A Practical Index for Text Retrieval Allowing Errors *

Ricardo Baeza-Yates Gonzalo Navarro

Department of Computer Science, University of Chile
Blanco Encalada 2120 - Santiago - Chile
{rbaeza,gnavarro}@dcc.uchile.cl

Abstract

We propose a text indexing technique for approximate pattern matching, which is practical and especially aimed at Information Retrieval (IR). Unlike other indices of this kind, it is able to retrieve any string that approximately matches a given search pattern. Every sequence of a fixed length appearing in the text is stored in the index, together with pointers to all the positions where it appears. The search pattern is cut into pieces so that at least one must match exactly. All the pieces are searched in the index and the union of candidate positions is verified. To reduce space requirements, pointers to blocks instead of exact positions can be used, which increases querying costs. We design an algorithm to optimize the pattern partition into pieces so that the total number of verifications is minimized. This also allows to know in advance the expected cost of the search and the expected relevance of the query to the user. We show experimentally the build time, space requirements and query times of our index, finding that it is a practical alternative for IR.

Keywords: Approximate String Matching, Information Retrieval, Text Indexing.

1 Introduction

The problem of approximate string matching has a number of applications in computer science, such as text retrieval, computational biology, signal processing, pattern recognition, etc. It is defined as follows: given a long text of length n, and a (comparatively short) pattern of length m, retrieve all the segments (or "occurrences") of the text whose *edit distance* to the pattern is at most k. The *edit distance* between two strings is defined as the minimum number of character insertions, deletions and replacements needed to make them equal. It is common to report only the endpoints of occurrences. We call $\alpha = k/m$ the "error ratio".

In the on-line version of the problem, it is possible to preprocess the pattern but not the text. The classical solution involves dynamic programming and is $O(mn)$ time [20]. Recently, a number of algorithms improved the classical one, for instance [24, 9, 22, 8, 27, 28, 4]. Some of them are "sublinear" in the sense that they do not inspect all the characters of the text, but of course the on-line problem is $\Omega(n)$ if m is taken as constant. In [4, 3], it is shown that [8] is the fastest algorithm for moderately low error ratios and pattern length. Our present work can be seen as an off-line version of that algorithm.

We are particularly interested in information retrieval (IR), where the text is normally so large that the on-line algorithms are not practical. Moreover, queries are more frequent than changes and therefore the text can be preprocessed, the query patterns are not too large (i.e. less than 25 letters), the alphabet size (σ) is not very small (36 at least) and expected error ratios are $\leq 1/3$ (since otherwise the query returns too many matches and is useless to the user).

*This work has been supported in part by Fondecyt grants 1950622 and 1960881.

Classical indices for text databases allow fast search of exact patterns. These indices, however, are unable to retrieve a word which has been mistyped. This is very common in texts obtained by optical character recognition (OCR), or when there is no quality assurance for the content of the database (e.g. when indexing the World Wide Web). Moreover, the query may be misspelled or we may not remember the exact spelling of a foreign name. The edit distance defined above captures very well such errors.

The first indexing schemes for approximate text retrieval have appeared only a few years ago. There are two types of indexing mechanisms: word-oriented and sequence-oriented. In the first one, the index is capable of retrieving every *word* whose edit distance to the pattern is at most k. In the second one, useful also when the text is not natural language, the index is capable of retrieving every *sequence*, without notion of word separation.

Indices of the first kind store the set of all different words of the text (the *vocabulary*) and use an on-line algorithm on the vocabulary, thus obtaining the set of words to retrieve. From that point on, the problem does not need to involve approximate matching anymore. Since the vocabulary is sublinear in size with respect to the text, they achieve acceptable performance. These indices are not capable, however, of retrieving an occurrence that is not a complete word. For instance, if an OCR system has erroneously inserted a space in the middle of a word in the text, it will be not possible to search that word with one error and retrieve it using a word-retrieving index. Examples of these indices are Glimpse [18], Igrep [1] and [5].

In the indices of the second kind, the words are disregarded. They also apply if words do not exist in the text, such as in DNA or protein databases.

One class of indices for this case is based on building the suffix tree of the text and traversing it instead of the text, to avoid its redundancies [25, 10, 11, 7]. The main problem with this approach is that suffix trees pose heavy space requirements: the index, unless compressed, is twelve times the size of the text. Approaches to compress the suffix tree are still in their beginnings and have not been implemented yet [14]. If the index does not fit in main memory (which is usually the case), the construction process is very costly, even if the suffix tree is converted to a suffix array [17], to which [11, 7] can be adapted.

A second class reduces the problem to exact matching of substrings of the pattern, and uses an index that searches the substrings with no errors [12, 23, 19]. Later, the occurrences of those matching substrings have to be verified to search the complete pattern. These indices can be efficiently built and take less space than the others. However, they are less tolerant to errors.

In this work we propose a sequence retrieving index especially aimed at IR scenarios, in the same lines of reducing the problem to exact matching. We show also an algorithm to optimize the partition of the pattern in order to minimize the number of text positions to verify. This also allows to predict the cost of the search and to give early feedback to the user about the approximate size of the result set. In case of too many verifications (which involves probably too many results), the user may preempt the search, given the poor precision to be obtained.

2 Previous Work

The idea of reduction to exact partitioning has been used many times for on-line searching [27, 8, 22, 4]. The basic idea is as follows: if a pattern occurs in the text with k errors, and if we cut the pattern in $k+1$ pieces arbitrarily, then at least one of the pieces must be present in the occurrence with no errors. This is easily seen by considering that each error modifies at most one piece of the pattern, and therefore at least one piece survives unchanged. To find all approximate occurrences it suffices to search all pieces and check their neighborhood.

Many generalizations of the idea have been studied. It has been shown that if the pattern is cut in less pieces (say j) then the subpatterns are to be searched with $\lfloor k/j \rfloor$ errors [4, 7].

Overlapping pieces have been considered in [22]. If *all* the pieces of length q (called q-grams) in the pattern are searched, then the search needs not to inspect every text position, but "samples" separated by h characters that are not inspected at all. Moreover, they may also force that at least s pieces are present in the candidate text area, by modifying h (s and h are related).

Recently, a particular case of matching more than one piece has been proposed [21]: if the pattern is cut in $k+s$ pieces, then at least s pieces must be present in every occurrence (moreover, they observe some positional constraints). This increases the tolerance to errors in long patterns. However, if the pattern is not long this partitioning gives very short pieces, which tend to trigger much more verifications.

Despite all generalizations, the original partitioning idea leads to the fastest on-line algorithm for moderate pattern length and error ratios, as shown in [4]. This is the typical case in IR.

The first idea to apply reduction to exact partitioning to indexing is [12], where the q-grams approach is used. The positions of all q-grams are stored. To search a pattern of length m, the text is divided into blocks of size $2(m-1)$. The number of all q-grams of the pattern that fall into each block is computed. Each text block with at least $m+1-(k+1)q$ q-grams falling in is verified with dynamic programming.

Independently, in [2] an alternative to Glimpse is proposed to allow more general searches. Instead of indexing every word as Glimpse does, they index every substring of a fixed length q. Although originally conceived for exact search, it is mentioned the possibility of combining the index with exact partitioning to answer approximate search queries.

The idea of q-grams is used again in [23] with a different approach, more oriented to sampling the text as in [22]. Every text *sample* is stored in the index (hence, the space requirements are reduced). Given a search pattern, its q-grams are searched in the index, and the rest proceeds as in the on-line version. The dependence between s and h allows to use a single index (with samples separated by h characters) for different m and k values (i.e. s is adjusted accordingly). Compression schemes are considered in [13], although the time complexity increases significantly.

Although the q-grams schemes have small space overhead, their tolerance to errors is very low for IR purposes, as shown in [4, 3] for its on-line version. In particular, it is lower than that of the on-line algorithm we are adapting [8].

A somewhat different idea is proposed in [19]. It uses an index where every sequence of the text up to a given length q is stored, together with the list of its positions in the text. Hence, the structure of the index is similar to the one we propose. However, the reduction to exact search is completely different. To search for a pattern shorter than $q-k$, all the maximal strings whose edit distance to the pattern is less than k are generated, and each one is searched in the index. Later, the lists are merged. To handle longer patterns, they are split in as many pieces as necessary to make them of the required length.

The length of the strings stored in the index is made small enough to be able to represent them as computer integers. This allows to build the index in $O(n)$ time, and very quickly in practice. The strings must be short also to avoid an explosive numbers of strings generated at search time.

Query complexity is sublinear for sufficiently low error ratios. This maximum allowed error ratio increases with the alphabet size. For example, the formula shows that it is 0.33 for $\sigma = 4$ and 0.56 for $\sigma = 20$. However, the scheme gets worse (because of the number of strings generated) as σ grows, which is the typical case in IR.

A useful concept to reduce the space requirements of these indices is *block addressing.* The main idea is to cut the text in a number of blocks. Instead of storing all the exact positions where each word or q-gram occurs, only the blocks where it appears are stored. At search time, the candidate blocks must be completely verified, which increases search times.

This concept has been used in word-retrieving indices [18, 5] with good results. It is also used in Grampse [16], which is based on [23] (although approximate search is not implemented yet). As opposed to block addressing, we denote *character addressing* the case when all the positions are recorded.

3 A New Indexing Scheme

Our proposal aims specifically at building a practical index for IR purposes. It indexes all q-grams and uses the simplest partitioning (i.e. in $k+1$ disjoint pieces). This can be seen as an off-line version of [8] (studied more in detail in [4, 3]). This is combined with a new pattern splitting optimization technique to minimize the number of verifications to perform, which is especially useful on natural language texts. Pointers to exact occurrences or to blocks can be used, although we show later that only character addressing gives a useful index.

At indexing time, we select a fixed length q. Every q-gram of the text is stored in the index (in lexical order). To resemble traditional inverted lists, we call *vocabulary* the set of all different q-grams. The number of different q-grams is denoted V, which is $\leq n$ (in a text of n characters there are n q-grams, but only V different q-grams). For the correctness of the algorithms, it is necessary that the last $q-1$ suffixes of the text are entered as q-grams too, even when they are of length $< q$. Together with each q-gram, we store the list of the text positions where it appears, in ascending positional order. Figure 1 shows a small example.

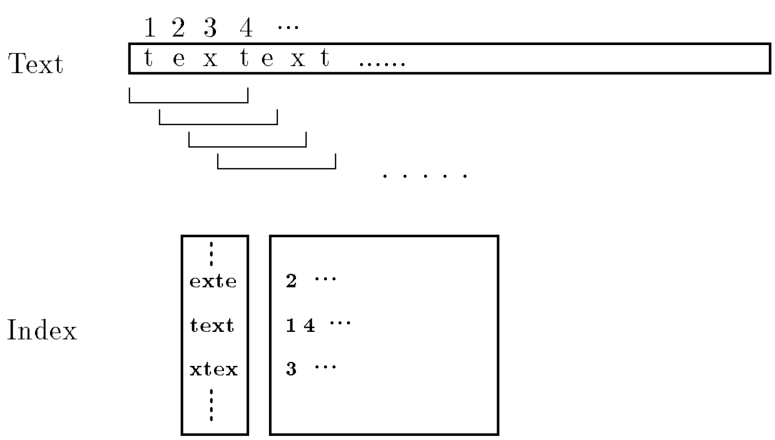

Figure 1: The indexing scheme for $q = 4$.

If block addressing is used, the text is divided in blocks of a fixed length b, and all the q-grams that start in the block are considered to lie inside the block. Only the ascending list of the blocks where each q-gram appears is stored in this case. This makes the index smaller (since there is only one reference for all the occurrences of a q-gram in a single block).

To search a pattern of length m with k errors, we split the pattern in $k+1$ pieces, search each piece in the index of q-grams of the text, and merge all the occurrences of all the pieces, since each one is a candidate position for a match. The neighborhood of each candidate position is then verified with a sequential algorithm. If blocks are used, each candidate block must be completely traversed with an on-line algorithm.

Of course the pieces may not have the same length q. If a piece is shorter than q, all the q-grams with the piece as prefix are to be considered as occurrences of the piece (they are contiguous in the index of q-grams). If the piece is longer, it is simply truncated to its first q letters (it is possible to verify later, in the text, whether the q-gram starts in fact an occurrence of the piece before verifying the whole area).

We describe now a splitting optimization technique to be used at query time.

When the pattern is split in $k+1$ pieces, we are free to select those pieces as we like. This idea is mentioned in [6] for an on-line algorithm as follows: knowing or assuming a given letter distribution for the text to search, the pieces are selected so that the probabilities of all pieces are similar. This minimizes the total number of verifications to perform, on average.

We can do much better here. They key point is that it is very cheap to compute in advance the *exact* number of verifications to perform for a given piece. We just locate the piece in the q-gram index with binary search. In the general case we obtain a contiguous region, for pieces shorter than q. By storing, for each q-gram, the *accumulated* length of the lists of occurrences, we can subtract the lengths at the endpoints of the region to obtain immediately the number of verifications to perform. The complete process takes $O(\log V) = O(\log n)$.

We describe a dynamic programming algorithm to compute the partition that minimizes the total number of verifications to perform. As a side result, we know in advance the total cost to pay to retrieve the results, which as explained is useful as early feedback to the user.

Let $pat[0..m-1]$ be the search pattern. Let $R[i,j]$ be the number of verifications to perform for the piece $pat[i..j-1]$ (computed as explained above), for every $0 \le i \le j \le m$. Using R we build two matrices, namely

- $P[i,k]$ = sum of the verifications of the pieces in the best partition for $pat[i..m-1]$ with k errors,
- $C[i,k]$ = where must the next piece start in order to obtain $P[i,k]$.

Hence, we need $O(m^2)$ space. Computing R as described takes $O(m^2 \log n)$, and the following algorithm computes the optimal partition in $O(m^2 k)$ time.

```
for (i = 0; i < m; i++)
    { P[i,0] = R[i,m]; C[i,0] = m; }
for (r = 1; r <= k; r++)
    for (i = 0; i < m - r; i++)
        { P[i,r] = min_{j ∈ i+1..m-r} (R[i,j] + P[j,r-1]);
          C[i,r] = j that minimizes the expression above; }
```

The final number of verifications is $P[0,k]$. The beginnings of the pieces are $\ell_0 = 0$, $\ell_1 = C[\ell_0, k]$, $\ell_2 = C[\ell_1, k-1]$, ..., $\ell_k = C[\ell_{k-1}, 1]$.

4 Analysis

We analyze the time and space requirements of our index.

To build the index we scan the text in a single pass, using hashing to store all the q-grams that appear in the text. This q must be selected as large as possible, but small enough for the total number of such q-grams to be small (practical values for natural language text are $q = 3..5$).

Although we scan every q-gram and any good hash function of a q-gram takes $O(q)$ time, the total expected time is kept $O(n)$ instead of $O(nq)$ by using a technique similar to Karp-Rabin [15] (i.e. the hash value of the next q-gram can be obtained in $O(1)$ from the current one). The occurrences are found in ascending order, hence each insertion takes $O(1)$ time.

Therefore, this index is built in $O(n)$ expected time and a single pass over the text. The worst case can be made $O(n)$ by modifying Ukkonen's technique to build a suffix tree in linear time [26].

We analyze space now. To determine the number of different q-grams in random text, consider that there are σ^q different "urns" (q-grams) and n "balls" (q-grams in the text). The probability of a q-gram to be selected in a trial is $1/\sigma^q$. Therefore, the probability of a q-gram *not* being hit in n trials is $(1-1/\sigma^q)^n$. Hence, the average number of q-grams hit in the n trials is $V = \sigma^q(1-(1-1/\sigma^q)^n) = \Theta(\sigma^q(1-e^{-n/\sigma^q})) = \Theta(\min(n,\sigma^q))$. This shows that q must be kept $o(\log_\sigma n)$ for the vocabulary space to be sublinear. We show practical sizes in the experiments.

We consider the lists of occurrences now. Since we index all positions of all q-grams, the space requirements are $O(n)$, being effectively $4n$ on a 32-bit architecture[1]. If block addressing is used (with blocks of size b), the same urn argument used above shows that the space requirements are $O(nV/b(1-e^{-b/V}))$, which is $o(n)$ if and only if $V = o(b)$.

We now turn our attention to the time to answer a query. The first splitting optimization phase is $O(m^2(k+\log n))$ as explained. Once we have all the positions to verify, we check each zone using a classical algorithm [24], at a cost of $O(m^2)$ each. This cost is exactly the same as in the on-line version [8], since it is related to the number of occurrences of the pieces in the text.

We analyze only the case of random text (natural language is shown in the experiments). Under this assumption, we discard the effect of the optimization and assume that the pattern is split in pieces of lengths as similar as possible. In fact, the optimization technique makes more difference in natural language texts, making the approach in that case more similar in performance to the case of random text.

Therefore, we split the pattern in pieces of length $\lfloor m/(k+1)\rfloor$ and $\lceil m/(k+1)\rceil$. In terms of probability of occurrence, the shorter pieces are σ times more probable than the others (where σ is the size of the alphabet). The total cost of verifications is no more than

$$\frac{(k+1)m^2}{\sigma^{\lfloor \frac{m}{k+1}\rfloor}}\, n$$

which is sublinear approximately for $\alpha < 1/(3\log_\sigma m)$.

On the other hand, if we use block addressing, we must find the exact candidate positions before verifying them with the above technique. To do this, we use the on-line version of our algorithm (i.e. [8]), which in turn finds the candidate areas and verifies using [24]. Excluding the above considered verifications, the on-line algorithm runs in $O(n)$ time . Therefore, we show under which restrictions a sublinear part of the text is sequentially traversed. This new condition goes together with $\alpha < 1/(3\log_\sigma m)$ in the case of block addressing.

The probability of a text position matching one piece is, as explained, $(k+1)/\sigma^{\lfloor m/(k+1)\rfloor}$. Therefore, the probability of a block (of size b) being sequentially traversed is

$$1-\left(1-\frac{k+1}{\sigma^{\lfloor \frac{m}{k+1}\rfloor}}\right)^b$$

and since there are n/b blocks and traversing each one costs $O(b)$, we have that the expected amount of work to traverse blocks is n times the above expression, which is

$$n\left(1-e^{-\frac{b(k+1)}{\sigma^{\lfloor m/(k+1)\rfloor}}}\right)\left(1+O\left(k/\sigma^{\lfloor \frac{m}{k+1}\rfloor}\right)\right)$$

The above expression is sublinear approximately for $\alpha < 1/\log_\sigma(bm)$.

[1] We store just one pointer for each q-gram position. This allows to index up to 4 Gb of text. Therefore we would use more than four bytes to index longer texts. On the other hand, we are not considering here the possibility of using a compressed list of positions, which can considerably reduce the space requirements.

5 Experiments

We show experimentally the index building times and sizes for different values of q, with character and block addressing. We also show the querying effectiveness of the indices, by comparing the percentage of the query time using the index against that of using the on-line algorithm. The experimental values agree well with our analysis in terms of the error ratios and block sizes up to where the indices are useful. All the tests were run on a Sun SparcClassic with 16 Mb of RAM, running SunOS 4.1.3.

For the tests we use a collection of 8.84 Mb of English literary text, filtered to lower-case and with all separators converted to a single space. We test the cases $q = 3..5$, as well as character addressing and block addressing with blocks of size 2 Kb to 64 K. Blocks smaller than 2 Kb were of no interest because the index size was the same as with character addressing, and larger than 64 Kb were of no interest because query times were too close to the on-line algorithm.

Figure 2 shows index build time and space overhead for different q values and block sizes. The size of the vocabulary file was 61 Kb for $q = 3$, 384 Kb for $q = 4$ and 1.55 Mb for $q = 5$, which shows a sharp increase.

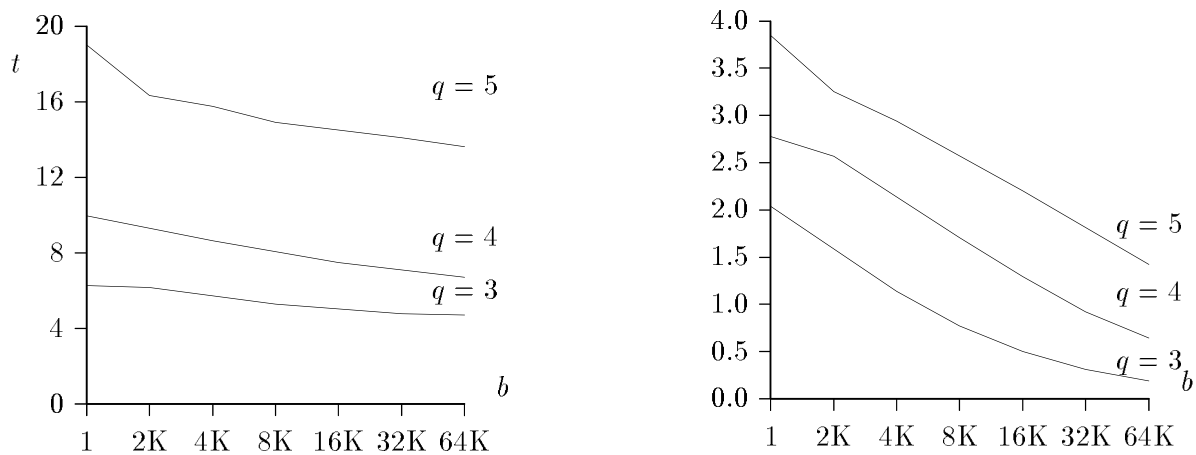

Figure 2: On the left, index construction times (minutes of user time). On the right, their space overhead (i.e. index space divided by text space).

We show now query times. We tested queries of length $m = 8$, 16 and 24. The queries are randomly chosen from the text at the beginning of non-stopwords (stopwords are words which carry no meaning and are normally not allowed in queries, such as `"a"`, `"the"`, etc.). This setup mimics common IR scenarios. For $m = 8$ we show tests with $k = 1$ and 2; for $m = 16$ with $k = 1..4$ and for $m = 24$ with $k = 1..6$. Every data point was obtained by averaging Unix's user time over 100 random trials.

Figure 3 shows query times as a percentage of the on-line algorithm. For lack of space we do not include the percentage of traversed text, which in principle is proportional to the data we show. However, the overhead of manipulating the index is high and we prefer to give the more realistic figures. The overhead of managing the index makes it better to use the on-line algorithm when the filtration efficiency of the index is not good (moreover, the indices with larger b become better because the overhead is less and the verifications are the same). In the character addressing index, this happens for $\alpha > 1/4$. Up to that point, the search times are under 10 seconds. The block addressing indices, on the other hand, cease to be useful too soon, namely for $\alpha > 1/8$.

6 Conclusions and Future Work

We have described a practical indexing scheme especially suited for IR and capable of retrieving any sequence matching a pattern with a given maximum number of errors. It is based on storing all text q-grams in the index together with their occurrences. Querying is performed by searching in the index pieces of the pattern and verifying the candidate positions. A variant pointing to blocks instead of exact positions is described too. We analyze and experimentally test our approach.

The experiments show that the scheme is practical when the pointers point to exact occurrences. The value q may be between 3 and 5, giving a tradeoff between index space and query performance. Depending on q and for a reasonable error level ($\alpha \leq 1/4$ in English text), querying the index takes 20% to 60% of the time of the on-line algorithm. The space overhead depends on q and is between two and four times the text size.

Pattern pieces longer than q are truncated. This loses part of the information on the pattern. This case could justify the approach of [21] of splitting the pattern in more pieces and forcing more than one piece to match before verifying. Extending the scheme to matching more than one piece reduces the number of verifications but leads to a more complex algorithm, whose costs may outweight the gains of less verifications. We are currently studying this issue, as well as improved pattern splitting heuristics.

Acknowledgements

We thank the useful comments and careful reading of an anonymous referee.

References

[1] M. Araújo, G. Navarro, and N. Ziviani. Large text searching allowing errors. In *Proc. 4th South American Workshop on String Processing, WSP'97*, 1997. Valparaíso, Chile. To appear.

[2] R. Baeza-Yates. Space-time trade-offs in text retrieval. In *Proc. WSP'93*, pages 15–21, 1993.

[3] R. Baeza-Yates and G. Navarro. A fast heuristic for approximate string matching. In *Proc. WSP'96*, pages 47–63. Carleton University Press, 1996.

[4] R. Baeza-Yates and G. Navarro. A faster algorithm for approximate string matching. In *Proc. CPM'96*, LNCS 1075, pages 1–23, 1996.

[5] R. Baeza-Yates and G. Navarro. Block-addressing indices for approximate text retrieval. In *Proc. CIKM'97*, 1997. Las Vegas, Nevada, Nov 11-15. To appear.

[6] R. Baeza-Yates and G. Navarro. Multiple approximate string matching. In *Proc. WADS'97*, LNCS 1272, pages 174–184, 1997.

[7] R. Baeza-Yates, G. Navarro, E. Sutinen, and J. Tarhio. Indices for approximate information retrieval. Technical Report TR/DCC-97-2, Dept. of Computer Science, Univ. of Chile, 1997.

[8] R. Baeza-Yates and C. Perleberg. Fast and practical approximate pattern matching. In *Proc. CPM'92*, pages 185–192, 1992. LNCS 644.

[9] W. Chang and J. Lampe. Theoretical and empirical comparisons of approximate string matching algorithms. In *Proc. CPM'92*, pages 172–181, 1992. LNCS 644.

[10] A. Cobbs. Fast approximate matching using suffix trees. In *Proc. CPM'95*, pages 41–54, 1995.

[11] G. Gonnet. A tutorial introduction to Computational Biochemistry using Darwin. Technical report, Informatik E.T.H., Zurich, Switzerland, 1992.

[12] P. Jokinen and E. Ukkonen. Two algorithms for approximate string matching in static texts. In *Proc. MFCS'91*, volume 16, pages 240–248, 1991.

[13] J. Kärkkäinen and E. Sutinen. Lempel-Ziv index for q-grams. In *Proc. ESA'96*, pages 378–391, 1996. LNCS 1136.

[14] J. Kärkkäinen and E. Ukkonen. Lempel-Ziv parsing and sublinear-size index structures for string matching. In *Proc. WSP'96*, pages 141–155. Carleton University Press, 1996.

[15] R. Karp and M. Rabin. Efficient randomized pattern-matching algorithms. *IBM J. Res. Development*, 31(2):249–260, March 1987.

[16] O. Lehtinen, E. Sutinen, and J. Tarhio. Experiments on block indexing. In *Proc. WSP'96*, pages 183–193. Carleton University Press, 1996.

[17] U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. In *ACM-SIAM Symposium on Discrete Algorithms*, pages 319–327, 1990.

[18] U. Manber and S. Wu. GLIMPSE: A tool to search through entire file systems. Technical Report 93-34, Dept. of CS, Univ. of Arizona, Oct 1993.

[19] E. Myers. A sublinear algorithm for approximate keyword searching. *Algorithmica*, 12(4/5):345–374, Oct/Nov 1994.

[20] P. Sellers. The theory and computation of evolutionary distances: pattern recognition. *J. of Algorithms*, 1:359–373, 1980.

[21] F. Shi. Fast approximate string matching with q-blocks sequences. In *Proc. WSP'96*, pages 257–271, 1996.

[22] E. Sutinen and J. Tarhio. On using q-gram locations in approximate string matching. In *Proc. ESA'95*, 1995. LNCS 979.

[23] E. Sutinen and J. Tarhio. Filtration with q-samples in approximate string matching. In *Proc. CPM'96*, pages 50–61, 1996.

[24] E. Ukkonen. Finding approximate patterns in strings. *J. of Algorithms*, 6:132–137, 1985.

[25] E. Ukkonen. Approximate string matching over suffix trees. In *Proc. CPM'93*, pages 228–242, 1993.

[26] E. Ukkonen. Constructing suffix trees on-line in linear time. *Algorithmica*, 14(3):249–260, Sep 1995.

[27] S. Wu and U. Manber. Fast text searching allowing errors. *CACM*, 35(10):83–91, 1992.

[28] S. Wu, U. Manber, and E. Myers. A sub-quadratic algorithm for approximate limited expression matching. *Algorithmica*, 15(1):50–67, 1996.

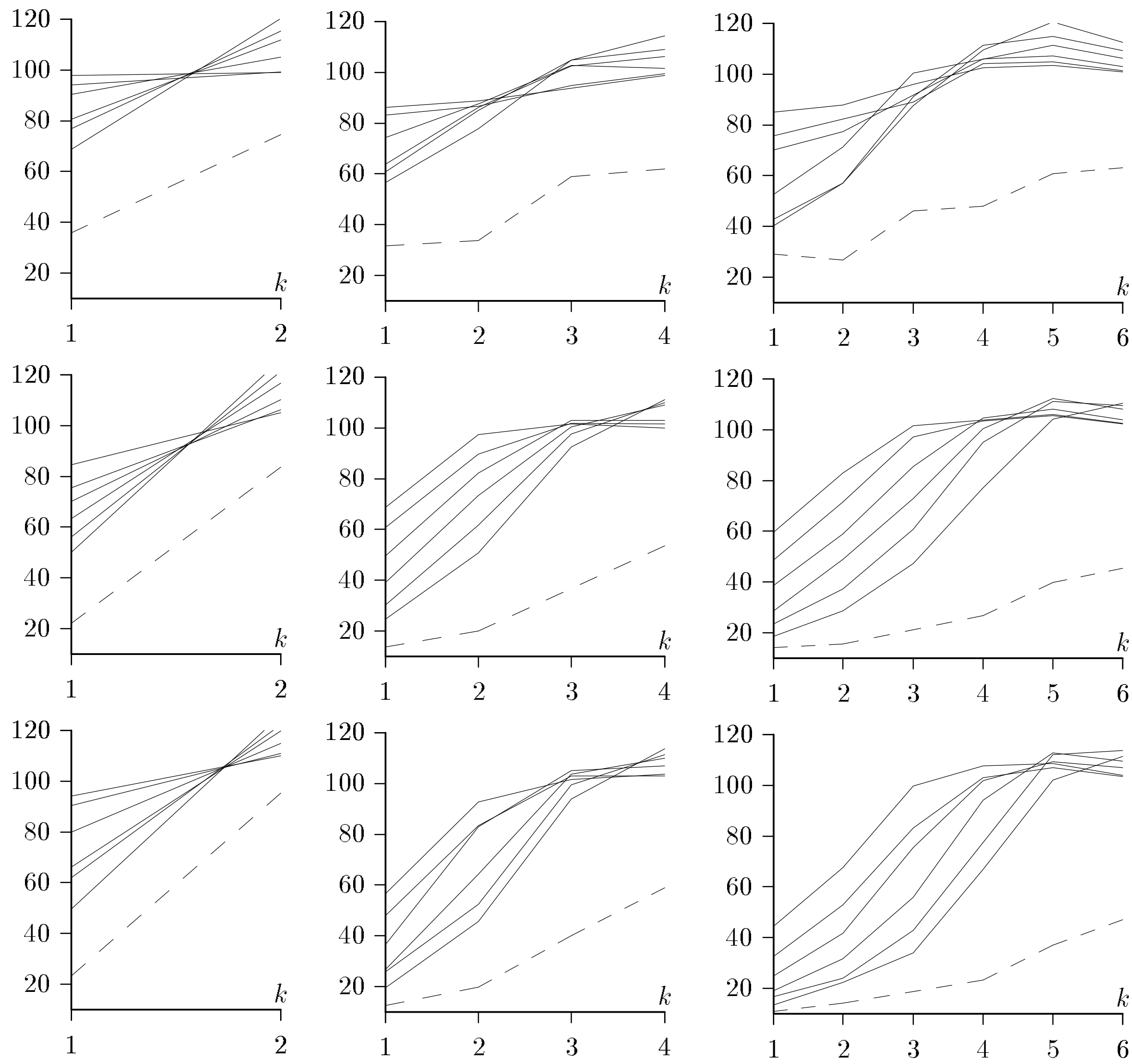

Figure 3: Query time using the index divided by query time using the on-line algorithm (percentage). The rows correspond to $q = 3$, 4 and 5. The columns correspond to $m = 8$, 16 and 24. The dashed line corresponds to character addressing, full lines to block addressing. From lower to upper (at $k = 1$) they correspond to $b = 2$, 4, 8, 16, 32 and 64 Kb.