Optimized Indirect Binary Search and Text Retrieval

(Preliminary Version)

Gonzalo Navarro (1)
Eduardo Fernandez Barbosa (2)
Chris Perleberg (1)
Ricardo Baeza-Yates (1)
Nivio Ziviani (2)

(1) Depto. de Ciencias de la Computación,
Universidad de Chile
Santiago, Chile
(2) Depto. de Ciencia da Computacao
Univ. Fed. de Minas Gerais,
Belo Horizonte, Brazil

Abstract

We present an algorithm that minimizes the expected cost of indirect binary search for data with non-constant access costs (e.g. disk data). The term "indirect" indicates 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 reduce overall retrieval costs. Both an optimal and a practical algorithm are presented, together with analytical and some experimental results, showing that standard binary search can be improved by 50% or more.

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

1 Introduction

In many applications, including text retrieval, sorted access to data is provided through an array of pointers to the data. This "indirect" array of pointers allows fast retrieval

using a binary search. Binary search minimizes the number of accesses needed to search sorted data, and if all accesses to the data have constant cost, minimizes the search time. However, many applications do not have constant data access costs, especially those applications that have data distributed across a disk, where data near to the disk head costs less to access than data farther away. In these applications it is necessary to minimize the total cost of the search, which is equivalent to the number of data accesses multiplied by the average access cost.

A new and simple type of index for text is the PAT array [2], or suffix array [4]. A PAT array achieves a compact representation of a digital tree called PAT tree, by storing only the external nodes of the tree. A PAT tree is a Patricia tree [5] built on all indexing points of a text database.

The suffix or PAT array defines each indexing point of the text as the string that begins at that text position and extends arbitrarily to the right. The string is called a semi-infinite string or a *sistring*. A PAT array is an array of pointers to the text, providing sorted access to all sistrings in the text. With a PAT array, all occurrences of a string prefix can be found in a text with n index points in $O(\log n)$ time, using binary search. This binary search is indirect since it is necessary to access both the index and the text to compare a key against a sistring.

Building a PAT array is similar to sorting variable length records, at a cost of $O(n \log n)$. Searching in a PAT array takes at most $2\, m \log_2 n$ character comparisons and at most $4 \log_2 n$ disk accesses, where m is the length of a given query. The extra space used by a PAT array is only one pointer per indexing point.

Due to the high retrieval costs inherent to disk technology, a naive implementation of PAT array indices for a large textual database in this medium may result in very poor performance. A solution to improve the performance of text retrieval from disk was proposed in [6], using an indices hierarchy model. This model consists of distributing the index structure in the memory hierarchy, using up to three storage devices with different cost-performance ratios: the CD-ROM optical disk (optionally), the magnetic disk and the main memory. The basic idea is to store most of the index structure in faster devices, so that fewer accesses are needed in the slower devices.

Still further improvements in the search time are needed. 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 lower than magnetic disks, and magnetic disks are more than a magnitude slower than main memory. We therefore use the algorithm presented in this paper to optimize the total cost of searches, not just the total number of accesses to disks. The algorithm takes into account both the expected partition produced by reading each track and the cost of accessing that track.

In Section 2 we show why binary search does not minimize costs when disk access times are considered, and we model this situation in order to characterize both the cost and the reduction of the problem size obtained by accessing a given track. In Section 3 an optimal (on average) algorithm is presented, which follows from the model in a straightforward way. Section 4 presents a practical algorithm, and in Section 5 its performance is analyzed and compared against that of the optimal algorithm.

2 Searching in a PAT Array on Disks

When the hierarchy model is used [6], the last phase of the search is performed by an indirect binary search in which a subset of the PAT array is stored in main memory, as a PAT block with b elements. Each entry of the PAT array is a pointer to the text file stored on disk, so in order to make each comparison, the position of the text file pointed to by the PAT must be read into main memory. Due to the nature of the text, the pointer addresses in the PAT array block point to random locations in the text. Thus, the sequence of disk positions to be visited during a binary search is also random, producing random displacements of the disk access mechanism.

The time cost of the binary search can be reduced if we chose pivots that need little disk head movement and that bisect the remaining pointers as closely as possible. Each head movement produced by accessing a pivot changes the costs of accessing remaining potential pivots, so the problem changes with each iteration of the binary search.

This problem is related to binary search when the nodes have different access probabilities. In that case, an optimal binary search tree is constructed by moving to the root those nodes that have higher probability. However, this solution cannot be applied to our case, mainly because our costs are dynamically varying, since they depend on the current head position of the reading device.

In order to present an optimal algorithm for this problem, we begin by defining the cost function for accessing a track and the reduction of the problem it produces.

2.1 Cost function

The cost function is the time needed to access a given track, that is, to read all useful sectors in it.

A cost function for disks has three components: *seek time*, *latency*, 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 (or rotation time) is the expected time spent waiting for the desired sector to pass under the reading head. The average latency is constant for a magnetic disk 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.

Although these costs vary from one disk to another, and especially between magnetic and optical disks, we can state the following *cost function* for reading n_s sectors from track t, with the reading head currently on track c:

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

We ignore in this definition some details, for example if one has to read two sectors from a multiplate disk which are in the same cylinder and rotational position but in different surfaces, that cannot be done without waiting two latencies.

Note that although it is possible to define this cost as a function of sector rather than track, we have chosen the above definition for simplicity. Thus, by "reading a track" it is meant to read all useful sectors in it.

2.2 Reduction function

Given a track t, it is possible to determine which of its sectors are useful, namely those which contain some text position present in our current PAT block. The number of useful sectors is called $size(t)$. We ignore details regarding the position to read inside the sector, which could be near the right end, making it necessary to read also the next sector to make a string comparison with the query.

We also know which elements in the PAT block could be compared if we had all useful sectors of t in memory. Those elements cut the PAT block in segments (see Figure 1). Once we have those sectors available for string comparison, only one of those segments would qualify for the next iteration. Since we are searching for an interval in the PAT array in which a string prefix occurs, we just search for one edge of the interval, which divides the PAT block into a portion that belongs to the interval and a portion which does not.

If we assume that the searched point 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 t "owns" positions $p_1, p_2, ...p_k$ of the PAT block.

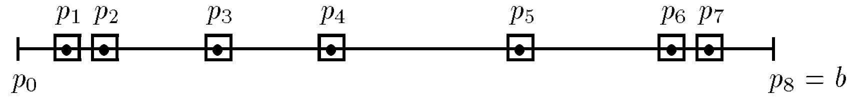

Figure 1: Partition of the PAT block generated by reading all keys on a track. After reading the track, only one of the segments will become the next subproblem.

Then, the *expected size* after reading that track is

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

that is

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

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

It can be shown by counting over all possible values for p_i (all positions have the same probability) that if one only knows that a track owns k positions, without knowing which positions, the expected size of the next iteration is

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

the approximation holds when $k << b$. 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.

3 An Optimal Algorithm

Here we present an optimal algorithm, using the definitions above, and show its impracticability. Then, using some simplifying assumptions, a feasible optimal algorithm is presented, still too complex for most uses, but useful for comparitive purposes.

Let's call $C(h, 1..b)$ the cost of the optimal algorithm when the current track is h and $1..b$ is the portion of the PAT block where we are searching. Then, the track t to be read next is that which minimizes

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

where $from(t)..to(t)$ is the selected segment after the comparisons made posible by reading track t.

Then, the optimal algorithm satisfies the recurrence equation

$$C(h, 1..b) = min_t(Cost(h, t, size(t)) + C(t, from(t)..to(t)))$$

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

However, we can develop an algorithm that optimizes the *expected* cost by replacing C with the *expected* C. 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; it can be reduced to $O(b^4)$ time and $O(b^3)$ space using dynamic programming. These CPU costs make it impractible for use in situations where b is large, since CPU calculations could demand more time than the savings produced by the smart search strategy.

We present here the naive algorithm, which is invoked as `MinCost(head,1,b)`, where `head` is the actual head position and `b` is the size of the PAT block. The function returns the optimal cost, while the track which must be read is the `s` which achieves this cost.

```
bPAT[1..b] = pat block where to search

AveCost (t,from,to)
      /* returns the average optimal cost to solve bPAT[from..to]
         if the search starts by reading track t */
   { P = set of segments that t produces on bPAT (recall section 2.2)
     return SUM (MinCost (t,p.from,p.to) * p.length / (to-from+1),
                 summing over all p in P)
   }
```

```
MinCost (head,from,to)
      /* returns the optimal cost to solve bPAT[from..to] if
         the disk head is currently on position 'head' */
   { S = set of useful tracks
                    (those which own some position of bPAT[from..to])
     return MIN (Cost (head,s,size(s)) + AveCost (s,from,to),
                  minimizing over all s in S)
   }
```

The dynamic programming algorithm stores previously computed values of `AveCost`, in a matrix `M[t,from,to]`, which can be computed for increasing values of `(to-from)`, to deliver the final answer $min_{\mathtt{t}}$ `M[t,1,b]`. That matrix needs to be computed only for the first iteration.

We develop a simpler heuristic strategy by weakening the definition of *expected* C, interpreting it as if one had no information about the contents of the PAT block (of course we use it for the $Cost$ function, but not for the C of the right side). That means to use simply a C averaged over all possible PAT 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.

In the next section we present an algorithm which follows these lines.

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 not having a weighting function at all, 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 later 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.

The next section presents an approximate analysis of this algorithm. 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 small enough to be tractable with the optimal strategy. It has to be experimentally determined whether this improvement is worth doing for small sizes.

The practical algorithm follows.

```
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 Equation (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 key comparison with nodes read)
       head = t
     }
```

Observe that we can traverse the PAT block from left to right, and keep the set of useful tracks (those which own some PAT block node). At the same time we can compute (recall Equation (1)) the sum of squares of the segments of the partition that each track produces in the PAT block, since it determines the average size of the subproblem 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.

Although the most important complexity measure is the total disk access cost, it is worth discussing the CPU cost of this algorithm, since although it is not the central point, it makes impractical other algorithms more close to the optimal in disk access cost because of their CPU intractability.

In the average case, this algorithm is $\mathrm{O}(b)$ at each iteration (note 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 CPU cost is

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

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

$$\begin{aligned} \sum_{i=0}^{\log_{\frac{1}{\omega}}(b+1)-1} b\omega^i \log_2(b\omega^i) &\approx \frac{b\left((1-\omega) b \log_2 b + \omega \log_2\left(\frac{1}{\omega}\right) b + (1-\omega)\log_2(b+1)\right)}{(b+1)(1-\omega)^2} \\ &\approx \frac{1}{1-\omega} b \log_{\frac{1}{\omega}} b \end{aligned}$$

Note that the better candidates for selection are those tracks that either are near to the current position, or those that generate a good partition of the PAT block, especially if they own more than one position of the PAT block.

5 Evaluation

In this section we evaluate the proposed 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. Finally, some experimental results comparing the optimal and the practical algorithms are presented.

5.1 Magnetic disks

5.1.1 A seek cost model for magnetic disks

The cost function of magnetic disks may be modeled by a function of the form

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

where σ is the sum of the latency and transfer time (which in fact depends on the number of sectors to read), and θ is the seek time per track.

According to [3], 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 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

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

where T is the total number of tracks (later, we make this definition more precise).

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

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

It is clear that 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.

It is important to note that in many environments, the sectors composing a file may be scattered on the disk. This will obviously degrade 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 to 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.

5.1.2 Analysis of the algorithm

Since a useful comparison value for the algorithm is how much better it is in relation with binary search, we begin by giving the binary search cost. The cost of each binary search step includes one seek, one rotational latency, and one transfer. Since the seek is random, we may use Equation (3) to show that on average, 1/3 of the disk surface is traversed (we denote by T the total number of tracks used up by the file, assuming contiguous allocation). The number of steps needed to to complete the search is $\log_2(b+1)$, where b is the initial PAT block size. Thus we have

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

Now we turn out 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 δ central positions of our PAT block, and read the nearest track which own some of those central positions. The process continues until our 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 (including the δ central ones) 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 Equation (1))

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

and the bound is obtained by considering $b \geq \delta$. Note the bound is of the form $Xb+Y$.

The next step is to obtain the average seek needed to access the nearest of any of the δ tracks, from a total of T. Let's call $\gamma = 1 - \frac{\delta}{T}$ the probability of a track not owning a value from the central δ. By symmetry, we analyze only the left half of the array of tracks. Suppose the disk head is in a given position, call it N (N is in the range $1..T/2$). Let's call M the minimum movement needed to reach a useful track. Then, for i in the range $0..N-1$, the probability to move more than i tracks is γ^{2i}, since the

move may be in any direction. In the range $N..(T-N)$, only the right direction is possible, thus the probability is γ^i.

Then, given the disk head is in position N, the expected movement is

$$\begin{aligned} E_N(M) &= \sum_{i=1}^{T-N} iP(M=i) = \sum_{i=0}^{T-N-1} P(M>i) \\ &= \sum_{i=0}^{N-1} \gamma^{2i} + \sum_{i=N}^{T-N-1} \gamma^i = \frac{1-\gamma^{2N}}{1-\gamma^2} + \frac{\gamma^N - \gamma^{T-N}}{1-\gamma} \end{aligned}$$

By summing up all (equally probable) values for N

$$E(M) = \frac{1}{T/2} \sum_{N=1}^{T/2} \frac{1-\gamma^{2N}}{1-\gamma^2} + \frac{\gamma^N - \gamma^{T-N}}{1-\gamma} \approx T \, \frac{\delta - 8\,e^{-\frac{\delta}{2}} + 5\,e^{-\delta} + 3}{2\,\delta^2} \approx \frac{T}{2\delta}$$

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

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

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

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

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

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

which gives us the final cost expression

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

The optimum value for δ depends on the rest of the parameters, and has to be numerically determined for each case. For example, for $b = 1000$, $T = 5000$, $\sigma = 10.3$, $\theta = 0.045$, we have $\delta = 16$, and $C(b) = 320$, a 37% of the standard binary search (simulations yield 33% for the real algorithm).

5.2 CD-ROM Disks

5.2.1 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 that the CD-ROM drives exhibit the capability of accessing nearby tracks from the current position with no displacement of the reading mechanism, with a negligible seek time. This feature is called *span access* and the number of tracks that can be accessed in this way is called *span size*. 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 [1] 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 delay (or latency) and the transfer time from disk to the main memory. The 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 the main memory, at the constant rate of 150 kilobytes per second (300 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. We call c the sum of latency and transfer time by sector read, so we have

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

The seek time (t_s), is a function of the distance that separates the current track and the next, $\Delta T = |T_i - T_j|$, where T_i and T_j are any two disk tracks. This function 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 Q is the span size, α and β represent the growing rates of the seek time as a function of the displacement of the access mechanism (in tracks) for short and long seeks, respectively, and 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 α, β and Q are: $\alpha \approx 1\ ms/\text{track}$, $0.02 \leq \beta \leq 0.04\ ms/\text{track}$, $200\ ms \leq t_0 \leq 600\ ms$ and $1 < Q \leq 60$ tracks.

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

5.2.2 Analysis of the algorithm

We begin this section again with an 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

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

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 node 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 these nodes in our considerations.

Assuming that we read any node within the span size, this node is at random, so using Equation (2) with $k = 1$, the expected size of the new PAT block after reading that node is bound by $\frac{2}{3}b$, 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

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

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 out of b to be without the span size is

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

We use A, B and γ just as shorthands.

Then, the cost expression satisfies the following recurrence

$$C(b) \;=\; (1-\gamma^b)\left(A + C\left(\frac{2}{3}\,b\right)\right) + \gamma^b\left(B + C\left(\frac{1}{2}\,b\right)\right) \qquad (4)$$

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

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 and then present an approximation, useful to compare the algorithm against binary search.

We begin by writing

$$C(b) \;\le\; A\log_{\frac{3}{2}} b + D(b)$$

and using induction it follows that a sufficient condition for D is

$$D(b) \;\ge\; D\left(\frac{2}{3}\,b\right) + \gamma^b\left(B - A\log_{\frac{3}{2}} 2\right)$$

with the border condition $D(1) = B$. By unfolding that leads to

$$D(b) \quad \geq \quad \left(B - A\log_{\frac{3}{2}} 2\right) \sum_{i=0}^{\log_{\frac{3}{2}} b} \gamma^{(\frac{3}{2})^i} + B$$

the summation can be bounded by extending to infinite and using an integral

$$\sum_{i=0}^{\infty} \gamma^{(\frac{3}{2})^i} \;\leq\; \gamma + \int_0^\infty \gamma^{(\frac{3}{2})^x} dx \;=\; \gamma + \frac{1}{\ln\frac{3}{2}} \int_1^\infty \frac{\gamma^u}{u} du \;\leq\; \gamma + \frac{1}{\ln\frac{3}{2}} \sum_{n=1}^{\infty} \frac{\gamma^n}{n} \;=\; \gamma + \log_{\frac{3}{2}} \frac{1}{1-\gamma}$$

finally, we get

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

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 be not 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. The problem is that there are two logarithms involved here: base 2 and base $\frac{3}{2}$. We handle this situation by using a stretched 0..1 scale, that will be expanded differently for each type of seek. We simulate the process of summing up costs while partitioning b by the process of integrating the average cost over the logarithmic stretched 0..1 interval. The probability function we use is

$$f(x) = \gamma^{b^{1-x}}$$

if $x = 0$, $f(x)$ is the probability of a track not within the span size at the beginning of the process; while at $x = 1$ it represents the end of the process. While x advances linearly, the exponent of γ decreases exponentially, so a linear sum (integration) in the range $x = 0..1$ of the costs weighted by $f(x)$ and advancing with the corresponding differential effectively simulates the logarithmic process. Finally, since this is a 0..1 scale, we have to multiply the final result by the number of steps for the corresponding seek. This is

$$\int_0^1 (1 - f(x))A dx_1 + f(x) B dx_2$$

by expanding the stretched intervals for dx_1 and dx_2, we have

$$C(b) \;=\; \log_{\frac{3}{2}}(b+1)\; A\; \left(1 - \int_0^1 f(x)dx\right) \;+\; \log_2(b+1)\; B\; \int_0^1 f(x)dx \qquad (5)$$

Although $f(x)$ has no primitive, the definite integral may be numerically computed for each b and γ, but again in order to get some idea of the involved values, we approximate it:

$$\int_0^1 \gamma^{b^{1-x}} dx \quad = \quad \frac{1}{\log b} \int_1^b \frac{\gamma^u}{u} du \quad \leq \quad \frac{1}{\log b} \sum_{n=1}^{\infty} \frac{\gamma^n}{n} \quad = \quad \frac{\log\frac{1}{1-\gamma}}{\log b}$$

Finally, we obtain the *approximate* cost of this algorithm:

$$C(b) \approx \left(c + \alpha\, \frac{Q}{4}\right) \log_{\frac{3}{2}} \left((b+1)(1-\gamma)\right) \; + \; \left(c + t_0 + \beta\, \frac{T}{3}\right) \log_2 \frac{1}{1-\gamma}$$

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 binary search. For example, for $b = 1000$, $T = 5000$, $c = 125$, $\alpha = 1$, $\beta = 0.03$, $t_0 = 400$ and $Q = 50$, this final approximation gives us $C(b)$ = 4601, 80.2% of binary search, while numerically integrating the expression (5) we obtain $C(b) = 4321$ (75.4%). By computing directly from the recurrence (4), we get $C(b) = 4491$ (78.4%). Simulations yield 66% for the real algorithm.

5.3 Experimental results

In order to determine how close the practical and the optimal algorithms are, simulations must be carried out. Although the simulators are finished, experiments are still ongoing. The initial results indicate savings of 60% in magnetic disks, and 35% in optical disks.

Without being still able to draw statistical results (especially since this process has a large variance), we found that most of the time the practical and the optimal algorithm make exactly the same decisions. In the few cases in which they take different paths (a bit more common with optical disks), the optimal beats the practical by 5-10% in the magnetic case, and 10-20% in the optical case. And in very few cases, one of them definitely beats the other, but this occurrs symmetrically, and is due just to luck. For example, the practical one may take a decision which on average is worse, but it is lucky and a little (low-probability) segment is selected, this way beating the optimal algorithm (recall that it is optimal just on average).

This shows that no more complex heuristic is advisable, since the one we propose is so close to optimal.

We are also going to use the simulators to show that the analysis are reasonably close to reality. Our first results confirm this fact.

6 Summary and Conclusions

We began by showing that binary search does not search optimally (in terms of time cost) a text that is indexed with a PAT array on disk, since data elements have different access costs depending on their position on the disk and the current head position.

We then proposed an algorithm to optimize the total access cost for this case, and showed that its CPU cost makes it impractical. We described a heuristic algorithm, which we analyzed and found to have similar performance to the optimal algorithm.

We have explored the use of our algorithms in both magnetic and optical disks. It has been shown that with magnetic disks, the total average search time may be reduced to less than 40% of the binary search time, while with optical disks, the cost is close to 65% of that of the binary search.

We must complete our simulations to measure the impact of the different parameters (e.g. text size) on this reduction factor, both for the optimal and the practical

algorithms, and determine how close they are. We also have to show that our analysis are reasonably close to reality. Finally, we must determine whether is it advisable to use the optimal algorithm for PAT blocks of small size.

This work shows that when making indirect search on disks, it is possible to use algorithms that are much better than the naive one, by minimizing the total cost instead of the number of accesses. This idea can be applied to other scenarios involving dynamically varying access costs.

Acknowledgements

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, and Programa de Cooperación Científica Chile-Brasil de Fundación Andes.

References

[1] 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.

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

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

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

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

[6] E. F. Barbosa R. Baeza-Yates 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, October 1994.