An Optimal Index for PAT Arrays

Gonzalo Navarro[1]

Dept. of Computer Science, University of Chile.
Blanco Encalada 2120, Santiago – Chile. gnavarro@dcc.uchile.cl

Abstract. We study the problem of keeping in main memory an index for a large PAT array stored in secondary storage. This index is used to minimize the number of accesses to secondary memory, performing the main part of the search in main memory. However, we have a maximum allowed size for this index, and it is not clear which is the optimum between keeping many short keys or few long keys. We first derive the optimality criterion, then develop and analyze an algorithm to find the optimum index while building the PAT array, and finally show a probabilistic algorithm that allows to trade efficiency for precision.

1 Introduction

Large textual databases need efficient indexing techniques to access the information they store.

Inverted files are the most common type of textual indices used in information retrieval systems [FBY92, chapter 3]. They impose a storage overhead between 30% and 100%, depending on the retrieval capabilities and the use of stopwords. The search time is logarithmic.

A newer type of index is the PAT array [FBY92, chapter 5] or suffix array [MM90], which is a compact representation of a digital tree (or trie [Knu73]) built on all the positions of interest of the text. Each position is represented by the string (or *suffix*) that starts there and extends arbitrarily to the right. The PAT array stores the leaves of that trie, such that each cell represents a suffix (by storing its start position), and suffixes become lexicographically sorted in the array.

PAT arrays achieve similar space and time complexity than inverted files. However, they can efficiently answer a number of queries that are difficult or inefficient over inverted files. For example, 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 are an interesting choice when designing an indexing scheme for text databases that are not updated frequently.

Since PAT arrays store pointers to the text, searching is indirect. That is, a key must be binary searched in the PAT array, but each comparison must access the text at the position indicated by the PAT array cell. Although binary searching is fast, the amount of seek that must be carried out on magnetic or optical disks may be significant and deteriorate the search time of a naive implementation.

In [BYBZ94, Bar95], a solution to improve the performance of text retrieval from disk is proposed, by using a hierarchy of indices. The hierarchy uses the main memory to store (part of) suffixes extracted from the PAT array at fixed intervals, so that a direct binary search can be performed in main memory, and hopefully only a small portion of the PAT array has to be read from disk to finish the (now indirect) search. We call "PAT index" the part of the array that is kept in main memory. [BYBZ94] considers a two-level (memory - disk) and a three-level (memory - magnetic disk - optical disk) hierarchy. Observe that the cost of the whole search is dominated by the final indirect search, and proportional to the size of the block to solve. In [BNBY+95] it is shown how to reduce the expected time of the final indirect search, by deviating from binary search. The idea is to partition the array not exactly in the middle but in a cell whose track is near the current one's, if the badness of the non-binary partition is balanced by the low cost of the access.

In this work we address the problem of how to minimize the expected size of the PAT array block that has to be read into main memory to finish the search. Since we have a limited amount of main memory, we have to balance between the length of the keys that represent the selected suffixes and the number of keys that can be hold. Longer keys minimize the probability of reading many blocks, but allow less keys to be included and therefore make the blocks longer. See Figure 1.

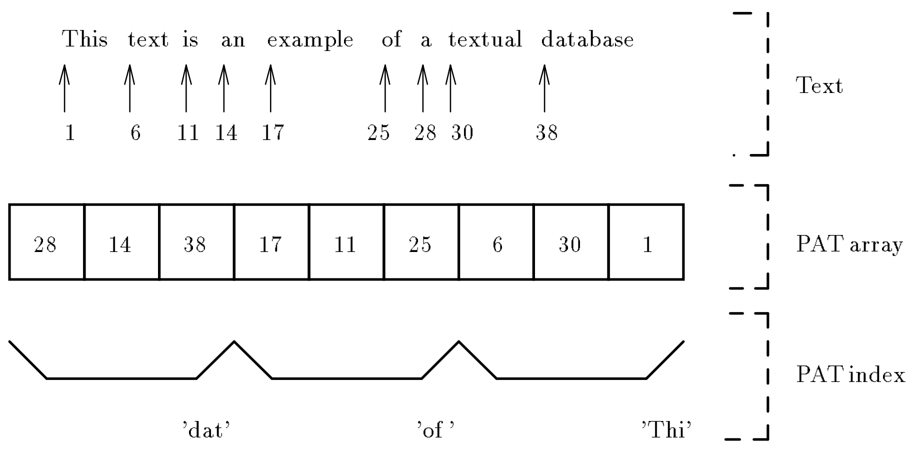

Fig. 1. An example of a text database, a PAT array built on it, and an index on the PAT array, taking a suffix each three positions, and taking three characters from each suffix.

The main contribution of this work is the optimality criterion for the PAT index and the different algorithms to collect the required statistics on the text database, allowing a tradeoff between precision and efficiency.

This work is organized as follows. In Section 2 we review previous work. In Section 3 we introduce notation and derive the optimality problem. In Section 4 we draw an algorithm to find the optimal index. In Section 5 we analyze the algorithm and the resulting index. In Section 6 we develop a probabilistic algorithm that collects approximate statistics on the text at low cost. In Section 7 we show experimental results that validate our model. Finally, in Section 8 we present our conclusions and future work directions.

2 Previous Work

The problem of selecting the optimal key length is also studied in [Bar95]. They consider whether or not is it better to force the keys in the PAT index to be different. They argue, and experimentally show, that it is preferable to allow repeated entries. The arguments regard not only that the keys are forced to be too long, but also that it is very costly to build the index. In this work, we prove analytically that forcing different keys is not optimal.

They do not solve, in fact, the problem of selecting the best key length, but a different one: to find a key length that does not produce more than a given bound of repeated entries. They present an algorithm that performs $\approx M/6$ disk accesses, where M is the total size, in bytes, of memory available for the PAT index. They experiment with a 600 Mb text database, where only the beginnings of non-stopwords are indexed. They find in practice that a key length between 20 and 30 is advisable.

We solve the problem of selecting the best key length, by running statistics on the text. We allow more or less precision for those statistics, our corresponding algorithms taking more or less time or memory.

3 The Problem

Let $T[1..N]$ be a long sequence of characters representing the content of a textual database. Let $P[1..n]$ be a PAT array built on the interesting positions of T (e.g. beginnings of words). Then, $P[i]$ represents the suffix $T[P[i]...]$, and we have that $T[P[i]...] \leq T[P[i+1]...]$ lexicographically.

Let M be the total amount of memory available for our PAT index. That PAT index holds one key of length ℓ for each block of the PAT array. Let $S[1..r]$ be our PAT index, where each $S[i]$ holds $T[P[bi]..P[bi]+\ell-1]$, where b is the size of the blocks. It holds $n \approx rb$, and it must hold $r\ell \leq M$. Our problem is to determine the best option for ℓ.

In order to search a key on this index, the ℓ first characters of the key are binary searched in the PAT index. The result of this search is a portion of S, call it $S[m..m+t-1]$, whose strings are equal to $key[1..\ell]$, where $t \geq 0$ ($t = 0$ means no equal string in the index). Hence, we must read into memory the portion of the PAT array corresponding to $P[bm..b(m+t+1)-1]$, i.e. $b(t+1)$ elements. Since the cost of the search is dominated by the final indirect search, our goal is to minimize the expected number of elements of P that must be read into memory and processed. We call that number T_ℓ.

Let p_ℓ be the probability that two strings of length ℓ match. These two strings are drawn from the text or from a query. Since the PAT array can be considered a uniform random sampling of n strings of size ℓ, the expected number of those strings equal to a given query is np_ℓ, thus the expected number of blocks occupied by those np_ℓ strings is $np_\ell/b = rp_\ell$. This is the expected value of t. Hence,

$$T_\ell = b\ (1 + rp_\ell) = n\left(\frac{1}{r} + p_\ell\right) = n\left(\frac{\ell}{M} + p_\ell\right) \tag{1}$$

what shows that the size of the block is proportional to the length of the PAT array, but the factor is quite small and can be optimized by properly selecting ℓ. The intuition behind the 1 that adds to rp_ℓ is that, even if the query is not present in the array, one block is read.

This proves analytically what [Bar95] argues empirically. The criterion of selecting the smaller ℓ that makes all keys different is not optimal, since that ℓ is one of those considered in the above equation, and not necessarily the optimal.

We have two choices: we can modelize our database by providing a theoretical value for p_ℓ and solve $p'_\ell = -1/M$; or we can compute p_ℓ for our present database and find the optimum by trying each value for ℓ. In the next section we provide an efficient algorithm to do the second.

Providing a theoretical model may be not easy, especially when dealing with natural language. Moreover, we can afford a reasonable overhead at indexing time to run the statistics on the text that determine p_ℓ. However, it must be kept in mind that we have to assure that the queries will follow the same probabilistic distribution of the text. If this is not the case, the only choice is to use a theoretical model.

4 An Algorithm to Compute the Optimal Index

We present an algorithm that computes p_ℓ for all values of ℓ while the PAT array is being built. The problem of finding the optimal ℓ is then trivial.

Consider all the sorted suffixes of the text, truncated at ℓ characters. Call $s_1, ...s_k$ the positions where a suffix differs from its previous. Call also $s_0 = 0$ and $s_{k+1} = n + 1$. Then, in our database, p_ℓ is the probability of two suffixes a and b being equal, i.e.

$$p_\ell = \sum_{i=0}^{k} P(a \in s_i..s_{i+1} - 1)P(b \in s_i..s_{i+1} - 1) = \sum_{i=0}^{k} \frac{(s_{i+1} - s_i)^2}{n^2} \tag{2}$$

We can consider the $s_i..s_{i+1}$ as "areas" where the suffixes do not change. The longer ℓ, the smaller the areas. See Figure 2.

Our algorithm traverses the PAT array computing in parallel p_ℓ for every relevant ℓ. Observe that there exists an ℓ such that $p_\ell = 1/n$ from there on. That one is the first non-relevant ℓ.

We keep two arrays, indexed by lengths (values for ℓ). `val` is used to compute the sum (2), while `last` keeps track of the last time that the suffix of each length

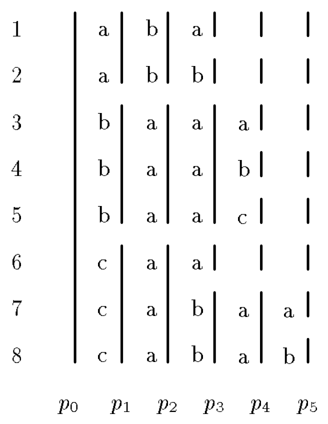

Fig. 2. An example of the execution of the algorithm. The lines show the "areas" for each level, whose lengths the algorithm computes. Note that when an area is cut at a given level, it is cut from there on.

differed from its previous (i.e. where the current area started). For each new suffix we consider, we measure at which length it differs from the previous. From that point on, all "areas" terminate and new areas begin, so we update **val** and **last**.

Figure 3 shows this algorithm. It is not the final algorithm, but helps to understand it. We use a pseudocode notation, the arrays starting at position 1.

The main problem of the algorithm is that we cannot know where to stop when updating the values. For example, the relevance of p_4 in Figure 2 is noticed only when we process the element 8. Because of this, the algorithm is $O(nM)$ time, which is too large.

```
ComputeLengths (T,P,n)

for k := 1 to M
   val[k] := 0
   last[k] := 0
for i := 1 to n+1
   if (i = 1) or (i = n) dif := 1
   else dif := first place where T[P[i-1]...]  and T[P[i]...]  differ
   for k := dif to M
      val[k] := val[k] + ((i - last[k])/n)**2
      last[k] := i
```

Fig. 3. Naive algorithm.

However, the same Figure 2 gives us the tip to improve the algorithm. While a length is not active (i.e. when the two current strings differ before that length), we are always adding $1/n^2$ to it in the internal **for**, what adds up $1/n$ along the whole process. So we start by assigning $1/n$ to each new length to consider, and only when it has an area longer than 1, we add as above and subtract $(s_{i+1} - s_i)/n^2$, that stands for all the $1/n^2$'s that we added by default for that area. This allows to work only for non-trivial areas, what leads to a much more efficient algorithm.

We keep the previous value for `dif`, that we call `pdif`. That is, `pdif` is the point where the previous suffix differed from its previous suffix. We need to work only between `dif` and `pdif`. If the first one is larger, we update the `last` array between both positions, since the areas that start are longer than 1 and deserve processing. If the second is larger, we update the `val` array between both positions, since all areas from `dif` to `pdif` are being ended. The trick of the $1/n$ initial value takes care of all size-1 areas. Figure 4 shows the final algorithm.

```
ComputeLengths (T,P,n)

size := 1
pdif := 1
for i := 2 to n+1
   if (i = n+1) dif := 1
   else dif := first place where T[P[i-1]...]  and T[P[i]...]  differ
   for k := dif to pdif-1
      val[k] = val[k] + ((i - last[k])**2 - (i - last[k]))/n**2
   if (dif > size)
      enlarge val and last to size dif
      for k := size to dif-1
         val[k] := 1/n
      size := dif
   for k := pdif to dif-1
      last[k] := i-1
   pdif := dif
```

Fig. 4. Final algorithm. Note that the arrays grow as needed.

Once the algorithm computes `val[0..size-1]`, we traverse those numbers looking for the minimal value of $\ell/M + \texttt{val}[\ell]$.

5 Analysis

There are two main components in the time cost of this algorithm. The first one is to find the first character where two consecutive strings differ; the other is the sum of the $|\texttt{pdif} - \texttt{dif}|$ values, i.e. the difference of two consecutive positions where two suffixes differ.

We first focus on the second part of the cost. In Figure 5 we repeat the example of Figure 2, this time boxing the areas corresponding to each length.

On the right side we show the trie for those suffixes. It should be clear that each boxed area corresponds to a non-leaf subtree of the trie, i.e. to an internal node. On the other hand, we work $O(1)$ for each area, since we start it when we set `last` and we end it when we update `val`. So the amount of work is proportional to the number of internal nodes of the corresponding trie.

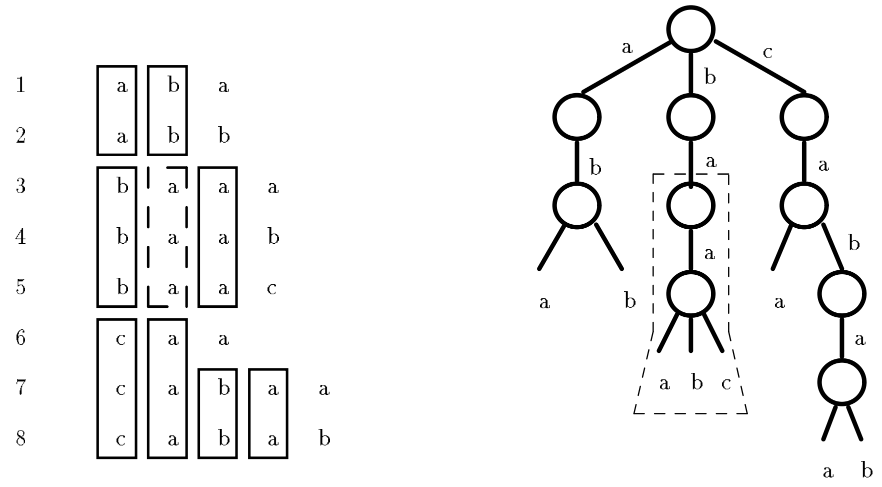

Fig. 5. The same example as before, now showing the areas as boxes and the associated trie. The dashed box has been mapped to its subtrie.

The first part of the cost may be null if the PAT building process provides the `dif` positions. If that is not the case, the cost corresponds to the external path length, since every path from the root to the parent of each leaf is the common part of a suffix with its previous one, and we have to compare one more character to detect the difference. Notice that if we run the algorithm on the trie instead of the PAT array, this cost becomes null too.

Finally, the expected space complexity is the height of the trie.

In [Szp92], these statistics are computed asymptotically on a suffix trie, for large n, using a Markovian model (which is quite good for natural language). We take here the simpler case of independent character generation (i.e. a Markovian model with no memory). The only difference in the general case is the constant factor of the results, not the order. The reader is referred to [Szp92] for more details.

Suppose our alphabet is composed from a finite or infinite number of symbols, call q_i the probability of the i-th symbol, and call $H = \sum_i q_i \log(1/q_i)$ the entropy of the language. Then

– The height of the suffix trie is

$$H_n = \frac{2 \ln n}{\lim_{\ell \to \infty} \frac{\ln 1/p_\ell}{\ell}} = O(\log n)$$

where good approximations can be obtained by using a reasonable large ℓ.

– The external path length is

$$E_n = \frac{n \ln n}{H} = O(n \log n)$$

– The number of internal nodes is

$$S_n = \frac{n}{H} = O(n)$$

This last result is taken from [Mah91], for random tries. Statistics for random tries are equivalent to suffix tries over a random text, except for $o(1)$ terms [SF96].

Therefore, the space complexity of our algorithm is $O(\log n)$, and the time complexity is $O(n)$ if we have already computed the `dif` values or we work on the trie, $O(n \log n)$ otherwise. The constants depend on the probability distribution of the language.

These costs are much cheaper than those of the PAT building process, which is $O(n \log n)$ string comparisons, each one costing from $\sum_\ell \ell(1-p_\ell)$ (at first, when random strings are compared), to $E_n/n = O(\log n)$ (the average height of a leaf) at last, when close strings are compared.

For example, if we have a uniformly distributed alphabet with c symbols, the space complexity is $2 \log_c n$, and the time complexity is $n/\ln c$ for precomputed `dif` and $n \log_c n$ otherwise. Moreover, the optimum ℓ for this case can be directly computed by differentiating Eq. (1), since $p_\ell = 1/c^\ell$:

$$-\frac{1}{M} = p'_\ell = -\frac{\ln c}{c^\ell} \quad \Rightarrow \quad \ell = \log_c(M \ln c)$$

In this case, the expected size of the block to work with is

$$T_\ell = n\left(\frac{\ln(M \ln c) + 1}{M \ln c}\right) = O(n \log M/M)$$

6 Approximate Statistics

Although the previous algorithm provides us the exact values of p_ℓ, it is far from practical in a text database stored on disk. This is because, although the algorithm is very fast in RAM, each comparison has to access the text at a random position on the disk, which takes about 15 ms on magnetic disks, or even 1/3 second on optical disks. If we consider a 600 Mb database on magnetic disk and indexing only non-stopwords, we need more than ten days.

To make this algorithm practical, we observe that in fact it is not necessary to compute the exact values of p_ℓ. A reasonable approximation is acceptable. We propose a probabilistic algorithm that is much cheaper than the exact algorithm, and that allows more or less precision depending on how much main memory we can provide.

The PAT building process proceeds by caching the text by portions into main memory, sorting the entries, writing the partial PAT block to disk, and later merging the partial blocks [FBY92]. When each block is sorted in main memory, we can run our full statistics algorithm only in the block (the text is also in main memory). Later, we can average our estimated p_ℓ over all blocks. This is extremely efficient, since no disk access is performed.

Suppose we can build the PAT array from blocks of size s at a time. That means that we perform $N = n/s$ samples in the algorithm, where each sample is obtained by running the exact algorithm on all the s elements of the PAT block.

Let X_ℓ^{ij} be 1 if the entries i and j of the PAT block are equal up to ℓ characters. Clearly, X_ℓ^{ij} has a binomial distribution with parameter p_ℓ. The process of sorting and then comparing consecutive entries is equivalent to comparing all the X_ℓ^{ij} against all (for each i and j). Hence, we are defining a random variable Y_ℓ, that comes from averaging the result of s^2 comparisons of each entry i against each other j.

$$Y_\ell = \frac{1}{s^2} \sum_{ij} X_\ell^{ij}$$

Obviously, $\mu(Y_\ell) = p_\ell$. The variance is more complicated. We have $\sigma^2(Y_\ell) = \mu(Y_\ell^2) - \mu(Y_\ell)^2$. The first term is

$$\begin{aligned} \mu(Y_\ell^2) &= \mu\left(\frac{1}{s^2}\left(\frac{1}{s^2}\sum_{ij}(X_\ell^{ij})^2 \ + \ \frac{1}{s^2}\sum_{ij \neq kl} X_\ell^{ij} X_\ell^{kl}\right)\right) \\ &= \frac{1}{s^2}\, p_\ell \ + \ \left(1 - \frac{1}{s^2}\right) p_\ell^2 \end{aligned}$$

where we used that $(X_\ell^{ij})^2 = X_\ell^{ij}$ (since it takes the values 0 or 1), and that X_ℓ^{ij} is independent of X_ℓ^{kl} if $(i,j) \neq (k,l)$. We can now compute $\sigma^2(Y_\ell) = p_\ell(1-p_\ell)/s^2$.

Since we average N samples of Y_ℓ, our final statistic has

$$\begin{aligned} \mu(\overline{Y_\ell}) &= p_\ell \\ \sigma^2(\overline{Y_\ell}) &= \frac{p_\ell(1-p_\ell)}{Ns^2} = \frac{p_\ell(1-p_\ell)}{ns} \end{aligned}$$

while from Chebyschev's inequality we know that

$$P(|\overline{Y_\ell} - p_\ell| > \epsilon) \leq \frac{\sigma^2(\overline{Y_\ell})}{\epsilon^2} = \frac{p_\ell(1-p_\ell)}{ns\epsilon^2} \leq \frac{1}{4ns\epsilon^2}$$

We measure the error in terms of the amount of additional work at query time. As shown in Eq. (1), the expected size of the block to solve in the final indirect search is $n(p_\ell + \ell/M)$. If we misestimate p_ℓ in ϵ, we could select a non-optimal ℓ'. However, the difference between both values of $p_\ell + \ell/M$ cannot be larger than 2ϵ, and hence the difference between the optimal and the obtained average block to solve is not larger than $2n\epsilon$.

We show in [BNBY+95] that the cost to solve a block of length b is $O(\log b)$ (since we deviate from binary search, the base of the logarithm is between 2 and 3/2). Hence, the number of disk accesses is less than

$$\log_{3/2}(n(p_\ell + \ell/M + 2\epsilon)) \leq \log_{3/2}(n(p_\ell + \ell/M)) + \frac{2\epsilon}{\log(3/2)(p_\ell + \ell/M)}$$

where the first term of the sum is the optimum number of disk accesses and the second one the extra accesses due to misestimation. Unfortunately, the number of extra accesses depends on the final size of the block, so it cannot be estimated beforehand. We can only use a gross bound $p_\ell + \ell/M \leq 1/M$, which gives $2M\epsilon/\log(3/2)$, i.e. about $5M\epsilon$ additional disk accesses. That means that, if we can afford k extra disk accesses for each query, we have to set $\epsilon = k/5M$.

Returning to the probability of misestimating by ϵ, we conclude that the probability of paying k more disk accesses because of the approximation is much less than $6M^2/nsk^2$. For instance, if we have 600 Mb of text and index only the beginning of non-stopwords (i.e. $n \approx 60$M), and use $M = 2$ Mb and $s =$ 1M (i.e. about 5 Mb of main memory, since we need a cell of the PAT block for each character), then the probability of making k additional accesses is $0.4/k^2$, which is more than reasonable (e.g. 10% of probability of performing two extra accesses). Keep in mind that our analysis is extremely pessimistic.

We can improve the approximation by enlarging s, i.e. sorting larger blocks in main memory. Therefore, we can trade main memory for precision, or time for precision (if we prefer to run the algorithm on disk or a combination of both). It is normally reasonable to assume that one has more memory for indexing than for querying. Even if we have the same memory, i.e. $s = M/5$, in the above example we have 10% of probability of performing more than 3 additional accesses.

7 Experimental Results

We tested the behavior of our algorithm in practice, both on random and English text. In the first case, we generated random strings on a uniformly distributed alphabet of 32 symbols (close to normal text), and indexed each position. In the second case, we filtered English literary texts, converting letters to lowercase, and all but letters and numbers to blanks; and indexing only word beginnings. Hence, in what follows, a text of size n means that it has n index points, although the real text is longer in the English case.

We ran our simulations on a Sun SparcClassic with 16 Mb RAM, of approximately SpecMark 26, running SunOs. We measured Unix user times, averaging each number over ten samples. Since the PAT building process is intended to be run in real main memory (no paging), user times accurately reflects the costs.

We first measured the time taken by building the PAT array and by computing the values of p_ℓ with our algorithm. Figure 6 shows the results. As it can be seen, the additional time paid to run our algorithm is less than 10% of the sorting process (and less than 5% on random text), which shows that we can determine the best partition with little overhead. It is also apparent that times are larger on natural language, since it is more common to have long repeated strings inside the text. For instance, the height of the tries, which is directly related to the space complexity of the algorithm, is between 7 and 9 for random text, while it goes from 75 to 200 for English text. The average depth of a leaf (i.e. the number of character comparisons per string), which is directly related to the time complexity of the algorithm, is near 4.3 for random text, and near 11 for English text.

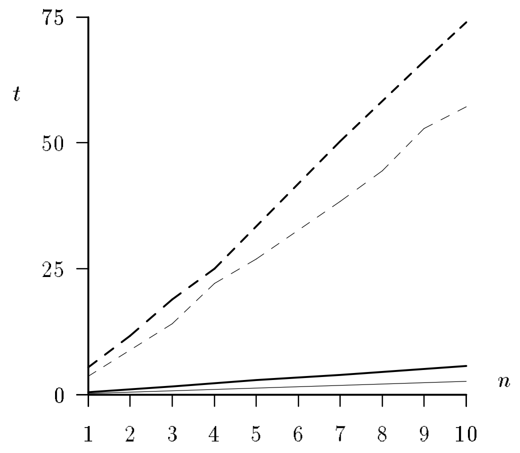

Fig. 6. Times, in seconds, to process a text with n index points (n is indicated in hundreds of Kb). Full lines are the times of our algorithm, dashed lines correspond to the PAT building process. Thick lines are for English text, thin lines for random text.

Our second experiment shows the behavior of ℓ when M grows. We tested with an English text of $n = 2\text{M}$. See Figure 7. The ℓ grows very slowly as expected, from 8 to 16. This does not contradict the results of [Bar95], since our text is small compared to theirs, at the point that past $\ell = 20$, our p_ℓ reflects more the structure of our particular text than the true statistics of natural language text. That is, although the true p_ℓ is not dependent on n, there are too few strings equal up to 20 characters in our texts, and hence we cannot accurately estimate p_ℓ from $\ell = 20$ on.

Finally, our third experiment shows how close to reality are our estimations. We plot the estimated values of T_ℓ (using Eq. (1)), together with the real sizes of the resulting blocks when searching for words randomly selected from the text. We averaged over 10,000 searches. As Figure 8 shows, our estimations are very accurate in the interesting area, what shows that our theoretical predictions accurately determine the optimum ℓ.

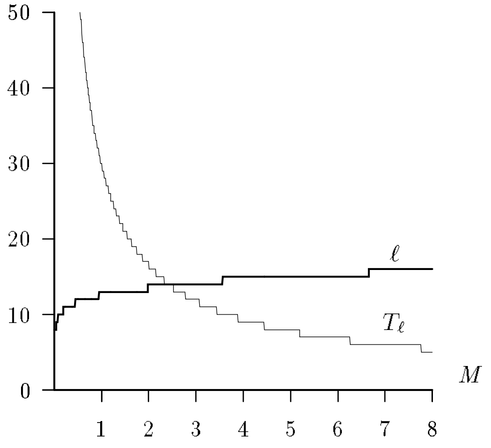

Fig. 7. The optimum ℓ (thick line) and the corresponding T_ℓ (thin line). M is expressed in Mb. Recall that T_ℓ is for $n = 2$M, and must be scaled if applied to other n.

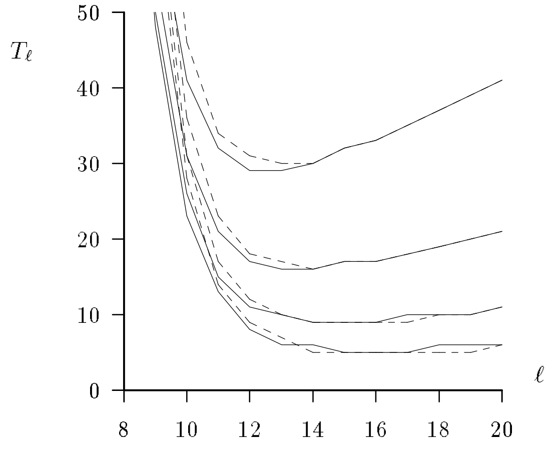

Fig. 8. The estimated (full line) and experimental (dashed) values of T_ℓ, for different values of M. From top to bottom, the lines are for $M = 1$, 2, 4 and 8 Mb.

8 Conclusions and Further Work

We addressed the problem of computing the best index for a PAT array given a limited amount of main memory. We found the formula to optimize and developed an efficient algorithm to compute the needed statistics on the PAT array, if no theoretical model can be used to find a formal solution. We analyzed the algorithm and, for uniformly distributed alphabets, the resulting index, finding that the cost of the algorithm is negligible compared to that of building the PAT array. We also developed and analyzed a probabilistic algorithm to compute the statistics approximately, which is practical for databases stored on disk. Finally, we conducted a number of experiments that allow to conclude that our theoretical model is satisfactory in terms of closeness to reality.

We need more experiments, particularly on larger text databases, so that we can find more stable values for p_ℓ, at least to $\ell = 40$ or 50. This would allow us to draw conclusions about the optimal values of ℓ in typical text, for each value of M. We need also to test the probabilistic algorithms developed, in order to measure the estimation error.

There are a number of interesting research directions. One of them is that the algorithm may be refined, since given that we minimize $p_\ell + \ell/M$ and that `val[k]-val[k+1]` can only increase along the process, whenever the difference is larger than $1/M$ we can discard `val[k]` as a candidate. So we can keep a linked list of surviving candidates instead of an array, and at the end the optimum is the first element of the list. This can reduce the constant factor of the time complexity.

It is also interesting to study other variations, for example: could it be convenient to use non-fixed partitions (i.e. different block sizes for each key), provided the space overhead it imposes? Is it better, in a natural language database, not to use a fixed ℓ but, for example, whole words? [BZ95].

Finally, the understanding of this process can provide us tools to analyze some compression schemes for PAT arrays, such as [BZ95, Bar95], where they store the differential strings of size $\ell -$ `dif`. For instance, since the average value for `dif` is E_n/n, and $\ell \leq H_n$, an immediate (though not very tight) bound for the size of the index with n strings is $H_n - E_n/n$. In the case of uniform distribution with c characters, this is $n \log_c n$. Another idea (that we have shown not optimal) is to select minimal ℓ such that no two strings in the PAT index are equal. Since for a given r the appropriate ℓ is $1 + H_r$, we can numerically solve $\ell = 1 + H_{M/\ell}$ to obtain ℓ.

Acknowledgments

We thank the many helpful comments of the anonymous referees.

References

[Bar95] E. Barbosa. *Efficient Text Searching Methods for Secondary Memory.* PhD thesis, Univ. Federal de Minas Gerais, Brazil, September 1995. Technical Report 013/95.

[BNBY+95] E. Barbosa, G. Navarro, R. Baeza-Yates, C. Perleberg, and N. Ziviani. Optimized binary search and text retrieval. In P. Spirakis, editor, *Proceedings of ESA'95 Third Annual European Symposium on Algorithms*, pages 311–326, Corfu, Greece, September 1995. Springer-Verlag. LNCS 979.

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

[BZ95] E. Barbosa and N. Ziviani. From partial to full inverted lists for text searching. In *Proceedings of WSP'95 Second South American Workshop on String Processing*, pages 1–10, Valparaíso, Chile, April 1995.

[FBY92] W. Frakes and R. Baeza-Yates, editors. *Information Retrieval: Data Structures and Algorithms.* Prentice-Hall, 1992.

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

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

[Mah91] H. Mahmoud. *Evolution of Random Search Trees.* John Wiley & Sons, 1991.

[MM90] U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. In *Proceedings of the 1st ACM-SIAM Symposium on Discrete Algorithms*, pages 319–327, San Francisco, CA, January 1990.

[SF96] R. Sedgewick and P. Flajolet. *Analysis of Algorithms.* Addison-Wesley, 1996.

[Szp92] Wojciech Szpankowski. Probabilistic analysis of generalized suffix trees. In A. Apostolico, M. Crochemore, Z. Galil, and U. Manber, editors, *Proceedings of CPM'92 Third Annual Symposium on Combinatorial Pattern Matching*, pages 1–14, Tucson, Arizona, April 1992. Springer-Verlag. L-NCS 644.

This article was processed using the LaTeX macro package with CUP_CS class