and dynamic search optimality in lists and trees. In *Symposium on Discrete Algorithm (SODA'02)*, 2002.

[2] Adam Kalai and Santosh Vempala. Follow the leader algorithm. *To be published*, 2001.

[3] Donald E. Knuth. *The art of computer programming, Vol 3.* Addison Wesley, 1997.

[4] D. S. Hirschberg L. L. Larmore and M. Molodowitch. Subtree weight ratios for optimal binary search trees. Tech. rpt. 86-02, ics dept., UC Irvine, 1986.

[5] Kurt Mehlhorn. Nearly optimal binary search trees. *Acta Informatica*, pages 5:287–295, 1975.