# Optimized Binary Search and Text Retrieval[1]

Eduardo Fernandes Barbosa[2] Gonzalo Navarro[3]
Ricardo Baeza-Yates[3] Chris Perleberg[3] Nivio Ziviani[2]

**Abstract.** We present an algorithm that minimizes the expected cost of indirect binary search for data with non-constant access costs, such as disk data. Indirect binary search means that sorted access to the data is obtained through an array of pointers to the raw data. One immediate application of this algorithm is to improve the retrieval performance of disk databases that are indexed using the suffix array model (also called PAT array). We consider the cost model of magnetic and optical disks and the anticipated knowledge of the expected size of the subproblem produced by reading each disk track. This information is used to devise a modified binary searching algorithm to decrease overall retrieval costs. Both an optimal and a practical algorithm are presented, together with analytical and experimental results. For 100 megabytes of text the practical algorithm costs 60% of the standard binary search cost for the magnetic disk and 65% for the optical disk.

KEY-WORDS: Optimized binary search, text retrieval, PAT arrays, suffix arrays, magnetic disks, read-only optical disks, CD-ROM disks.

## 1 Introduction

To provide efficient information retrieval in large textual databases it is worthwhile to preprocess the database and build an index to decrease search time. One important type of index is the PAT array [Gon87, GBY91] or suffix array [MM90], which is a compact representation of a digital tree called a PAT tree, reducing space requirements by only storing the external nodes of the tree. A PAT tree or suffix tree [Knu73] is a Patricia tree [Mor68] built on the positions of interest in the text database. Each position of interest (or index point) is called a semi-infinite string or suffix, defined by its starting position and extending to the right as far as needed to guarantee uniqueness. In a PAT array the data

1 The authors wish to acknowledge the financial support from the Brazilian CNPq - Conselho Nacional de Desenvolvimento Científico e Tecnológico, Fondecyt Grant No. 1930765, IBM do Brasil, Programa de Cooperación Científica Chile-Brasil de Fundación Andes, and Project RITOS/CYTED. We also wish to acknowledge the fruitful suggestions from an anonymous referee.
2 Departamento de Ciência da Computação, Universidade Federal de Minas Gerais, Belo Horizonte, Brazil
3 Departamento de Ciencias de la Computación, Universidad de Chile, Santiago, Chile

access is provided through an indirect sorted array of pointers to the data. This array allows fast retrieval using an indirect binary search on the text.

When all data elements have constant access time, then binary search minimizes the number of accesses needed to search sorted data and also minimizes the total search time. However, some applications do not have data with constant access costs, including applications that have data distributed across a disk, where data near to the current disk head position costs less to access than data further away. For these applications we show that it is possible to improve the total search cost, which is equivalent to the number of data accesses multiplied by the average access cost.

The full inverted file is the most common type of index used in information retrieval systems. It is composed of a table and a list of occurrences, where an entry in the table consists of a word and a list of addresses in the text corresponding to that word. In general, inverted files need a storage overhead between 30% and 100%, depending on the data structure and the use of stopwords, and the search time is logarithmic. Similar space and time complexity can be achieved by PAT arrays. The great advantage of PAT arrays is its potential use in other kind of searches that are difficult or inefficient over inverted files. That is the case when searching for a long sequence of words, some types of boolean queries, regular expression searching, longest repetitions and most frequent searching [GBY91]. Consequently, PAT arrays should be considered seriously when designing a text searching method for text databases that are not updated frequently.

Due to the high non-constant retrieval costs inherent to the disk technology, a naive implementation of PAT arrays for large textual databases may result in a poor performance. A solution to improve the performance of text retrieval from disk was proposed in [BYBZ94], using two models of hierarchy of indices. They consist of a two-level hierarchy model that uses the main memory and one level of external storage (magnetic or optical devices) and a three-level hierarchy model that uses the main memory and two levels of external storage (magnetic and optical devices). Performance improvement is achieved in both models by storing most of higher index levels in faster memories, so that only a reduced interval of the PAT array is left on the disk, thus decreasing the number of accesses in the slowest device of the hierarchy.

The main goal of this paper is to present an algorithm that improves the expected retrieval time of the indirect binary search in an interval of a PAT array. For both magnetic and optical disks, we are interested in reducing the time complexity of the search in the last level, because optical disks have poor random access performance, more than a magnitude slower than magnetic disks, and magnetic disks are more than a magnitude slower than main memory. As magnetic and optical disks have non-uniform data access times depending on the current head position, our algorithm optimizes the total retrieval time, not the total number of disk accesses. The algorithm takes into account both the expected partition produced by reading each track and the cost of accessing that track.

## 2 Searching in a PAT Array on Disks

A PAT array is an array of pointers to suffixes, providing sorted accesses to all suffixes of interest in the text. With a PAT array it is possible to obtain all the occurrences of a string prefix in a text in logarithmic time using binary search. The binary search is indirect since it is necessary to access both the index and the text to compare a search key with a suffix. Figure 1 illustrates an example of a text database with nine index points and the corresponding PAT array.

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|
| 28 | 14 | 38 | 17 | 11 | 25 | 6 | 30 | 1 |

```
This text is an example of a textual database
↑    ↑    ↑  ↑  ↑        ↑  ↑ ↑       ↑
1    6    11 14 17       25 28 30     38
```

**Fig. 1.** PAT array or suffix array

To search for the prefix *tex* in the PAT array presented in Figure 1 we perform an indirect binary search over the array and obtain [7, 8] as the result (position 7 of the PAT array points to the suffix beginning at the position 6 in the text, and position 8 of the PAT array points to the suffix beginning at position 30 in the text). The size of the answer in this case is 2, which is the size of the array interval. Searching in a PAT array takes at most $2\,m \log n$ character comparisons and at most $4 \log n$ disk accesses, where $n$ is the number of indexing points and $m$ is the length of a given query. Building a PAT array is similar to sorting variable length records, at a cost of $O(n \log n)$ accesses on average. The extra space used by a PAT array is only one pointer per index point.

When the hierarchy model is used, the last phase of the search is performed by an indirect binary search in which a reduced interval of the PAT array is stored in main memory, as a PAT block with $b$ elements. The hierarchy model [BYBZ94] divides the PAT array into equal-sized blocks with $b$ elements and moves one element of each block to main memory together with a fixed amount of characters from the text related to the element selected from each block. Using this information, a preliminary binary search can be performed directly in the upper level. At the end of this search we know that the desired answer is inside the $b$ elements of the PAT block. This small PAT block is transferred to main memory and the exact answer is found through an indirect binary search between the PAT block in main memory and text suffixes in disk. This way, the cost is no more than $2 \log_2 b$ disk accesses. This is the part of the search that we intend to improve.

The entries of the PAT block are pointers to random positions in the text file stored on disk. Consequently, the sequence of disk positions to be visited during a binary search produces random displacements of the disk access mechanism.

Our strategy to reduce the overall binary search time looks for pivots that need little disk head movement and that bisect the PAT block as closely as possible. Each disk head movement produced by accessing a pivot changes the costs of accessing the remaining potential pivots, changing the problem with each search iteration. This problem seems closely related to a binary search over nodes with different access probabilities, for which an optimal binary search tree can be constructed by moving the nodes with higher access probability to the root[Knu73]. A solution for searching in a non-uniform access cost memory is also closely related[AACS87, Kni88]. However, these solutions do not directly apply, since in our problem the costs vary dynamically, depending on the current position of the disk reading mechanism.

The following definitions are used in the presentation of the optimal and the practical algorithms:

1. Let $b$ be the size of the current PAT block (a reduced interval of the PAT array initially transferred from disk to main memory). Note that $b$ is reduced at each iteration in the optimal and the practical algorithms.
2. Let $Cost(h, t, n_s)$ (*cost function*) be the time needed to read $n_s$ sectors from track $t$, with the reading mechanism being currently on track $h$. Thus

   $$Cost(h, t, n_s) = Seek(|h - t|) + Latency + n_s \times Transfer \qquad (1)$$

   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-ROMs, which rotate faster reading inner tracks than when reading outer tracks. In our analysis, we use an average CD-ROM latency. Transfer time is the time needed to transfer the desired data from the disk head to main memory.

   We ignore some details in this definition. For example, because we assume a constant latency for each access, reading two sectors in the same cylinder and rotational position (but different surfaces) of a multidisk drive will cost two constant latency times. It is clear that defining the access cost as function of sector is more accurate, however we have chosen to define it as a function of track (seek time plus a constant latency time) for simplicity.
3. Let an *useful sector* be a sector that contains a piece of text with at least one related pointer present in the current PAT block.
4. Let $size(t)$ be the number of useful sectors in a track $t$.
5. Let $newsize(t)$ (*reduction function*) be the expected number of useful sectors of the next iteration after reading track $t$.

   Assuming that the search for a pointer can occur anywhere in the PAT block with equal probability, then 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 track $t$ "owns" positions $p_1, p_2, ...p_k$ of the PAT block. Figure 2 shows an instantiation of a partition of the PAT block containing 8 segments generated by the text keys of a track $t$. After reading track $t$ we can compare the search key with the text keys and only one of the segments becomes the next subproblem.

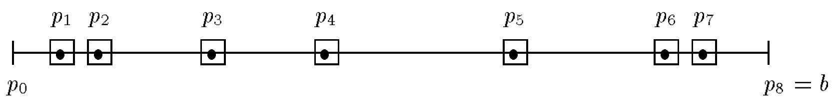


**Fig. 2.** A PAT block partition generated by all text keys of a track

Thus

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

that is

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

where $p_0 = 0$ and $p_{k+1} = b$.
Assuming that all positions have the same probability, if one only knows that a track owns $k$ positions without knowing which positions, then, the expected size of the next iteration (by counting over all possible values for $p_i$) is

$$newsize(k) \quad = \quad \frac{(b-k)(2b-k)}{b(k+2)} \quad \approx \quad \frac{2}{k+2}\, b \tag{3}$$

where the approximation holds for $k << b$. An important improvement that an optimal algorithm can make is to search for tracks owning two or more PAT block positions, since it would drastically reduce the new size at a negligible additional cost. The optimal and practical algorithms that we present take full advantage of this idea.

6. Let $C(h, 1..b)$ be the cost of the optimal algorithm when the reading mechanism is currently on track $h$ and $1..b$ is the current portion of the PAT block where we are searching.

## 3 An Optimal Algorithm

First, we present an optimal algorithm, using the definitions above, and show its impracticability. Second, using some simplifying assumptions, a feasible optimal algorithm is presented, still too costly for most uses, but useful for comparative purposes.

From Eq. (1) the track $t$ to be read next is the one that minimizes

$$Cost(h, t, size(t)) + C(t, from..to)$$

where $from..to$ is the selected segment from the comparisons made possible by reading track $t$.

Thus, the optimal algorithm satisfies the recurrence equation

$$C(h, 1..b) = min_t(Cost(h, t, size(t)) + C(t, from..to) \tag{4}$$

Unfortunately, it is not possible to know in advance which segment $from..to$ will be selected after reading $t$, so it is not possible to recursively compute the $C$ function of the right side of Eq. (4).

However, we can develop an algorithm that optimizes the *expected* cost by replacing $C$ with the *expected* $C$. In this case, our recurrence is converted to

$$C(h, 1..b) = min_t(Cost(h, t, size(t)) + \sum_{i=1}^{k+1} \frac{(p_i - p_{i-1} - 1)}{b} C(t, p_{i-1} + 1, p_i - 1))$$

It is then possible to take each candidate segment and recursively compute the cost of the algorithm provided this segment is selected and the search starts from $t$, and then sum up the costs for all segments weighted by the probability for each segment to be selected. With a naive implementation, this strategy requires $O(3^b)$ time. The time can be reduced to $O(b^4)$ (with $O(b^3)$ space) by using dynamic programming. The time cost makes it impractical for use in situations where $b$ is large, since calculations could demand more time than the savings produced by the smart search strategy. We do not include the pseudocode of this algorithm for lack of space.

We develop a simpler heuristic strategy by weakening the definition of *expected* $C$, interpreting it as if no information about the contents of the PAT block is available (of course we use it for the $Cost$ function in Eq. (1), but not for the $C$ of the right side). That means to use a $C$ averaged over all possible PAT array block contents, and to use some weighting strategy to favor those tracks whose neighborhood is "good", in the sense of owning a large number of positions of the current PAT block. More formally,

$$C(h, b) = min_t(Cost(h, t, size(t)) + weight(t) \times C'(newsize(t)))$$

where $C'(x)$ is an average estimate of the cost of the algorithm with a PAT block of size $x$ (see the next section).

In the next section we present a practical algorithm which follows this idea.

## 4 A Practical Algorithm

To design a practical algorithm from the general heuristic principle stated in the previous section, we need the following: a suitable weighting function, and an estimation of the average cost of the algorithm, which is part of the of the definition of the very same algorithm.

The simplest weighting function is a uniform one, that is $weight(t) = 1$ for all tracks. This is equivalent to not taking into account the neighborhood of tracks, but only their contents and distance from current position. We show that this simple strategy is quite close to optimal, so the effort of making a more complex analysis at each iteration is not worth doing. Figure 3 presents the practical algorithm.

```
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 Cost(head,s,size(s)) + C'(newsize(s))
    move to t and read all useful sectors
    bPAT = appropriate new partition (after search key comparison
                                      with keys read)
    head = t
  }
```

**Fig. 3.** Practical algorithm

Note that the better candidates for selection are those tracks that either are near to the current head position or generate a good partition of the PAT block. A track near to the current position may avoid an expensive seek cost and, on the other hand, a candidate track that owns many positions increases the probability of making the next partition (*newsize*) much smaller than $b/2$.

The next section presents an approximate analysis of this algorithm for magnetic and optical disks. By using these formulas to estimate the cost of smaller instances of the problem, we are able to complete the definition of the algorithm, thus eliminating its self-reference.

Note that it is possible to apply the practical algorithm until obtaining a PAT block size small enough to be tractable with the optimal strategy that uses the $O(b^4)$ time dynamic programming mentioned in the previous section. It has to be experimentally determined whether this improvement is worth doing for small sizes.

Observe that we can traverse the PAT block from left to right, and keep the set of useful tracks. At the same time we can compute the sum of squares of the segments of the partition that each track produces in the PAT block (recall Eq. (2)), since it determines the average size of the subproblem that that track generates (*newsize*). Note that if the PAT block is traversed from one side to another, 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. This way, both $S$ and *newsize* can be computed in one pass, that is, $O(b)$ time.

In the average case, this algorithm is O($b$) in each 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 $S$). Of course it is $\mathrm{O}(b \log b)$ in the worst case. The space requirement is $\mathrm{O}(b)$.

In the next section we show that this algorithm makes, on average, less than $\log_{\frac{1}{\omega}}(b+1)$ iterations, where $\frac{1}{2} \leq \omega < 1$ is the expected reduction in the size of the PAT block (i.e. the size of the PAT block at iteration $i$ is $bw^i$). Comparing this with classical binary search, we note that more iterations are required. The total average CPU cost is

$$b \left( \sum_{i=0}^{\log_{\frac{1}{\omega}}(b+1)-1} \omega^i \right) = \frac{b^2}{(b+1)(1-\omega)} \approx \frac{b}{1-\omega}$$

which is linear. The worst case occurs when the search for each candidate takes $\log b$, making each iteration $O(b \log b)$:

$$\sum_{i=0}^{\log_{\frac{1}{\omega}}(b+1)-1} b\omega^i \log_2(b\omega^i) \approx \frac{b}{1-\omega} \log_{\frac{1}{\omega}} b$$

# 5 Analytical Results

In this section we evaluate the practical algorithm, both for magnetic and optical disks. In each case, an analytical cost model is presented and analytical bounds for the algorithm under this cost model are obtained.

## 5.1 Magnetic Disks

The following definitions are used in the analysis of the practical algorithm for magnetic disks:

1. Let $C(b)$ be the the retrieval cost for magnetic disks when retrieving from a PAT block with $b$ elements.
2. Let $\sigma$ be the sum of the latency and transfer time (which really depends on the number of sectors to read) and let $\theta$ be the seek time per track.
3. Let $T$ be the number of tracks occupied by the text file and let $\Delta T$ be the distance that separates any two disk tracks.
4. Let $\delta$ be a small number of central positions in the current PAT block.

**A Seek Cost Model for Magnetic Disks** The cost function of magnetic disks may be modeled by a function of the form

$$f(\Delta T) = \sigma + \theta \Delta T$$

According to [HP90], typical disks have 500–2000 tracks by surface, each of them divided in 32 sectors. Sectors hold 512–2048 bytes. The typical value for latency is 8.3 ms, while transfer rates vary from 1 to 4MB per second. Average seek times range from 12 ms to 20 ms. Disks have from 1 to 20 plates, that is, 2 to 40 surfaces. The set of tracks from all surfaces which are at the same distance to the center is called a *cylinder*. For practical purposes, one can treat a disk with $k$ surfaces and 32 sectors as if it had only one surface, but whose tracks held $32 \times k$ sectors, with the same latency (8.3 ms). So the following discussion assumes only one surface, although the number of surfaces must be taken into account when calculating the number of tracks (cylinders) required by a file of a specified size.

Average seek time means the sum of all possible head displacements, divided by the number of possible displacements. This is

$$\frac{2}{T^2}\sum_{i=1}^{T}\sum_{j=1}^{i}(i-j) \quad = \quad \frac{T^2-1}{3T} \quad \approx \quad \frac{T}{3} \tag{5}$$

From the above discussion, we get the following values (in milliseconds):

$$\begin{aligned} \sigma &= 8.3 + n_s \times (0.125..2.0) \\ \theta &= 0.018..0.12 \end{aligned}$$

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 least cylinders as possible, so it should fill cylinders as completely as possible.

In many environments the sectors composing a file may be scattered on the disk. This obviously degrades the performance of any algorithm, although our algorithms are also optimal (in their own sense) under this situation. Another problem is that under different operating system policies, the cost model may vary. For example, some disk administrators do not serve requests that would make the disk head switch to the opposite direction of movement until the last request in the current direction is served. Under this scheme, those tracks that are following the current direction are much cheaper than the others. Both algorithms are able to handle all of these complications provided the cost function is appropriately defined. However, in the analysis we assume contiguous allocation and the simple cost model, which is optimistic if the file is scattered on the disk.

**Analysis of the Algorithm** It is useful to compare the performance of our practical algorithm with that of the standard binary search. The cost of each binary search step includes one seek, one rotational latency, and one transfer. Since the seek is random, we may use Eq. (5) to show that on average, 1/3 of the disk surface is traversed. The number of steps needed to complete the search is $\log_2(b+1)$, where $b$ is the initial PAT block size. Thus we have

$$\mathit{Binary\ Search\ Magnetic\ Cost}\ (b) \quad = \quad \left(\sigma + \frac{T}{3}\theta\right)\ \log_2(b+1) \tag{6}$$

Now we turn our attention to our algorithm. Since we are not able to analyze the real algorithm, we use a simplified model, whose predictions are to be experimentally tested against the real algorithm, to show its precision. It is important to note that this model is an *upper bound* for the expected case of the algorithm, so its predictions are always pessimistic.

The model is as follows. Suppose there are no tracks with more than one useful sector (this is worse than reality). At each step, we select the $\delta$ central positions of our PAT block, and read the nearest track which owns some of those central positions. The process continues until the PAT block size is $\leq \delta$. At this point, we traverse the disk from one end to the other, in one pass, reading any useful track, until the PAT block becomes empty. Since the real algorithm considers all (useful) tracks and selects the best one taking into account just seek cost and the generated partition, this model can never make a better decision than the real algorithm.

We first obtain the expected size of the new PAT block. This is (by using the same idea of Eq. (2))

$$\frac{1}{\delta b} \sum_{i=\frac{b-\delta}{2}}^{\frac{b+\delta}{2}-1} (i-1)^2 + (b-i)^2 \quad = \quad \frac{b}{2} - 1 + \frac{\delta^2 + 14}{6b} \quad \leq \quad \frac{b}{2} + \frac{\delta}{6} + \frac{4}{3}$$

and the bound is obtained by considering $b \geq \delta$. Note that the bound is of the form $Xb + Y$, with $X = 1/2$ and $Y = \delta/6 + 4/3$.

The next step is to obtain the average seek cost needed to access the nearest of the $\delta$ tracks, from a total of $T$. By summing over all possible positions at this track we can prove that the seek cost is approximately $T/2\delta$.

Thus, a bound of the cost for size $b$ (until obtaining a block of size $\leq \delta$) is

$$C(b) \quad = \quad \sigma + \theta \frac{T}{2\delta} + C(Xb + Y) \qquad\qquad (b > \delta)$$

By unfolding the right side of this recurrence, we get its closed expression

$$\begin{aligned} C(b) \quad &= \quad \left(\sigma + \theta \frac{T}{2\delta}\right) \log_2 \left(\frac{6}{3\delta - 8}\, b - \frac{3\delta + 16}{3\delta - 8}\right) \; + \; C(\delta) \\ &\leq \quad \left(\sigma + \theta \frac{T}{2\delta}\right) \log_2 \left(\frac{6}{3\delta - 8}\, b\right) \; + \; C(\delta) \end{aligned} \tag{7}$$

The value of $C(\delta)$ corresponds to solve the PAT block of size $\delta$, which consists of linearly traversing the disk surface and reading any useful track. On average, half of the $\delta$ tracks are read, and half of the surface is traversed. Thus,

$$C(\delta) \quad = \quad \sigma \frac{\delta}{2} + \theta \frac{T}{2}$$

which gives us the final cost expression

$$C(b) \quad \leq \quad \left(\sigma + \theta \frac{T}{2\delta}\right) \log_2 \left(\frac{6}{3\delta - 8}\, b\right) + \sigma \frac{\delta}{2} + \theta \frac{T}{2}$$

It is possible to prove that the optimal value for $\delta$ is:

$$\delta = \sqrt{\frac{\theta T}{\sigma}\log_2 b} + O(1)$$

For example, for a 160 megabytes file, $b = 1000$, $T = 5000$, $\sigma = 10.3$, $\theta = 0.045$, we have $\delta = 15$, and $C(b) = 320$ milliseconds, a 37% of the cost of the standard binary search (Eq. (6) gives 850 milliseconds). The result obtained by numerically finding the optimal $\delta$ differs by less than one. For an actual disk with a track capacity of 32 sectors and 4 recording surfaces, simulations for the practical algorithm yield 34% of the cost of the standard algorithm.

### 5.2 CD-ROM Disks

The following definitions are used in the analysis of the practical algorithm for CD-ROM disks:

1. Let $C(b)$ be the the retrieval cost for CD-ROM disks when retrieving from a PAT block with $b$ elements.
2. Let $c$ be the sum of the latency and transfer time by sector read and let $t_s$ be the seek time.
3. Let $T$ be the number of tracks occupied by the text file. Let $\Delta T$ be the distance that separates any two disk tracks.
4. Let $Q$ be the span size, the capability of accessing nearby tracks from the current position with no displacement of the reading mechanism.
5. Let $\alpha$ be the growing rate of the seek time as a function of the displacement of the access mechanism (in tracks) for short seeks.
6. Let $\beta$ be the growing rate of the seek time as a function of the displacement of the access mechanism (in tracks) for long seeks.

**A Seek Cost Model for CD-ROM Disks** The cost function of the CD-ROM drive is highly dependent on disk position and the amount of the displacement of the access mechanism. An important feature to be considered is the *span size* capability $Q$. In actual CD-ROM drives the span size is up to 60 tracks, depending on the type of the drive. 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 200 to 600 milliseconds.

The set of tracks covered by a span in a CD-ROM might be compared to the set of tracks belonging to a cylinder in a magnetic disk. In [BZ92] the set of tracks inside a span is considered as an *optical cylinder*. Thus, the data access in CD-ROM disks has two modes: (i) *proximal access*, for tracks inside the optical cylinder, and (ii) *non-proximal access*, for tracks outside the optical cylinder boundaries. These two modes are also known as *short seeks* and *long seeks*, respectively.

Other components of the access time to a given sector are the rotational latency and the transfer time from disk to main memory. The rotational latency is directly proportional to the position of the data on the disk, due to the constant linear velocity (CLV) physical format, costing from 65 milliseconds (inner track) to 153 milliseconds (outer track) to locate a sector. The transfer time is directly proportional to the amount of data transferred from disk to main memory, at the constant rate of 150 kilobytes per second (300 or 600 kilobytes per second in some drives). Any data in the CD-ROM is accessed by giving the physical address of the corresponding sector, and the sector size is always 2048 bytes. So, the sum of latency and transfer time by sector read is

$$c = (65..153) + 13$$

The seek time may be linearized, considering a slope between 0.02 and 0.04 milliseconds per track for non-proximal accesses and 1 millisecond per track for proximal accesses. The expression for $t_s$ is:

$$t_s = f(\Delta T) = \alpha \times \Delta T \quad for\ \Delta T \leq Q\ (proximal\ access)$$

$$t_s = f(\Delta T) = t_0 + \beta \Delta T \quad for\ \Delta T > Q\ (non-proximal\ access)$$

where $t_0$ represents the seek time for the first track outside the boundaries of the current optical cylinder, therefore requiring a seek. Some typical values for $\alpha$, $\beta$ and $Q$ are: $\alpha \approx 1$ millisecond/track, $0.02 \leq \beta \leq 0.04$ milliseconds/track, $200 \leq t_0 \leq 600$ milliseconds and $1 < Q \leq 60$ tracks.

We have also to consider that the optical head adjusts itself every time a new access is done, centering the anchor point on the track it has just moved to.

**Analysis of the Algorithm** We begin this section with the analysis of binary search on this cost model. Since the probability for a random track to be within the span size is negligible, we have

$$Binary\ Search\ Optical\ Cost\ (b) = \left(c + t_0 + \frac{T}{3}\beta\right)\ \log_2(b+1) \qquad (8)$$

We use a different model to approximately analyze the behavior of our algorithm on the optical cost model, since the one used for the magnetic case is far from optimal here. The idea is as follows: at any time, if there is a text key within the span size, we read it; else we read the track owning the middle position of the PAT block. Again, this model cannot perform better than our algorithm, on average, since we include both situations in the practical algorithm.

Assuming that we read any sector within the span size, this sector is at random, so using Eq. (3) with $k = 1$, the expected size of the new PAT block after reading that sector is bounded by $2/3\ b$, while of course the non-proximal access cuts the PAT block by half.

Since the disk head is in the middle of the span size, the expected cost for the proximal access is

$$A \quad = \quad c + \alpha \frac{Q}{4}$$

while for the non-proximal access, since the track to read is at random but surely outside the span, we have the expected cost

$$B = c + t_0 + \beta \left( \frac{Q}{2} + \frac{T-Q}{3} \right)$$

Finally, the probability for the nearest track to be outside of the span size is

$$\left(1 - \frac{Q}{T}\right)^b = \rho^b$$

where $A$, $B$ and $\rho$ are used as shorthands.

Then, the cost expression satisfies the following recurrence

$$C(b) = (1-\rho^b)\left(A + C\left(\frac{2}{3}\,b\right)\right) + \rho^b\left(B + C\left(\frac{1}{2}\,b\right)\right) \tag{9}$$

with the border condition $C(0) = 0$. Note that $A < B$.

Although this recurrence is hard to solve, it is possible to numerically compute any desired value. In order to provide a deeper insight on the complexity of this algorithm, we first prove a bound for $C(b)$ and then present an approximation, useful to compare the algorithm against the standard binary search. By using induction and bounding summations with integrals, we can prove that:

$$C(b) \leq A \log_{\frac{3}{2}} b + B + \left(B - A \log_{\frac{3}{2}} 2\right)\left(\rho + \log_{\frac{3}{2}} \frac{1}{1-\rho}\right)$$

In order to be able to compare with binary search, it is mandatory to find a tight value for the constant term, although it may not be a formal bound. The idea is to replace the sum of the costs for all traversed values of $b$ by an integral, thus we have to use a logarithmic scale. By using this technique, we can prove that an approximate solution for this recurrence is:

$$C(b) \approx \frac{c + \alpha\,\frac{Q}{4}}{\log_2 3 - 1} \log_2(b+1) + \left(c + t_0 + \beta\,\frac{T}{3} - \frac{c + \alpha\,\frac{Q}{4}}{\log_2 3 - 1}\right) \log_2 \frac{1}{1-\rho}$$

This final formula does give us a good understanding of the performance of the algorithm, and although it is not formally an upper bound, it is tight enough to extract percentages to compare it against the standard binary search. For example, for $b = 1000$, $T = 5000$ (equivalent to approximately 120 megabytes), $c = 125$, $\alpha = 1$, $\beta = 0.03$, $t_0 = 400$ and $Q = 50$, this final approximation gives us $C(b) = 4601$, 80.2% of the cost of the binary search (Eq. (8) gives 5726 milliseconds). By computing directly from the recurrence (9), we get $C(b) = 4491$ (78.4%). Simulations for the practical algorithm yield 70% of the cost of the standard binary search.

## 6 Experimental Results

We developed a simulation program to perform the actions of the practical algorithm. The simulator maps the text file on the disk sectors and tracks, either magnetic or optical, and computes the exact 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.

The parameters of interest in the simulation are: the text size (in our experiments we used texts ranging from 1 to 245 Megabytes), the PAT block size, $b$ (usually ranging from 256 to 2048 elements), and the access time function for the disk and reading device, either magnetic or optical. We consider an average word length of $\overline{W} = 6$ characters. Thus, given a text size with $M$ bytes, the number of index points of the corresponding PAT array is given by $n = M/\overline{W}$. We assume that all files are contiguously stored in the disk starting at track 1.

For each iteration of the practical algorithm the simulator scans the current PAT block and sort the tracks in ascending order, so that we can compute the sum of squares as described in Eq. (2). Then, a next track is selected according to the cost minimization criteria of the practical algorithm and a new partition in the current PAT block is obtained, until the search key is found. We run a set of 400 successful random searches for each text and PAT block size, both for optical and magnetic disks. For comparison purposes, the same set of random pointers and search key for each simulation run is used by the practical algorithm and by the standard binary search algorithm.

Table 1 presents the results for magnetic disks and Table 2 presents the results for CD-ROM disks. The values in both tables represent the gain (*PracticalCost / StandardBinaryCost*) in percentage terms (*StardardBinaryCost = 1*). All values are within 95% confidence interval.

$SectorLength = 512$ bytes; $TrackCapacity = 64$ sectors;
$Surfaces = 8$; $\sigma = 8.3ms$; $\theta = 0.045$; $(BinaryCost = 1.00)$;

| PAT Block Size (b) | Text 1.0MB | Text 15.4MB | Text 30.7MB | Text 61.4MB | Text 122.9MB | Text 245.8MB |
|---|---|---|---|---|---|---|
| 256 | 0.22±0.08 | 0.65±0.14 | 0.71±0.2 | 0.67±0.1 | 0.60±0.14 | 0.52±0.12 |
| 512 | 0.19±0.06 | 0.62±0.1 | 0.68±0.16 | 0.72±0.1 | 0.68±0.18 | 0.56±0.1 |
| 1024 | 0.18±0.04 | 0.56±0.16 | 0.65±0.1 | 0.70±0.18 | 0.65±0.12 | 0.55±0.08 |
| 2048 | 0.15±0.04 | 0.50±0.08 | 0.59±0.1 | 0.62±0.1 | 0.60±0.1 | 0.54±0.1 |

**Table 1.** Performance gain ($PracticalCost/BinaryCost$) for magnetic disks

$Q = 30; \quad \alpha = 1; \quad t_0 = 300; \quad \beta = 0.03; \quad (BinaryCost = 1.00);$

| PAT Block Size (b) | Text 1.0MB | Text 15.4MB | Text 30.7MB | Text 61.4MB | Text 122.9MB | Text 245.8MB |
|---|---|---|---|---|---|---|
| 256 | 0.48±0.1 | 0.51±0.1 | 0.60±0.16 | 0.72±0.15 | 0.78±0.14 | 0.80±0.2 |
| 512 | 0.41±0.14 | 0.50±0.12 | 0.58±0.12 | 0.70±0.2 | 0.78±0.16 | 0.78±0.14 |
| 1024 | 0.35±0.1 | 0.44±0.1 | 0.55±0.12 | 0.65±0.18 | 0.70±0.1 | 0.76±0.14 |
| 2048 | 0.32±0.08 | 0.42±0.06 | 0.50±0.08 | 0.61±0.12 | 0.63±0.08 | 0.70±0.14 |

**Table 2.** Performance gain ($PracticalCost/BinaryCost$) for CD-ROM disks

Some comments can be derived from the experimental results, as follows:

1. We observe that this process presents an intrinsically large variance. However, this is not a restriction since the probability of the standard binary search to perform better than the practical algorithm is very small. For example, if the desired answer is exactly at the root of the binary search process, then the standard algorithm is faster than ours. However, considering the values of $b$ we are using, the probability of the answer to be at the root level of a subtree corresponding to a PAT block is very small. We observed that the practical algorithm performs better than the standard binary search in more than 95% of the cases, for all the parameters used in our experiments.
2. The results for magnetic disks have shown a non-monotonic variation of the gain. We found that the disk parameters in the cost function have different weights in the overall retrieval cost, depending on how much the file is spread on the disk tracks. 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, gives 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 gain 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).
3. The experimental average head displacement, in tracks, using the standard binary search is $0.31T \leq AverageHeadDisp \leq 0.37T$, which matches quite closely the assumption of $T/3$ used in our analysis. The same measure for the practical algorithm presents no significant difference for small files: for instance, a 1 megabyte file has an experimental average head displacement of $0.35T$. However, for large files the practical algorithm beats the standard binary search: for instance, for files ranging from 100 to 245 megabytes we obtained an average head displacement of only $0.1T$. This result confirms that the savings on seek costs have more weight for larger files.

4. Finally, we compared the cost reduction in terms of the time needed to search for a given key. For example, a text file of 30.7 megabytes, stored in a magnetic disk, with a PAT block of 1024 elements, has an average cost of 100 milliseconds using the standard binary search and 60 milliseconds using the practical algorithm. The same file stored in a CD-ROM disk has an average cost of 3.5 seconds using the stardard binary search and 1.8 seconds using the practical algorithm.

## References


[AACS87] A. Aggarwal, B. Alpern, K. Chandra and M. Snir. "A Model for Hierarchical Memory", *Proc. of the 19th Annual ACM Symp. of the Theory of Computing*, New York, 1987, 305-314.

[BYBZ94] R. Baeza-Yates, E.F. Barbosa and N. Ziviani. Hierarchies of indices for text searching. In *Proceedings RIAO'94 Intelligent Multimedia Information Retrieval Systems and Management*, pages 11–13. Rockefeller University, New York, Oct. 1994.

[BZ92] E. F. Barbosa and N. Ziviani. Data structures and access methods for read-only optical disks. In R. Baeza-Yates and U. Manber, editors, *Computer Science: Research and Applications*, pages 189–207. Plenum Publishing Corp., 1992.

[GBY91] G. H. Gonnet and R. Baeza-Yates. *Handbook of Algorithms and Data Structures*. Addison-Wesley, 1991.

[Gon87] G. H. Gonnet. PAT *3.1: An Efficient Text Searching System.* User's Manual. Center for the New Oxford English Dictionary. University of Waterloo, Waterloo, Canada, 1987.

[HP90] J. L. Hennesy and D. A. Patterson. *Computer Architecture. A Quantitative Approach.* Morgan Kaufmann Publishers, Inc., 1990.

[Kni88] W.J. Knight. Search in an Ordered Array having Variable Probe Cost. *SIAM J. of Computing* 17 (6), Dec. 1988, 1203-1214.

[Knu73] D.E. Knuth. *The Art of Computer Programming: Sorting and Searching*, volume 3. Addison-Wesley, Reading, Massachusetts, 1973.

[MM90] U. Manber and G. Myers. Suffix Arrays: A new method for on-line string searches. *ACM-SIAM Symposium on Discrete Algorithms*, pages 319–327, Jan. 1990.

[Mor68] D. R. Morrison. PATRICIA - Practical Algorithm To Retrieve Information Coded in Alphanumeric. *Journal of the ACM*, 15(4):514–534, 1968.



This article was processed using the LaTeX macro package with LLNCS style