Binary Searching with Non-uniform Costs and Its Application to Text Retrieval[1]

Gonzalo Navarro[2]
Eduardo Fernandes Barbosa[3]
Ricardo Baeza-Yates[2]
Walter Cunto[4]
Nivio Ziviani[3]

Abstract

We study the problem of minimizing the expected cost of binary searching for data where the access cost is not fixed and depends on the last accessed element, such as data stored in magnetic or optical disk. We present an optimal algorithm for this problem that finds the *optimal* search strategy in $O(n^3)$ time, which is the same time complexity of the simpler classical problem of fixed costs. Next, we present two practical linear expected time algorithms, under the assumption that the access cost of an element is independent of its physical position. Both practical algorithms are online, that is, they find the next element to access as the search proceeds. The first one is an *approximate* algorithm which minimizes the access cost disregarding the goodness of the problem partitioning. The second one is a *heuristic* algorithm, whose quality depends on its ability to estimate the final search cost, and therefore it can be tuned by recording statistics of previous runs.

We present an application for our algorithms related to text retrieval. When a text collection is large it demands specialized indexing techniques for efficient access. One important type of index is the suffix array, where data access is provided through an indirect binary search on the text stored in magnetic disk or optical disk. Under this cost model we prove that the optimal algorithm cannot perform better than $\Omega(1/\log n)$ times the standard binary search. We also prove that the approximate strategy cannot, on average, perform worse than 39% over the optimal one. We confirm the analytical results with simulations, showing improvements between 34% (optimal) and 60% (online) over standard binary search for both magnetic and optical disks.

Key words: Optimized binary search, non-uniform costs, text retrieval, secondary memory.

1 Introduction

The problem of searching with non-uniform access costs arises in many different areas: distributed systems, databases, robotics, text retrieval and geographic information systems, among others. Searching with non-uniform access costs means that each element of the set has an arbitrary access cost, which in the most general case may depend on the full access history of the search, for example if each access modifies all the subsequent costs in some way.

[1]The authors wish to acknowledge the financial support from the Brazilian CNPq - Conselho Nacional de Desenvolvimento Científico e Tecnológico Grant No. 520916/94-8, Project PRONEX, Fondecyt Grant No. 95-0622, Programa de Cooperación Científica Chile-Brasil de Fundación Andes, and Project RITOS/CYTED.

[2]Departamento de Ciencias de la Computación, Universidad de Chile, Santiago, Chile
[3]Departamento de Ciência da Computação, Universidade Federal de Minas Gerais, Belo Horizonte, Brazil
[4]Departamento de Computación, Universidad Simón Bolívar. Caracas, Venezuela

The simple case where each element has a fixed access cost (not modified by the search process) is considered in [1, 12]. In particular, [12] shows an $O(n^3)$ algorithm to build the optimal search strategy. A related problem in which the elements have the same access cost but different access probability is considered in [7]. They present an $O(n^3)$ time solution to build the optimal search strategy. In [13], an $O(n^2)$ solution is devised for the same problem. In [11, 13], under the further restriction that all searches are unsuccessful, an $O(n \log n)$ algorithm is presented. Moreover, it is easy to combine [12] and [7] to obtain an $O(n^3)$ algorithm for different access costs and probabilities, or for the worst case optimal strategy. It is also easy to find counterexamples showing that the techniques of [11, 13] do not work for different access costs.

In this paper we study a particular case of the full access history case, namely when we perform a binary search on an array, and the cost to access each element depends on the previous element accessed. This is the case of binary searching an array on disk, since the last element accessed determines the position of the disk head, which alters the access cost of every element. We present an $O(n^3)$ algorithm to find the optimal search strategy for this problem. This is the same complexity as in the case of fixed costs, which makes the algorithm an important contribution. We prove that for a fixed cost model, the optimal algorithm cannot improve over the standard binary search by a factor higher than $\Omega(\log n)$. This holds under an *independence assumption*: the access cost and the positions of the elements in the array are independent.

We also develop two practical algorithms for the same problem. Those algorithms do not necessarily yield the optimal solution but they give a reasonably good approximation at much less CPU cost ($O(n)$ in total, under the independence assumption). Both algorithms are online, i.e. they find the next element to access as the search proceeds. By only solving the necessary subproblems a lot of time is saved, at the expense of a non-optimal search strategy. The first one is an approximate algorithm, which disregards the goodness of the partition and considers only the access cost of the elements. The other algorithm is a heuristic, whose quality depends on its ability to estimate the final search cost, and therefore it can be improved over the time by storing the results of previous runs.

We also present an application for our algorithms related to text retrieval. When a text collection is large it is necessary to build an index for efficient retrieval. An important type of index is the PAT array [8, 9] or suffix array [14], which is an array of pointers to the text which are lexicographically sorted. Data access using PAT arrays is provided through an indirect binary search on the text which is usually stored in magnetic disk or optical disk. The independence assumption holds in this application.

We prove that the strategy delivered by the approximate algorithm does not cost on average more than 39% over that of the optimal algorithm. That is, our approximate algorithm is 1.39-optimal, where the optimality is measured over the average cost of the search trees delivered. We also prove that the optimal search strategy cannot cost less than $\Omega(1/\log n)$ times the binary search cost, on average.

We also present simulation results on both optimal and practical algorithms. For one gigabyte of text stored in magnetic disks the performance of the optimal strategy is 35% of standard binary search, while the approximate and heuristic algorithms give solutions which cost 64% and 54%, respectively, of standard binary search. Similarly, for 256 megabytes of text stored in CD-ROM disks the optimal strategy is 34% of standard binary search, while the approximate and heuristic algorithm obtain 69% and 65%, respectively. Some of the results of this paper were presented in [6].

The outline of this paper follows. In Section 2 we formally describe the problem. In Section 3 we present and analyze an algorithm to find the optimal search strategy. In Section 4 we develop cheaper practical algorithms for the same problem. In Section 5 we analyze the optimality of the techniques. In Section 6 we show an application related to text retrieval. In Section 7 we draw some simulation results related to the application. Finally, in Section 8 we present our conclusions and future work directions.

2 Problem Description

Let $A[1..n]$ be a sorted array. Without loss of generality, we assume ascending order. We define a *cost function*

$$w : [1..n] \times [1..n] \longrightarrow \mathcal{R}$$

where $w(i,j)$ is the cost of accessing $A[j]$ given that the last accessed element was $A[i]$. This models the fact that the cost depends on the last element accessed.

In the same way we model the access frequency of the array. There are two closely related variants of the problem: successful and unsuccessful search. In the successful case we find the searched element in the array, while in the unsuccessful case the element is not found, and the search ends between two adjacent elements of the array where the searched element should be. More generally, we model the access frequency with two *probability functions*

$$p : [1..n] \longrightarrow \mathcal{R}$$

and

$$q : [1..n+1] \longrightarrow \mathcal{R}$$

where $p(i)$ is the probability that the i-th element of the array is searched and $q(j)$ is the probability that the search is unsuccessful and ends between the elements $j-1$ and j of the array. The elements 1 and $n+1$ stand for the searches that lie outside the range of the array elements. We also define

$$P(i,j) = q(i) + p(i) + q(i+1) + p(i+1) + ... + q(j) + p(j) + q(j+1)$$

which stands for the probability that the search lies between the elements i and j in the array (including the surrounding unsuccessful searches between $i-1..i$ and $j..j+1$).

A *search path* on A is any sequence of positions $i_0, i_1, .., i_\ell$. The *cost* of such a search path is defined as

$$w(i_0, i_1) + w(i_1, i_2) + ... + w(i_{\ell-1}, i_\ell)$$

Standard binary searching (i.e. access the middle of the array, then the middle of the remaining portion, and so on) involves tracing a search path driven by the comparisons between the searched element and the elements of the array. If we draw the balanced search tree for A, every path from the root to a leaf (the leaves representing inter-elements positions, with no data) is a possible search path for an unsuccessful search. A successful search is represented by a path from the root to an internal node. In this sense, the tree represents the search plan, accounting for every possible result obtained from the comparisons.

However, *any* binary search tree over A is valid. Our problem is how to devise a search plan whose total cost is minimized. That is, to build a binary search tree where either the

average or the worst case cost over all the search paths is minimized (the trees to minimize the average and the worst case are not necessarily equal).

It is clear that a balanced search tree is optimal on average for uniform costs and uniform access probabilities. This case corresponds to $w(i,j) = 1$ and either $p(i) = 1/n,\ q(j) = 0$ (successful case) or $p(i) = 0, q(j) = 1/(n+1)$ (unsuccessful case). Balanced trees are also optimal for uniform costs and worst case (in this case the access probabilities are not important).

We first show an algorithm for cost functions that depend only on the element to be accessed. It uses a dynamic programming scheme adapted from the ideas presented in [7, 13], where the optimal tree for $A[i..j]$ is built by previously knowing the optimal solution to all the subintervals. The formula for the optimal worst case tree is

$$cost(i,j) = \min_{k \in i..j} \{w(k) + \max(cost(i,k-1), cost(k+1,j))\}$$

where $cost(i+1,i) = 0$ for all i. In the worst case, all searches are unsuccessful. In the average case, the successful search can terminate at an internal node, while an unsuccessful search always ends in a leaf (whose inspection cost is taken as zero). The formula for the optimal average case search tree is

$$cost(i,j) = \min_{k \in i..j} \left\{ w(k) + \frac{P(i,k-1)}{P(i,j)}\ cost(i,k-1) + \frac{P(k+1,j)}{P(i,j)}\ cost(k+1,j) \right\}$$

3 An Algorithm to Find the Optimal Search Strategy

Since we are binary searching in the array, the key idea to solve the problem for the more complex cost function $w(i,j)$ that depends on the last position visited comes from noticing that if we have to solve the problem $A[i..j]$, it is because we have just accessed either $A[i-1]$ or $A[j+1]$. Thus, we proceed as before, this time filling two matrices: *Lcost* stores the cost when we have just read the left extreme $(i-1)$ of the array and *Rcost* stores the cost when we have just read the right extreme $(j+1)$ of the array.

The two matrices are filled in a synchronized manner. For the worst case the two corresponding formulas are

$$Lcost(i,j) = \min_{k \in i..j} \{w(i-1,k) + \max(Rcost(i,k-1), Lcost(k+1,j))\}$$

$$Rcost(i,j) = \min_{k \in i..j} \{w(j+1,k) + \max(Rcost(i,k-1), Lcost(k+1,j))\}$$

while for the average case they are

$$Lcost(i,j) = \min_{k \in i..j} \left\{ w(i-1,k) + \frac{P(i,k-1)}{P(i,j)}\ Rcost(i,k-1) + \frac{P(k+1,j)}{P(i,j)}\ Lcost(k+1,j) \right\}$$

$$Rcost(i,j) = \min_{k \in i..j} \left\{ w(j+1,k) + \frac{P(i,k-1)}{P(i,j)}\ Rcost(i,k-1) + \frac{P(k+1,j)}{P(i,j)}\ Lcost(k+1,j) \right\}$$

The algorithm is $O(n^3)$ time, the same cost of the simpler problem. At the top level we may assume a fixed initial position, or solve the top level problem for every initial position. This does not affect the whole complexity, since the solutions to the subproblems are the same.

This optimal algorithm needs to build the complete search strategy before proceeding with the search. This is acceptable if we can build the tree beforehand and later on process a large number of queries with the already built search tree. Although we need $O(n^2)$ space to build the optimal tree, only $O(n)$ space is needed to store it.

On the other hand, if we are going to perform a single query, the CPU cost of building all possible optimal paths may outweight the gains in many cases. In the next section we present two good search strategies to cope with this problem.

4 Two Online Practical Algorithms

The construction of the optimal search tree to solve a single query may be prohibitive in certain applications. We present two practical algorithms that choose a good next access point at each iteration of the search process, without building all possible search paths.

Both algorithms build the search path top-down, instead of the bottom-up dynamic programming strategy. In fact, the search tree is not completely built, only the path that we actually need to solve this particular query. This is why the algorithms are called "online": they are run for each query, unlike the optimal one which is run once and gives the optimal answer for any possible future query. In this case the obtained search path is not necessarily optimal. We call the first practical algorithm an "approximate" algorithm and the second one a "heuristic", because in the first case we can prove an optimality bound, while the second one depends heavily on heuristic decisions.

We show that both algorithms have an average CPU cost of $O(n)$ if some assumptions hold. This is much less than the $O(n^3)$ cost of the optimal algorithm.

4.1 An Approximate Algorithm

The optimal algorithm tries to balance, at each moment, the cost to access a given element and the goodness of the resulting partition. Binary search can be seen as an algorithm that simply disregards the access cost and optimizes the goodness of the partition. Our approximate algorithm, on the other hand, simply disregards the goodness of the partition and optimizes the access cost.

At the beginning of the search we consider the cost to access all the elements and then select the cheapest one. The comparison against this element will split the array in two parts (probably not equal), and the search process will discard one of the two parts. We now select, from the new subproblem (i.e. the selected partition), the element which is cheapest to access, and so on. Notice that the costs in the second iteration have no relation to those of the first iteration, since they depend on the last element accessed.

Formally, instead of the previous *Lcost* and *Rcost* optimal cost functions, we define *Acost* as the average cost for the approximate algorithm. Since we proceed top-down, we know which is the last accessed element, and hence *Acost* takes a third argument, h, that indicates the current access point. On average, we have

$$Acost(i,j,h) = w(h,k) + \frac{P(i,k-1)}{P(i,j)}\ Acost(i,k-1,k) + \frac{P(k+1,j)}{P(i,j)}\ Acost(k+1,j,k)$$

while to optimize the worst case we have

$$Acost(i,j,h) = w(h,k) + \max(Acost(i,k-1,k), Acost(k+1,j,k))$$

where in both cases k minimizes $w(h,k)$. Notice that the algorithm takes the same decision regardless of whether we are optimizing the worst case or the average cost, and regardless of the access probabilities.

We analyze now the performance of this algorithm. It is important to notice that what we are analyzing is the CPU cost to obtain the search strategy, *not* the cost of the obtained search strategy. This last cost is the subject of the next section. As we will see shortly, however, both analysis are related.

At each step we search the minimum cost of the current array partition. This search costs $O(j-i)$ time. Initially $j-i=n$, but the next subproblems become smaller and smaller. The additional space requirement is $O(1)$.

In the worst case this algorithm is $O(n^2)$, since the next partition may have only one element less than the current one. This is the case, for instance, of the cost being an increasing function of the distance between the last accessed element and the current element (or just an increasing function in case of fixed access costs). Assume for instance that we begin accessing $A[1]$. The next element to access will be $A[2]$, then $A[3]$ and so on. There will be n iterations of the algorithm for a total cost of $O(n^2)$. However, if this fact is known beforehand we do not even need to run this algorithm to find the minimum for each subproblem.

On the other hand, the analysis is very different under an *independence assumption*: the access costs and the positions of the elements in the array are not related, i.e. the position of the elements in the array can be considered as a random function of their access cost. In fact, this approximate algorithm only works well if this assumption holds.

Under the independence assumption, picking the element with least access cost yields a random element of the array. That is, at each iteration the array is partitioned at a random position (instead of the middle as in binary search). This resembles the quicksort pivoting process. Let $T(i,j)$ be the average amount of work to perform for $A[i..j]$. We want to prove that $T(i,j) \leq 4(j-i)$, and hence $T(1,n) \leq 4n$. This would prove that the total amount of work is linear on average.

We prove it by induction on $j-i$. The base case is $T(i,i)=0$, which satisfies the condition. For $j > i$ we have on average

$$T(i,j) = (j-i) + \frac{1}{j-i} \sum_{k=i}^{j} \left(\frac{P(i,k-1)}{P(i,j)} T(i,k-1) + \frac{P(k+1,j)}{P(i,j)} T(k+1,j) \right)$$

which by the induction hypothesis is

$$T(i,j) \leq (j-i) + \frac{4}{j-i} \sum_{k=i}^{j} \left(\frac{P(i,k-1)}{P(i,j)} (k-i) + \frac{P(k+1,j)}{P(i,j)} (j-k) \right)$$

which since $P(k+1,j)/P(i,j) = 1 - (p(k)+P(i,k-1))/P(i,j) \leq 1 - P(i,k-1)/P(i,j)$, can be pessimistically rewritten as

$$T(i,j) \leq (j-i) + \frac{4}{j-i} \sum_{k=i}^{j} (g(k)\ (k-i) + (1-g(k))\ (j-k))$$

where we have defined $g(k) = P(i,k-1)/P(i,j)$, which is an increasing function of k, going from 0 to 1 as k goes from i to $j+1$. By rewriting the above equation as

$$T(i,j) \leq (j-i) + \frac{4}{j-i} \left(\sum_{k=i}^{j} (j-k) + \sum_{k=(i+j)/2}^{j} g(k)(2k-i-j) - \sum_{k=i}^{(i+j)/2} g(k)(i+j-2k) \right)$$

it is clear that the worst that can happen is that $g(k) = 0$ for $k < (i+j)/2$ and $g(k) = 1$ for $k > (i+j)/2$. In this case the above sum gives

$$T(i,j) \leq (j-i) + 3(j-i) = 4(j-i)$$

which proves our claim, i.e. we work $T(1,n) \leq 4n = O(n)$.

Still under the independence assumption, we can consider the worst case of the search. That is, we assume that the partition will be random but select the worst search path that can occur under this assumption. In this case we have the formula

$$T(i,j) = (j-i) + \frac{1}{j-i} \sum_{k=i}^{j} \max(T(i,k-1), T(k+1,j))$$

which yields $T(1,n) \leq 4.58\ n = O(n)$.

Another parameter of interest is the average number of iterations to perform on average (instead of the total amount of work). In the same spirit of the above analysis, we call $S(i,j)$ this number, which satisfies $S(i,i) = 0$ and

$$S(i,j) = 1 + \frac{1}{j-i} \sum_{k=i}^{j} \left(\frac{P(i,k-1)}{P(i,j)}\ S(i,k-1) + \frac{P(k+1,j)}{P(i,j)}\ S(k+1,j) \right)$$

By using a similar technique as above, we can prove by induction that

$$S(1,n) \quad \leq \quad \frac{\ln 2}{1 - \ln 2}\ \log_2 n \quad \leq \quad 2.26 \log_2 n \tag{1}$$

a result that does not change if we consider the worst case under the independence assumption.

This shows that the number of iterations is logarithmic and the total cost is linear if the independence assumption holds, in the worst or average case, no matter which the access probabilities are.

4.2 A Heuristic Algorithm

We show now a heuristic to obtain a good next access point in $O(n)$ time. The basic idea is to mimic the formula used by the optimal algorithm. However, since we are proceeding top-down, the information on the cost to solve the subtrees is not available. We replace that information with *estimated* values which depend only on the size of the subproblem. Those estimations can be obtained analytically or using previous runs of the same algorithm.

Formally, instead of the previous $Lcost$ and $Rcost$ optimal cost functions, we define as $Hcost$ the cost for the heuristic algorithm. For the average case we have

$$Hcost(i,j,h) = \min_{k \in i..j} \left\{ w(h,k) + \frac{P(i,k-1)}{P(i,j)}\ Ecost(k-i) + \frac{P(k+1,j)}{P(i,j)}\ Ecost(j-k) \right\} \tag{2}$$

and for the worst case it holds

$$Hcost(i,j,h) = \min_{k \in i..j} \{ w(h,k) + \max(Ecost(k-i), Ecost(j-k)) \}$$

where $Ecost(n)$ is the estimated cost to solve a problem of size n (using this very same heuristic algorithm).

The fact that $Ecost(n)$ depends only on the size of the problem is chosen for simplicity. That cost function can be more complex, for instance it may take into account where is located the segment of size s in the whole array (i.e. defining $Ecost(i,j)$), as well as which is the last element accessed (i.e. defining $L.Ecost(i,j)$ and $R.Ecost(i,j)$). However, since $Ecost()$ is determined analytically or based on previous runs, it is not immediate that it will be possible to successfully estimate a more complex version. The success of the heuristic depends heavily on a good estimation of $Ecost()$.

We analyze now the performance of this algorithm. At each step it is $O(j-i)$ time. As in the previous section we have that initially $j-i=n$ but the sizes of the problems are reduced in the subsequent iterations. The additional space requirement is also $O(1)$.

As in the previous case, this algorithm is $O(n^2)$ in the worst case. Under the independence assumption we can borrow the average result from the previous section using a simple argument: the heuristic algorithm selects with higher probability those elements which are in the middle of the array, while the approximate algorithm disregards the positions completely. This shows that the search tree of the heuristic algorithm must be at least as well balanced as that of the approximate algorithm. Since this last tree has an average leaf depth of $O(\log n)$, the same happens to the heuristic algorithm. The same can be said about the total cost of the algorithm, which keeps $O(n)$ since the tree is now better balanced than before. Formally, for this argument to be true it suffices that the probability of being selected increases as the elements are closer to the middle of the array. This happens whenever $Ecost()$ is an increasing function.

Interestingly, the same argument can be used to show that the optimal algorithm must produce a tree whose leaves are at depth $O(\log n)$. Moreover, the constant 2.26 obtained in the previous section (Eq. (1)) is an upper bound for the heuristic and optimal trees too.

If the independence assumption does not hold, we cannot prove in general a cost better than $O(n^2)$. However, under the fixed cost model (independent on the last element accessed) we have that for many cost functions the new subproblem of the optimal tree is of size ωn on average, where $1/2 \leq \omega < 1$ (see, e.g. [12]). After i iterations the size of the problem is $\omega^i n$. The algorithm therefore performs $\log_{1/\omega} n$ iterations, and the CPU cost is

$$n \left(\sum_{i=0}^{\log_{1/\omega} n} \omega^i \right) \leq \frac{n}{1-\omega}$$

which is $O(n)$. If the heuristic solution is reasonably close to the optimal one, this bound applies to the heuristic algorithm too.

5 Analysis of Optimality

We analyze in this section the competitive ratio of the cost obtained by the optimal algorithm against standard binary search, and the practical algorithms against the optimal one.

The results of this section are rather general. We could not obtain good general results for the case of access costs dependent on the last accessed element. However, we found interesting bounds for the optimality of our algorithms applied to a fixed cost function.

We obtain more specific results later on, for the particular cost functions involved in the text retrieval application. In that case we are able to prove very similar bounds for the specific cost function involved, which depends on the last element accessed.

5.1 The Optimal Algorithm

A first observation is that the upper bound $OptimalCost/StandardBinaryCost \leq 1$ is tight, since when all the access costs tend to a constant the optimal strategy is the standard binary search.

We obtain now a lower bound for $OptimalCost/StandardBinaryCost$ for the case of fixed access costs. Say that $c_1, ..., c_n$ are the costs $w(i)$ to access the elements, where the c_i are arranged in increasing order. The optimal search strategy gives a tree which is not well-balanced if it finds that this undesirable arrangement is outweighted by a better total search cost. Hence, the best that can happen to the optimal algorithm is that the balanced binary search tree is already optimal. This happens when the element with minimum cost c_1 is in the middle of the array, c_2 and c_3 are at positions $n/4$ and $3n/4$, c_4 to c_7 are at positions $n/8$, $3n/8$, $5n/8$ and $7n/8$, and so on. It is easy to obtain the average or worst-case cost formula for this case. For instance, the average uniform unsuccessful search cost is

$$OptimalCost \quad \geq \quad c_1 + \frac{1}{2}(c_2 + c_3) + \frac{1}{4}(c_4 + ... + c_7) + ... \quad \geq \quad \sum_{i=1}^{n} c_i/i$$

while the formula for standard binary search trees is obtained by considering that the costs are randomly placed in the tree and the search follows a complete path from the root to a leaf:

$$StandardBinaryCost \quad = \quad \frac{\log_2 n}{n} \sum_{i=1}^{n} c_i$$

which gives a lower bound for the competitivity of the optimal search cost. For instance, if $c_i = \Theta(i^\alpha)$, for $\alpha > 0$, then $OptimalCost/StandardBinaryCost = \Omega(1/\log n)$. The same result holds for exponentially increasing c_i values. On the other hand, the bound is $\Omega(1)$ if $c_i = O(\text{polylog}(i))$. This gives a lower bound on what can be expected from any optimization algorithm under the independence assumption.

5.2 The Approximate Algorithm

It is difficult to find a good general bound for the competitivity of the approximate algorithm. Although under the independence assumption we know that the number of iterations will be no more than $2.26 \log_2 n$ (Eq. (1)), and that we will access the cheapest elements, it is possible that the optimal algorithm accesses only the $\log_2 n$ cheapest elements and that the next element accessed by the approximate algorithm is arbitrarily expensive. We need more information about the cost function. We show later an example related to a specific application where we are able to prove an upper bound.

5.3 The Heuristic Algorithm

The competitivity of the heuristic algorithm against the optimal one is difficult to assess, since it is largely dependent on the $Ecost()$ estimation. For instance, if we use $L.Ecost()$ and $R.Ecost()$ as explained in Section 4.2, we are in principle able to make those estimations approach the real optimal $Lcost()$ and $Rcost()$ values, which makes the heuristic algorithm *equal* to the optimal one. For instance, this can be statistically obtained after a large number of queries where the real costs for all the intervals are recorded. The smaller intervals will converge first to the correct values and the larger will follow them inductively. The cost for

intervals of size 1 are correctly predicted from the beginning. Once all the intervals inside (i, j) are correctly computed, we will accurately compute (i, j). Moreover, we can keep track of which cells are already computed correctly. This can be seen as truly computing the bottom-up matrix along the querying process instead of doing it before answering any query (a kind of lazy evaluation approach).

On the other hand, if $Ecost()$ is incorrectly computed the heuristic algorithm can perform poorly. Assuming that the heuristic algorithm is at least as good as the standard binary search, we have that

$$\frac{HeuristicCost}{OptimalCost} \leq \frac{StandardBinaryCost}{OptimalCost} = O(\log b)$$

6 An Application to Text Retrieval

In an information retrieval environment the user expresses his information needs by providing strings to be matched and the information system retrieves those documents containing the user specified strings. When the text collection is large it demands specialized indexing techniques for efficient text retrieval. One important type of index for text retrieval is the PAT array [8, 9] or suffix array [14]. A PAT array is a compact representation of a digital tree called PAT tree, because it stores only the external nodes of the tree. A PAT tree is a Patricia tree [15] built on all the suffixes of a text database. The PAT tree, also called suffix tree, was originally described in [13]. Each position of the text is called a semi-infinite string or suffix, defined by its starting position and extending to the right as far as needed or to the end of the text to guarantee uniqueness. In a PAT array the data access is provided through an indirect sorted array of pointers to the data. Figure 1 illustrates the PAT array or suffix array for a text example with nine text positions.

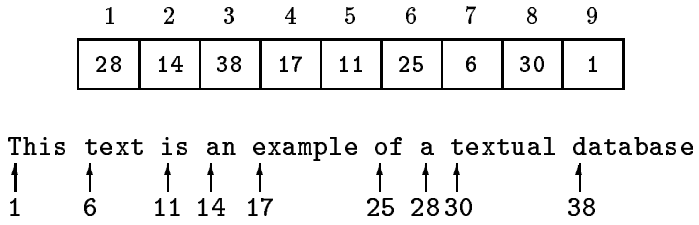

Figure 1: PAT array or suffix array.

This array allows fast retrieval using an indirect binary search on the text. However, we must access the text on disk to compare the query string against the suffix pointed to by a given position of the PAT array. Hence, the cost of the comparison is affected by the distance between the current disk head position and the disk position of the suffix in the text, as illustrated in Figure 2.

Since the visited disk positions are random, a naive strategy using a balanced search tree may be too expensive, since a random seek involves a significant cost, especially on optical disks. Then, it is reasonable to spend CPU time in order to save disk seek time.

In [4], an indexing mechanism is proposed to perform the main part of the search in main memory. However, there is always a final part that must be searched indirectly in disk, and

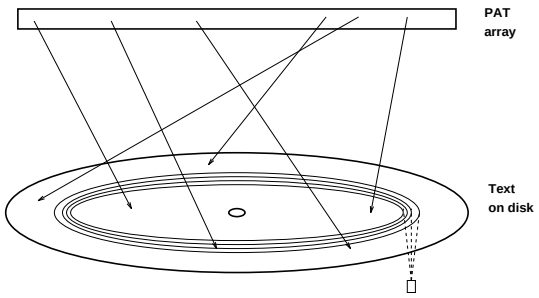

Figure 2: The physical disk model. Both the PAT array and the text are on disk (although the PAT block to solve is brought to main memory). The PAT array points to random positions of the text.

whose cost dominates the total search time. With this scheme, the final part of the PAT array can be kept in main memory too, though not the text.

Only unsuccessful searches are performed in the PAT array, since the search key is converted into two infinite-length limiting keys, which are those actually searched. For example, a binary search for the key `"texts"` in the example of Figure 1 converts it to the limiting keys `"texts"` (included) to `"textt"` (not included). This finds the text positions 6 and 30, corresponding to the interval from positions 7 to 8 of the PAT array. Every PAT array position is searched with the same probability in practice.

In the rest of this section we adapt the general searching algorithm to this problem (which is slightly more complicated). We first give a short introduction to magnetic and optical disk technology, then model the problem of searching using PAT arrays on disks, and finally present the optimal and online algorithms.

6.1 Disk Technology

To understand the cost model associated to disks, a short explanation on disk technology is needed [10, 3, 16]. Disks are divided in concentric *tracks* (or *cylinders* in the case of multiplate disks), which are subdivided in *sectors*. The sector is the minimum retrievable data unit. The reading device is a *disk head* that must move to the appropriate track and wait until disk rotation places the appropriate sector under the reading head.

Although the costs vary between magnetic and optical disks, in both cases there are three components: *seek time*, *latency time*, and *transfer time*. Seek time is the time needed to move the disk head to the desired disk track and therefore depends on the current head position. Latency time (or rotation time) is the time needed to wait for the desired sector to pass under the disk head. The average latency time is constant for magnetic disks and variable for CD-ROM disks, which rotate faster reading inner tracks than reading outer tracks. Transfer time is the time needed to transfer the desired data from the disk to main memory, which is proportional to the number of blocks transferred.

A simple seek cost model for magnetic disk considers the cost increasing linearly with the seek distance, in tracks. We consider a more sophisticated cost model [16], in which a single seek is composed of: (i) a *speedup* time, required to accelerate the disk arm until it reaches half of the seek distance or a fixed maximum speed; (ii) a *coast* time for long seeks, where the disk arm reachs its maximum speed; (iii) a *slowdown* time, needed for the disk arm to rest close to the desired track and (iv) a *settle down* time, where the disk head is precisely adjusted

to the desired track. *Very short seeks* ($\leq$ 4 tracks) are dominated by the settle down time ($\approx$ 1–3 milliseconds), while *short seeks* ($\leq$ 200–400 tracks) are dominated by the acceleration phase and the time is proportional to the square root of the seek distance plus the settle down time. *Long seeks* spend most of the time moving at a constant speed over a large number of tracks and the time is proportional to the seek distance plus a constant overhead.

The following definitions are used for the exact calculation of seek time. Let h be the current track, t the next track to access, D_{lim} the distance (in tracks) that bounds a short seek, $C_{short(long)}$ a constant overhead for short (long) seeks in milliseconds, $\theta_{short(long)}$ a proportionality factor for short (long) seeks in milliseconds/track. For $|h-t| \leq D_{lim}$, a short seek is

$$Seek(h,t) = C_{short} + \theta_{short} \times (|h-t|)^{1/2}$$

and for $(|h-t|) > D_{lim}$, a long seek is

$$Seek(h,t) = C_{long} + \theta_{long} \times |h-t|$$

Let $Access(h,t,n_s)$ (access cost function) be the time needed to read n_s sectors from track t, with the reading mechanism being currently on track h and S_{tt} the sector transfer time. Thus

$$Access(h,t,n_s) = Seek(h,t) + Latency + n_s \times S_{tt}$$

Let σ be the sum of the latency and transfer time in the magnetic disk. Hence, the total cost to read a sector in a track t, being on track h, is

$$Access(h,t,1) = \sigma + Seek(h,t)$$

Let T be the number of tracks occupied by a text file. For our purposes it is better for the file to be contiguously allocated on the disk to reduce seek time. That also means that it should use as few cylinders as possible, so it should fill cylinders as completely as possible (we discuss other situations in [6]).

Let b be the size of the current PAT block. Making the simplifying assumption of contiguous allocation, the average cost of naive binary search is

$$\left[\sigma + Seek\left(0, \frac{T}{3}\right)\right] \log_2(b+1)$$

since the average distance between two random text positions is $T/3$.

In CD-ROM disks the cost function is highly dependent on disk position and on the amount of the displacement of the access mechanism. An important feature to be considered is the *span size* capability Q, since inside the span the disk head is not physically displaced, and hence seek costs are negligible. The data access located within span boundaries requires a seek time of only 1 millisecond per additional track, while the access of tracks outside the span size may require 160 to 400 milliseconds.

Let α and β be the growing rate of the seek time as a function of the displacement of the access mechanism (in tracks) inside and outside the span, respectively. Let t_0 be the constant factor added when a track outside the span is accessed.

The total access cost to read a sector at track t with the head anchored at track h is

$$Access(h,t,1) = \begin{cases} \sigma + \alpha|h-t| & \text{if } |h-t| \leq Q/2 \\ \sigma + t_0 + \beta|h-t| & \text{if } |h-t| > Q/2 \end{cases}$$

This model is a simplification. The rotational latency is directly proportional to the position of the data on the disk, due to the constant linear velocity (CLV) physical format. We are using an average value. The seek time is linearized, although it also depends on the position on the disk.

Assuming contiguous allocation (quite realistic on optical disks) the naive search cost is

$$\left(\sigma + t_0 + \frac{T}{3}\beta\right) \log_2(b+1)$$

In Section 7 we give values to all the parameters of both cases.

6.2 Modeling the Problem

We explain now how the general problem of building the optimal search tree is adapted to this application.

When binary searching on PAT arrays the cost of the comparisons is affected by the distance between the current disk head position (which depends on the last position visited) and the disk position of the suffix (which is determined by the PAT array position to visit). This can be modeled with our cost function, namely

$$w(i,j) = Access(\mathrm{PAT}[i], \mathrm{PAT}[j], 1)$$

As explained, only unsuccessful searches are performed and every place is equally probable. This corresponds to the uniform unsuccessful case, i.e.

$$\frac{P(i,k-1)}{P(i,j)} = \frac{k-i+1}{j-i+2} \qquad \frac{P(k+1,j)}{P(i,j)} = \frac{j-k+1}{j-i+2}$$

Finally, since the PAT array is in practice a random permutation of the text suffixes, the independence assumption holds in this application.

We need now the following definitions. Let b be the size of the current PAT block. Note that b is reduced at each iteration in the algorithms. Let $track(i)$ be the disk track where the position i of the PAT block points to. We say that $track(i)$ "owns" position i. Let a *useful sector* be a sector that owns at least one position in the current PAT block. Let $useful(t)$ be the number of useful sectors in a track t. Let $newsize(t)$ be the expected size of the PAT block at the next iteration, after reading track t.

Given the high cost to access a track compared to the transfer time, once we are in position to read one track, we read it completely. Hence, by accessing one element in the array we are in position to perform more than one comparison at the same cost. The search is not binary anymore, but multi-way. When we read a track, we compare the query against *all* the elements pointing to that track, cutting the array in many *segments*. Only one segment qualifies for the next iteration.

Because we use a uniform model, the probability that a given segment is selected for the next iteration is proportional to its size. More formally, suppose the positions of a PAT block of size b are numbered $1..b$, and that positions $p_1, p_2, ...p_k$ of the PAT block point to track t (i.e. track t "owns" positions $p_1...p_k$). After reading track t we can compare the search key with the text strings and only one of the segments qualifies as the next subproblem. Figure 3 shows an instance of a partition of the PAT block containing 8 segments generated by the text strings of a track t.

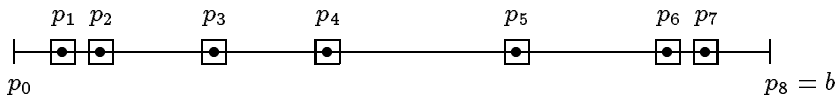

Figure 3: A PAT block partition generated by all text keys of a track t.

Thus

$$newsize(t) = \sum_{i=1}^{k+1} (Length\ of\ segment\ i) \times (Prob.\ of\ i\ being\ selected)$$

that is

$$newsize(t) = \sum_{i=1}^{k+1} \frac{(p_i - p_{i-1} - 1)^2}{b} \tag{3}$$

where $p_0 = 0$ and $p_{k+1} = b$.

6.3 The Optimal Algorithm

We need to adapt the optimal algorithm to this case, since the search is not binary anymore. A comparison may have more than two outputs. More specifically, for a given position i, if $track(i)$ owns k positions then there are $k+1$ possible outcomes after reading all the text on $track(i)$.

We define the costs as before, adapted to this problem and with more algorithmic detail. Observe that, except for the first and the last segments, the other segments must have $Lcost = Rcost$, since the same track owns their previous and next positions, and hence their costs are the same regardless of from where are we getting into them.

Thus, we fill two matrices:

- $\texttt{Lcost}[x, y]$ is the optimal time to solve the problem from positions x to y, when the disk head is on $track(x-1)$ (undefined if $x = 1$).
- $\texttt{Rcost}[x, y]$ is the optimal time to solve the problem from positions x to y, when the disk head is on $track(y+1)$ (undefined if $y = b$).

The matrices are filled diagonally for increasing values of $y - x$. Each matrix cell stores the minimum cost to solve the block $[x, y]$, starting either from the left $track(x-1)$ position or from the right $track(y+1)$ position of the block:

- Initially, $\texttt{Lcost}\ [x, x-1] = \texttt{Rcost}\ [x, x-1] = 0$.
- Then,

$$\begin{aligned}\texttt{Lcost}\ [x, y] = \min_{t=track(j), j \in x..y} \quad & (Access(track(x-1), t, useful(t)) + \\ & \frac{p_1 - p_0 - 1}{y - x + 1}\ \texttt{Rcost}\ [p_0 + 1, p_1 - 1] + \\ & \sum_{i=2}^{k+1} \frac{p_i - p_{i-1} - 1}{y - x + 1}\ \texttt{Lcost}\ [p_{i-1} + 1, p_i - 1])\end{aligned}$$

and the same for `Rcost`, replacing $x-1$ by $y+1$ inside $Access(...)$. Notice that, for $i \in 2..k$, `Lcost` = `Rcost` inside the sum.

In the top level, we may compute the optimal cost assuming we start at a specific track, or for all possible tracks. We must also store the track t selected at each position, to be able to build the optimum search tree. The time cost of this algorithm is $O(b^3)$, which makes it impractical for use at query time if b is large, since calculations could demand more time than the savings produced by the smart search strategy.

However, the scheme may be useful at indexing time. Although the matrix needs $O(b^2)$ space, the optimum search tree needs only $O(b)$ space, and it can be stored together with each block in the index. At query time, we just use the optimal precomputed tree to drive the search.

Stated that way, the scheme duplicates the space requirements of the index, but this can be greatly reduced by observing that the easiest part of the tree to compute is that needing more space: the leaves. That is, we do not store the leaves of the tree but recompute the needed leaves at query time. Leaves correspond to size-1 problems. This saves at least half of the space (it can be more, since the tree may not be binary).

In general, we can store all the tree except the last ℓ levels, then using at most $1/2^\ell$ additional space, and having to work $O((2^\ell)^3) = O(8^\ell)$ time and use $O(4^\ell)$ space to rebuild that part of the index at the needed point. For instance, we may pay for 1/32 (3%) additional index space and perform near 30,000 CPU operations on 1Kb space at index time, which is negligible. This is a very attractive trade-off to achieve optimality.

Next we present two alternatives to avoid the precomputation of the optimal search tree when this is not possible. We can apply a practical algorithm until obtaining a PAT block size small enough to be tractable with the $O(b^3)$ optimal strategy mentioned in the previous section. However, it has been found experimentally that this is not necessary, because all the algorithms behave quite similarly for small b.

6.4 The Approximate Algorithm

The general online approximate algorithm presented in Section 4.1 can be used in the more complex scheme with almost no changes. The corresponding formula is

$$Acost(i,j,h) = Access(h,t,useful(t)) + Ecost(newsize(t))$$

where $t = track(k)$ for $k \in i..j$ which minimizes $Access(h,t,useful(t))$.

Figure 4 presents the approximate algorithm. Observe that we can simply traverse the PAT block from left to right, keeping the cheapest track to access. This is $O(b)$ in the worst and average case. The total space requirement is $O(1)$.

For the access cost functions that we are considering, this algorithm behaves in a very simple way. If started on an extreme of the text, it will make a single sequential pass over the text, reading in its way any suffix which is still inside the current PAT block partition. As the search proceeds, less and less suffixes will be of interest.

The analysis is very similar to that of the general approximate algorithm of Section 4.1. However, we can improve the constants now. In our application, a track may own many positions, which makes it able to generate a new partition much smaller than $b/2$. For simplicity we assume in this analysis that a track owns just one position (which is pessimistic).

```
Search (bPAT, head)
  while (size of bPAT > 0)
  {
    compute S = set of useful tracks (which own a position of bPAT)
    t = s in S which minimizes Access(head,s,useful(s))
    move to t and read all keys from useful sectors
    bPAT = appropriate new partition (after search key comparison with keys read)
    head = t
  }
```

Figure 4: Online approximate algorithm.

In this application, there is no relationship between the cost of an element and its position in the array, i.e. the independence assumption holds. As explained, this algorithms optimizes the access cost with no regard to the goodness of the partition, and therefore its access pattern to the array is truly random. In our application we are interested in unsuccessful searches where any leaf is equally probable (i.e. $p(i) = 0, q(j) = 1/(n+1)$). Let $T(b)$ the total amount of work to perform on a block of size b, then $T(0) = 0$ and

$$T(b) = b + \frac{1}{b} \sum_{i=1}^{n} \left(\frac{i}{b+1} T(i-1) + \frac{b-i+1}{b+1} T(n-i) \right)$$

whose solution is $S(b) = 3b - 4H_{b+1} + 2 = O(b)$. That is, the constant is not 4 but 3 in our case. On the other hand, let $S(b)$ be the average number of iterations to perform for an array of b elements. Then $S(0) = 0$ and

$$S(b) = 1 + \frac{1}{b} \sum_{i=1}^{b} \left(\frac{i}{b+1} T(i-1) + \frac{b-i+1}{b+1} T(b-i) \right) \tag{4}$$

whose solution is $S(b) = 2(H_b - 1 + 1/(b+1)) = 2 \ln b + O(1)$, which is $O(\log b)$, and more precisely 39% over standard binary search (instead of our general result of 126% over standard binary search of Eq. (1)). This also proves that the average leaf depth in the optimal tree is $O(\log b)$, and that the average number of iterations of the approximate algorithm is $O(\log b)$. Moreover, there are at most 39% more iterations over standard binary search (on average).

6.5 The Heuristic Algorithm

The general online heuristic algorithm presented in Section 4.2 can also be used in the more complex scheme, although there are some complications. The corresponding formula is

$$Hcost(i,j,h) = \min_{t=track(k), k \in i..j} (Access(h,t,useful(t)) + Ecost(newsize(t)))$$

where $Ecost(b)$ is the average estimate of the cost of the algorithm for a PAT block of size b. The estimation can be done by storing times of previous runs or by analytical or experimental data about the performance of the algorithm. We show later simulation results to estimate $Ecost$. Figure 5 presents the heuristic algorithm.

Observe that we can traverse the PAT block from left to right, computing the set of useful tracks. At the same time we can compute the sum of squares of the segments of the partition

```
Search (bPAT, head)
  while (size of bPAT > 0)
  {
    compute S = set of useful tracks (which own a position of bPAT)
    compute newsize(s), for each s in S (recall Eq. (2))
    t = s in S which minimizes Access(head,s,useful(s)) + Ecost(newsize(s))
    move to t and read all keys from useful sectors
    bPAT = appropriate new partition (after search key comparison with keys read)
    head = t
  }
```

Figure 5: Online heuristic algorithm.

that each track produces in the PAT block using Eq. (3), since it determines the average size of the subproblem generated by that track (*newsize*). If the PAT block is traversed from left to right, it is easy to accumulate the sum of squares, by recording the previous node owned by each track, together with the current sum of squares. Therefore, both S and *newsize* can be computed in one pass.

We analyze this algorithm now. In the average case, this algorithm is $O(b)$ per iteration (note that b decreases at each step), since at most b tracks may be useful and they may be stored in a hash table to achieve constant search cost (when searching for a track in S). In the worst case, the algorithm is $O(b \log b)$ per iteration. The total space requirement is $O(b)$.

Since, as explained, this heuristic tries to balance the goodness of the partition with the access cost, we can pessimistically borrow the average results of the previous section. This shows that we work at most $3b = O(b)$ on average and perform at most $1.39 \log_2 n$ iterations. On the other hand, we work $O(b^2 \log b)$ in the worst case.

6.6 Analysis of Optimality

We resume in this section the general analysis of optimality of Section 5, this time focused on our particular search cost. We consider here a cost function which depends on the last element accessed. Our aim is to show that for this application

$$\frac{OptimalCost}{StandardBinaryCost} = \Omega\left(\frac{1}{\log b}\right)$$

from where it follows that, assuming $HeuristicCost \leq StandardBinaryCost$,

$$\frac{HeuristicCost}{OptimalCost} = O(\log b)$$

where we recall that b is the size of the array and n is the text size. We also prove that

$$\frac{ApproximateCost}{OptimalCost} \leq 1.39$$

which shows that our approximate algorithm cannot deliver a solution whose average cost is more than 39% over the optimal one. That is, our approximate algorithm is 1.39-optimal, where the optimality is measured over the average cost of the solutions (i.e. search trees) delivered.

For simplicity we explain the case of a binary search, since the case of multiway searching does not affect the order of the solution. We use a simplified access cost function, namely

$$w(i,j) \;=\; X \;+\; Y\;|\mathrm{PAT}[i] - \mathrm{PAT}[j]|$$

which in both disk models is correct for long seeks. The details for shorter seeks do not affect the order of the solution.

Imagine that we are at the beginning of the text on disk. On average, the text suffixes corresponding to the PAT block are uniformly spread in the text. The best that can happen at this point is that the cheapest element to access (the first suffix of this block) divides the array in two (i.e. $\mathrm{PAT}[b/2]$ is the first text suffix of this block). The best next thing that can happen is that the next two suffixes correspond to $\mathrm{PAT}[b/4]$ and $\mathrm{PAT}[3b/4]$, and so on. If all this happens, we complete any binary search with a single pass over the text on disk. Following this scheme, the leaves of the tree are in the second half of the text, and hence the unsuccessful search ends somewhere in that second half. Hence, this optimistic optimal search cost is

$$OptimalCost \;\geq\; MinOptimalCost \;=\; X\;\log_2 b \;+\; Y\;3n/4$$

while the standard binary search cost is

$$StandardBinaryCost \;=\; (X + Y\;n/3)\;\log_2 b$$

(since $n/3$ is the average distance between two random positions in the interval $(1,n)$). This shows that the competitive ratio $OptimalCost/StandardBinaryCost$ is $\Omega(1/\log b)$. That is, we cannot improve the binary search by a factor larger than $O(\log b)$. The smaller the PAT block, the closer the binary search to the optimal strategy.

We consider now the approximate algorithm. Since it ignores the goodness of the partition and looks only to the cheapest access cost, if we start at the beginning of the text it will pass sequentially over the text (never going back), reading any suffix that is inside the current partition. Since the array is partitioned at random positions in this process, the analysis of the previous section applies to the number of accesses (Eq (4)). Since we never go back in the text, we at most read it completely. Hence,

$$ApproximateCost \;\leq\; X \times 1.39 \log_2 b \;+\; Y\;n$$

which divided by $MinOptimalCost$ gives the upper bound

$$ApproximateCost \;\leq\; 1.39 \times MinOptimalCost \;\leq\; 1.39 \times OptimalCost$$

Since the strategy delivered by the approximate algorithm is possible, the optimal strategy cannot be worse. Hence, since $OptimalCost \leq ApproximateCost$, we have

$$MinOptimalCost \;\leq\; OptimalCost \;\leq\; 1.39 \times MinOptimalCost$$

It is interesting to compare this result against previous work. In [12], it is proven that the optimal search cost on an array where $w(i) = i^p$ is $O(n^p \log n)$. This is similar to our case when $p = 1$, although there is no advantage in accessing nearby positions. In this case, the optimal search cost is a constant fraction of the standard binary search cost.

On the other hand, in [1, 2] a different model is introduced, called "hierarchical model with block transfer", where once position j is accessed, the neighbor elements can be also accessed at low cost. This resembles (though it is not equal to) the disk cost model. Perhaps not surprisingly, they show that for $w(i) = i^\alpha$ for real $\alpha > 0$, the optimal binary search cost is $O(n^\alpha)$, which is $\Theta(1/\log n)$ times the cost of standard binary search. This corresponds to our competitive ratio (in our case $\alpha = 1$).

7 Simulation Results

We developed a simulation program to perform the actions of the optimal, approximate and heuristic algorithms. We did not perform real experiments because they depend on the scheduling algorithm for moving the disk arm and the placement algorithm for storing files in the disk. Moreover, normal operating systems do not give a time profile for each disk operation nor separate them from other internal tasks. Therefore, from real experiments it is not possible to extract, from the overall time, how much corresponds to each task when searching, and it is not possible to assume that the file is stored contiguously in the disk. In fact, the results of our simulation show that managing the disk as we propose for our particular application pays off, and this could make worthwhile to modify the operating system code for this application.

The simulator maps the text file on the disk sectors and tracks, either magnetic or optical, and computes the time needed to access and read any disk position. For a text with n index points and a PAT block with b elements, the simulator generates b random pointers in the range $1..n$. These pointers represent a set of random disk text positions which are stored in a table with b entries. The track number corresponding to each entry is also computed and stored in the table. By definition, all text index points associated to PAT array entries are in lexicographic order. We use this property to associate an integer (in ascending order from left to right) to each PAT block entry as a text representation. This approach has been validated experimentally in [5].

The parameters of interest in the simulation are: the text size, the PAT block size, b (usually ranging from 32 to 512 elements [4]), and the access time function for the disk and reading devices, either magnetic or optical, as shown in Section 6.1. We consider an average word length of 6 characters. We assume that all files are contiguously stored in the disk starting at track 1. In our experiments we used texts ranging from 256 megabytes to 1 gigabyte for magnetic disks and ranging from 64 megabytes to 256 megabytes for CD-ROM disks.

For each iteration of the heuristic algorithm the simulator scans the current PAT block and computes the sum of squares as described in Eq. (3). Then, a next track is selected according to the cost minimization criteria of the heuristic algorithm and a new partition in the current PAT block is obtained, until the search key is found.

To implement the heuristic algorithm it is necessary to have a definition for $Ecost(s)$ of Eq. (2), the expected cost to solve a problem of size s. The definition of this cost formula is important for the performance of the heuristic algorithm. We tested a number of different approach for the cost formula.

The simplest choice is to assume that the algorithm behaves as the naive binary search, where

$$Ecost(n) = StandardBinaryCost(n)$$

One approach that presented good results is the following: the selected track t is the one

that minimizes the sum of the access cost for track t from track h and the average size of the new partition problem, both normalized to the range $[0 \ldots 1]$, i.e.

$$\frac{Cost(h,t)}{MaxCost(h,t)} + \frac{NewSize(t)}{MaxNewSize(t)}$$

where $MaxCost(h,t)$ and $MaxNewSize(t)$ are the largest values for the two components in the current block. This approach to estimate $Ecost(s)$ presented results up to 15% better than the naive binary search approach.

An alternative approach to estimate $Ecost(s)$ is to record statistics on previous runs of the algorithm, using them to estimate the costs in the future. This scheme does not need any theoretical assumption, and can adapt to cost changes that may happen along time (e.g. the text file becomes more fragmented on disk).

We now describe the simulation of the approximate algorithm. For a text with n index points, a PAT block with b pointers is obtained using the same procedure given in the beginning of this section. The text positions (disk tracks) are sorted in ascending order in an auxiliary array, together with their original position (from $1 \ldots b$) in the PAT block, that is, the index of each text pointer in the original block. The search starts by reading the first text position in the sorted array. The text suffix pointed to by this position is compared with the search key and a new left or right bound for the next subproblem is found. Next, a new block partition is selected and the algorithm can now decide if a given text position in the current block is worth reading or not. During the next iterations, the algorithm proceeds by reading only the interesting text positions of the current block, until the key is found.

Table 1 presents the parameters used for magnetic and CD-ROM disks simulations, for both approximate and optimal algorithms [5, 16].

Parameters	Magnetic disk HP97560	Optical disk CD-ROM
Disk capacity (MB)	1,370	600
Sector length (bytes)	512	2,048
Track capacity (sectors)	72	9–21 (variable)
Number of heads (tracks/cylinder)	19	1
Number of cylinders	1,962	22,500
Transfer rate (MB/second)	2.4	1.2
Sector transfer time (S_{tt}, millisecs)	0.2	1.6
Average latency (t_r, millisecs)	7.5	61.0
Short seek overhead (C_{short}, millisecs)	3.24	1.0
Long seek overhead (C_{long}, t_0, millisecs)	8.00	160.0
Short seek boundary (D_{lim}, tracks)	383	20–40 (span size)
Short seek factor (θ_{short}, millisecs/track)	0.400	–
Long seek factor (θ_{long}, millisecs/track)	0.008	–
Short seek rate (α, millisecs/track)	–	1.0 (inside span)
Long seek rate (β, millisecs/track)	–	0.01 (outside span)

Table 1: Parameters used for magnetic and CD-ROM disks simulations.

As our model is based on unsuccessful search, we ran a set of 200 unsuccessful random searches for each text and PAT block size, both for optical and magnetic disks. For comparison

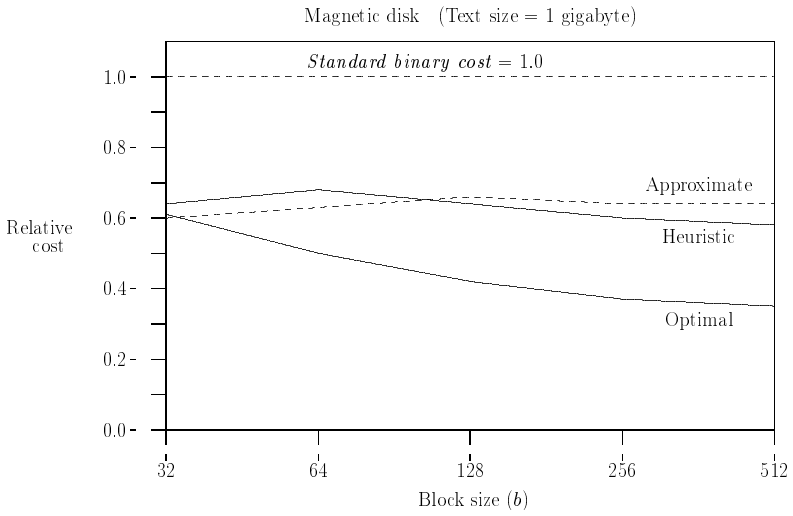

Figure 6: Relative cost for the optimal, approximate and heuristic algorithms for data stored in magnetic disks.

purposes, the same set of random pointers and search key for each simulation run were used by all the algorithms (optimal, heuristic, approximate and the standard binary search algorithm). The simulations were repeated for successful searches, for comparison purposes. We found that the analytical results for unsuccessful searches are very close to our simulation results, while the simulated successful searches are 10 to 15% faster than the unsuccessful ones. For example, using the magnetic disk specifications presented in Table 1, the analytical cost (unsuccessful) for a naive binary search on 512 megabytes of text with a block size of 256 pointers is 116 milliseconds. Our simulator gives 109 milliseconds for an unsuccessful search and 96 milliseconds for a successful search.

The results for the optimal, approximate and heuristic algorithms are shown in Figure 6 for magnetic disks, for 1 gigabyte text files. Figure 7 shows the results for CD-ROM disks, for 256 megabytes text files. The values in both plots represent the relative cost, having the standard binary search as reference ($StandardBinaryCost = 1$). The average values used in the plots for the approximate and heuristic algorithms are 95% confident within the interval of at most ± 0.16 for magnetic disks and at most ± 0.20 for CD-ROM disks.

In the following paragraphs we present some comments derived from the simulation results.

The simulation results have shown that each disk parameter in the cost function have different significance in the overall retrieval cost, depending on how much the file is spread on the disk tracks. Thus, a non-monotonic variation of the relative cost can be observed for the approximate and heuristic algorithms, depending on file size, file layout and disk geometry. Small files occupy few tracks in the disk and each track owns many positions of the PAT block, which makes the savings on latency larger than the savings on seek costs. Large files, distributed in many tracks on the disk, give more margin for savings on seek costs. We verified experimentally this conclusion, by cancelling separately the influence of the seek costs and latency costs in the simulator. The relative cost is monotonic on both file size and b when we consider only the latency cost (with seek cost null), and non-monotonic on file size and constant on b when we consider only seek cost (with latency cost null).

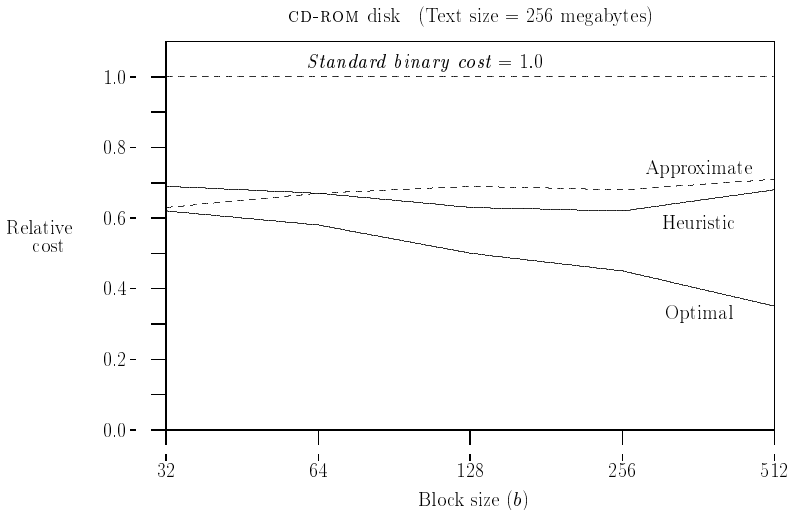

Figure 7: Relative cost for the optimal, approximate and heuristic algorithms for data stored in CD-ROM disks.

The experimental average head displacement, in tracks, using the standard binary search is $0.31T \leq AverageHeadDisp \leq 0.37T$ (recall that T is the number of tracks occupied by a text file). The same measure for the heuristic algorithm presents no significant difference for small files: for instance, a 10 megabytes file has an experimental average head displacement of $0.35T$. However, for large files the heuristic algorithm beats the standard binary search: for instance, for files larger than 250 megabytes we obtained an average head displacement smaller than $0.1T$. This result confirms that the savings on seek costs have more weight for larger files.

The simulations for the approximate algorithm confirmed that most of the savings rely on the reduction of the average seek distance. For this algorithm, the experiments have shown that the average number of seeks are very close to $O(\log_2 b)$, which is the expected cost of the standard binary search. We also observed that the non-linearity of the disk cost function imposes a penalty on the performance gain of both approximate and heuristic algorithm, due to: (i) in the magnetic disk, the convex function (square root for short seeks) makes the sum of a large number of short seeks larger than one long seek comprising the same space; (ii) in the CD-ROM disk, the double slope cost function exhibits a much higher slope for short seeks, producing a similar distance-cost trade-off problem. We run a set of simulations using a purely linear cost function and we obtained a much higher performance gain of both the approximate and heuristic algorithms. Thus, if the storage device exhibits linear cost function, the new practical algorithms present higher performance gains for similar searching problems.

The analytical upper bound $ApproximateCost/OptimalCost \leq 1.39$ could not be verified with the experiments, for large values of b, due to the distance-cost trade-off problem. However, it was verified when we used a linear cost function. As the optimal algorithm gives more weight to the goodness of the partition, its behavior is almost independent of the cost function.

Finally, the simulation results for the optimal algorithm show that a performance gain can be achieved over the approximate and heuristic algorithms. However, we emphasize that the optimal strategy cannot be directly used at search time because of its higher complexity

(although it can be used at indexing time). In practical terms, this means going from the range of milliseconds to several minutes in actual execution time. On the other hand, both algorithms perform quite similarly for small blocks. For example, for $b = 32$ and a text file of 1 gigabyte stored in a magnetic disk, the optimal strategy costs 75 milliseconds, while the approximated algorithm costs 79 milliseconds. Thus, it is not worth to switch from the practical to the optimal algorithm, when the value of b is small with large texts.

8 Conclusions

We addressed the problem of searching with different access costs, when they depend not only on the element to be accessed, but also on the last element visited. We have shown an optimal tree construction algorithm whose complexity is the same as the classical solution to the simpler problem of costs depending only on the element to be accessed. We also presented two practical online algorithms that compute only the part of the tree that is needed, at a much smaller cost. The algorithms perform quite well, and are suitable for practical cases in which a costly preprocessing cannot be afforded. We also present bounds on the competitive ratios.

We presented a case study to which the algorithms can be adapted, related to the different access costs of indirect search on disks. This application is a real problem brought from the text retrieval field. We described the cost model, enhanced the general algorithm to account for the real world added complications and presented simulation results regarding the performance gains over magnetic and optical disks. We also proved bounds on the competitive ratio of the optimal and practical algorithms tailored to this application.

The problem of searching with different access costs is quite general, and different complications appear for particular cases. Our treatment of the more complex case in which the cost depends on the last element visited is a step toward more general solutions. For example, the cost could be dependent on a longer previous set of accesses, or on the full history of the search. It is easy to solve the problem of dependence on k previous accesses in $O(n^{k+2})$ time, but good solutions to those problems are still to be devised. We believe that dependence on the full history is NP-Complete.

Acknowledgements

The authors would like to thank the anonymous referees for their valuable suggestions. We wish to acknowledge the helpful comments of Chris Perleberg and Jayme Swarczfiter.

References

[1] A. Aggarwal, B. Alpern, K. Chandra and M. Snir. A model for hierarchical memory. *Proceedings of the 19th Annual ACM Symposium of the Theory of Computing*, 1987, 305-314.

[2] A. Aggarwal, K. Chandra and M. Snir. Hierarchical memory with block transfer. *Proceedings of the 28th Annual Symposium on Foundations of Computer Science*, 1987, 204-216.

[3] M. Andrews, M. A. Bender and L. Zhang. New algorithms for the disk scheduling problem. *Proceedings of the 28th Annual ACM Symposium of the Theory of Computing*, 1996, 550–559.

[4] R. Baeza-Yates, E.F. Barbosa and N. Ziviani. Hierarchies of indices for text searching. *Information Systems 21* (6) (1996), 497–514.

[5] E. F. Barbosa. *Efficient text searching methods for secondary memory.* Ph.D. thesis, Technical Report 017/95, Department of Computer Science, Universidade Federal de Minas Gerais, Brazil, 1995.

[6] E. F. Barbosa, G. Navarro, R. Baeza-Yates, C. Perleberg and N. Ziviani. Optimized binary search and text retrieval. In Paul Spirakis, editor, *Proceedings of the 3rd Annual European Symposium on Algorithms (ESA'95)*, Springer-Verlag Lecture Notes in Computer Science, v. 979, 1995, 311-326.

[7] E. Gilbert and E. Moore. Variable length encodings. *Bell System Technical Journal 38* (4) (1959), 933–968.

[8] G. H. Gonnet. *Pat 3.1: an efficient text searching system.* Center for the New Oxford English Dictionary. University of Waterloo, Waterloo, Canada, 1987.

[9] G. H. Gonnet, R. Baeza-Yates and T. Snider. New indices for text: Pat trees and Pat arrays. In W. B. Frakes and R. Baeza-Yates, editors, *Information Retrieval Data Structures and Algorithms*, Prentice-Hall, Englewoods Cliff, N.J., 1992, 66–82.

[10] J. L. Hennesy and D. A. Patterson. *Computer Architecture. A Quantitative Approach.* Morgan Kaufmann Publishers, Inc., second edition, 1995.

[11] T. C. Hu and A. C. Tucker. Optimal computer-search trees and variable-length alphabetic codes. *SIAM Journal on Applied Math.*, 21 (1971), 514-532.

[12] W. J. Knight. Search in an ordered array having variable probe cost. *SIAM Journal on Computing* 17 (6) (1988), 1203-1214.

[13] D. E. Knuth. *The Art of Computer Programming: Sorting and Searching*, volume 3. Addison-Wesley, Reading, Massachusetts, 1973.

[14] U. Manber and G. Myers. Suffix arrays: a new method for online string searches. *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, 1990, 319–327.

[15] D. R. Morrison. PATRICIA - practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM 15* (4) (1968), 514–534.

[16] C. Ruemmler and J. Wilkes. An introduction to disk drive modeling. *IEEE Computer* 27 (3) (1994), 17–29.