Indexing Methods for Approximate Text Retrieval
(Extended Abstract)

Ricardo Baeza-Yates*
Gonzalo Navarro*
Erkki Sutinen†
Jorma Tarhio‡

Abstract

While the problem of on-line approximate string matching is well studied, only recently the first off-line indexing techniques have emerged. We study the different indexing mechanisms for this problem, proposing a taxonomy to classify them. We also propose and analyze two new techniques which are adaptations of recent on-line algorithms. For the final version we plan to experimentally compare all the algorithms in terms of index construction time, space overhead, query efficiency and tolerance to errors, determining the best compromises for each case.

1 Introduction

Approximate string matching is a recurrent problem in many branches of computer science, with applications to text searching, computational biology, pattern recognition, signal processing, etc.

The problem can be stated 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.

In the on-line version of the problem, the pattern can be preprocessed but the text cannot. The classical solution uses dynamic programming and is $O(mn)$ time [31, 30]. Later, a number of algorithms improved this to $O(kn)$ time in the worst case or even less on average, by using cleverly the properties of the dynamic programming matrix (e.g. [12, 22, 36, 9]) or parallelizing the computation in the bits of computer words (e.g. [43, 41, 44]). Recently, an $O(n)$ worst-case time algorithm for short patterns based on bit-parallelism was presented in [5].

Another trend is that of "filtration" algorithms: a fast filter is run over the text quickly discarding uninteresting parts. The interesting parts are later verified with a more expensive algorithm. Examples of filtration approaches are [10, 35, 33, 8, 29]. Some are "sublinear" in the sense that they do not inspect all the text characters, but the on-line problem is $\Omega(n)$ if m is taken as constant.

If the text is large and has to be searched frequently, even the fastest on-line algorithms are not practical, and preprocessing the text becomes necessary. Therefore, many indexing methods have been developed for exact string matching [40]. However, only a few years ago, indexing text for approximate string matching was considered one of the main open problems in this area [43, 2].

*Dept. of Computer Science, University of Chile. This work has been supported in part by Fondecyt grants 1950622 and 1960881.

†Dept. of Computer Science, University of Helsinki, Finland. This work was supported by the Academy of Finland.

‡Dept. of Computer Science, University of Joensuu, Finland.

In this paper we survey indexes for approximate text search. We cover in detail the known approaches and propose and analyze two new indexing mechanisms, based on extensions of [5]. For the final version we plan to experimentally compare all the techniques in terms of indexing times, space overhead, querying times and tolerance to errors, pointing out as a conclusion which are the best indexes under different circumstances.

2 Taxonomy Outline

There are two types of indexing mechanisms: word-oriented and sequence-oriented. In the first one, more oriented to natural language text and information retrieval, the index can retrieve 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 will retrieve every *sequence*, without notion of word separation.

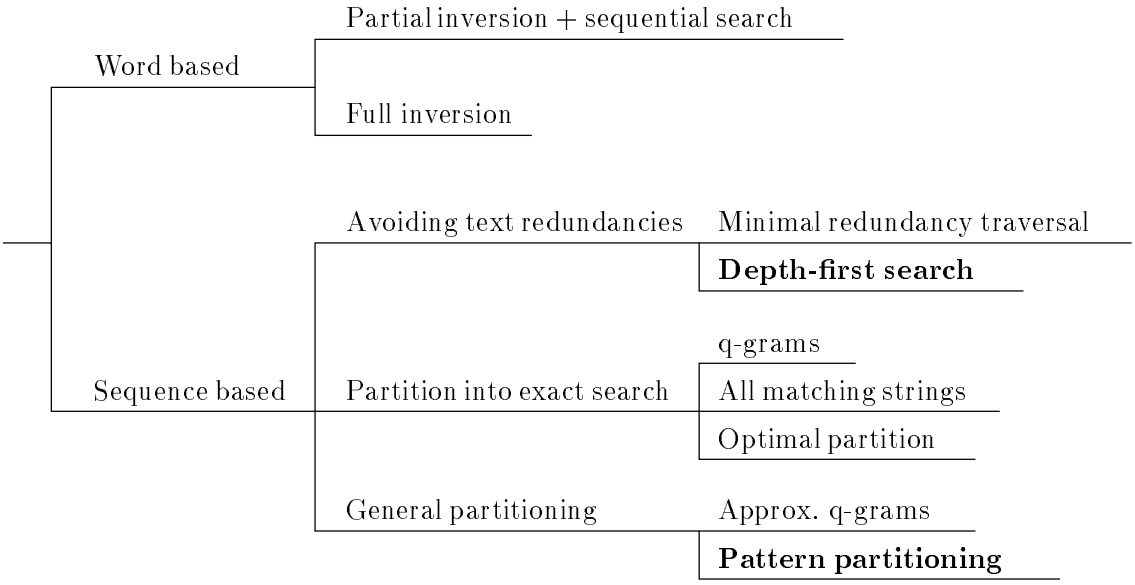

Figure 1: Taxonomy of indexes for approximate string searching (categories where we propose new algorithms are highlighted).

Current word-oriented indexes solve the problem by using a classical on-line algorithm on the set of words (i.e. the vocabulary), thus obtaining the set of words to retrieve [25, 7, 1]. The rest may proceed without using approximate matching. Since the vocabulary is sublinear in size with respect to the text, they can achieve good performance. These indexes are not capable of retrieving an occurrence that is not a sequence of words. However, this is in many cases exactly what is wanted.

The problem becomes more complex if words are not used. This is the case, for example, of genetic databases (DNA or proteins). The solutions in this case fall into three classes.

One class of algorithms is based on searching the complete pattern allowing k errors, in a way that avoids the repetitions of the text. They typically traverse the suffix tree [21] of the text instead of the text itself [38, 11, 13]. A second class reduces the problem to exact matching of substrings of

the pattern, and uses an index that retrieves exact substrings [17, 34, 23, 28, 6, 20]. The third class also takes pattern substrings, but not as many to reduce the problem to exact matching, and hence approximate search of the substrings is necessary, using some of the above indexes [34]. There are a number of proposals to reduce the space requirements of many of these indexes [27, 7, 19, 18].

Different indexes involve different trade-offs between indexing time, space overhead, querying time and tolerance to errors. For example, those of the second class tend to use smaller indices but can be used with a low number of errors. We focus on the more powerful sequence oriented indexes in this work, since the others only solve a simplification of the original problem.

Before presenting in detail the different indexing schemes, we give some general concepts. We use the standard notation for strings over a finite alphabet of size σ. We call a q-gram any string of length q. We use $d(a,b)$ to denote the edit (Levenshtein) distance between two strings, which is the minimal number of insertions, deletions and substitutions needed to transform a into b. We denote by w the word size of the computer used, which is commonly assumed to be $O(\log n)$. We use $0 < \alpha < 1$ as the error ratio k/m. Most analysis assume uniformly distributed text.

Many approaches use what we call the *partitioning lemma*. Its most general form, proved in [5], establishes that if a string matches a pattern, then we can partition the pattern in j pieces as we like and some piece will appear in the string with at most $\lfloor k/j \rfloor$ errors. The particular case of $j = k + 1$ (reducing to exact matching) was proved before [43, 8]. Anther particular case appears in [28].

3 Word-Oriented Indexes

3.1 Partial Inversion plus Sequential Search

The first proposal for a word-oriented index was due to Manber and Wu [25]. In a very practical approach, they propose a scheme based on a modified inverted file and sequential approximate search.

The index structure is as follows: the text is logically divided into "blocks". The index stores all the different words of the text. For each word, the list of the blocks where the word appears is kept.

To search a word allowing errors, an on-line approximate search algorithm (in this case, Agrep [42]) is run over the index of words. Then, for every block where a matching word is present, a new sequential search is performed over that block (using Agrep again).

The idea of using blocks makes the index small, at the cost of having to traverse parts of the text sequentially. The index is small not only because the pointers to the blocks are small, but also because all the occurrences in a single block are referenced only once.

Glimpse uses 250–256 blocks, which works well for moderate-size texts. For larger texts, it is possible to point to files instead of blocks, or even to occurrences of words. Typical figures for the size of the index with respect to the text are: 2-4% for blocks, 10-15% for files, and 25-30% for words.

Glimpse is not only an algorithm, but a software intended to index medium-size text collections. The small index size makes it an attractive choice. It works well for texts of up to 200 Mb and moderate error ratio. In an experiment run on a DEC 5000/240, 70 Mb of text were indexed in 9 minutes and the index took 2.7% of the text. Queries were answered in 2-10 seconds, depending on the query.

Baeza-Yates and Navarro [7] propose an alternative search technique for the same index. They search the pattern in the vocabulary (using [5, 4], which is especially well suited for short patterns, like words), but once the list of matching words of each block is obtained, approximate search is not used anymore. Instead, a multiple exact pattern matching algorithm is used to search the matching words in the text blocks. Experiments show that the algorithm is five times faster than Glimpse on approximate word queries. The size of the blocks can be tuned to meet different trade-offs between index space or query time, as shown experimentally in the paper. They also show analytically that it is not possible to have an index of this type which is sublinear in size and search times simultaneously.

3.2 Full Inversion

Araújo, Navarro and Ziviani take the approach of full inversion [1]. For each word, the list of all its occurrences in the text are kept and the text is never accessed.

The search on the vocabulary is as before (using [5]), but the second phase of the search changes completely: once the matching words in the vocabulary are identified, all their lists are merged. Phrases can also be searched, by splitting them into words. The approach is much more resistant to the size of the text collection, and is shown to work well with text collections of more than 1 Gb.

The index is built in a single pass over the text, and in linear time. The construction proceeds in-place, in the sense that the space requirement to build the index is that of the final index.

The analysis shows that, under the assumption that the vocabulary size is $O(n^\beta)$ for $\beta \approx 0.5$ (which is validated in that work and in previous ones [15]), the retrieval costs are near $O(\sqrt{n})$ for useful searches (i.e. those with reasonable precision).

Although the space requirements of this index are similar to those of inverted lists (i.e. 30-40%), an applicable word compression scheme is presented in [27], which not only reduces overall space usage to 50% in practice but also improves indexing and search times.

Experiments run on a Sun SparcStation 4 with 128 Mb RAM show an indexing performance of 4 Mb per minute and a space overhead of 35% (excluding stop-words). A 1 Gb text collection can be searched for single word queries in nearly 2 seconds for $k \leq 2$. Even using pointers to words, Glimpse does not work well with such large texts.

4 Sequence-Oriented Indexes

4.1 Avoiding Text Redundancies

We describe here the algorithms that avoid repetition of the same work due to the same text. They use a suffix tree as a device to factor the redundancies present in the text.

4.1.1 Minimal Redundancy Traversal

This technique is based on simulating a sequential algorithm, but running it on the suffix tree of the text instead of the text itself. Since every different substring in the text is represented by a single

node in the suffix tree, it is possible to avoid the repetitions that occur in the text.

The first papers on these lines were due to Jokinen and Ukkonen [17, 38]. Later, Cobbs [11] improved their results. The algorithm is based on the fact that the state of the search at a given point in the text is only influenced by the last characters read ($m + k$ or less). They call "viable prefixes" the substrings that can be prefixes of an approximate occurrence of the pattern. The algorithms traverse in the suffix tree all the different viable prefixes, simulating the dynamic programming algorithm.

A suffix tree can be built in linear time with respect to the text size, given enough main memory [26, 39]. In the improved version [11], the complexity of the search is $O(mQ)$ plus the size of the output, where Q is the number of distinct viable prefixes in the text ($Q \leq n$). In [38], a worst-case analysis shows that $Q = O(\min(n, m^{k+1}\sigma^k))$, where σ is the alphabet size. If k is small, $Q << n$.

A weak point in this scheme is the large space needed by suffix trees. This is important not only by itself, but also because linear time construction is not practical if the suffix tree does not fit in main memory. A suffix tree indexing every character typically takes up a space of $12n$ bytes.

This can be partially overcome by using compression: in [19], a technique is proposed which obtains an index of size $O(nH/\log n)$, where $H = O(1)$ is the entropy of the text. They show experiments on natural language where the space requirements are $2.5n$ bytes. This is much better, although still insufficient to index in main memory for medium-size texts. The search time is $O(\sqrt{nmH \log m/\log n})$.

4.1.2 Depth-First Search over the Tree

A different algorithm over suffix trees is due to Gonnet [13]. In this case, the search method is simpler. Since every substring of the text (i.e. every potential occurrence) starts at the root of the suffix tree, it is sufficient to explore every path starting at the root, descending by every branch up to where it can be seen that that branch does not represent an occurrence of the pattern.

As is, this algorithm is not better than those of minimal redundancy, since it explores more nodes. However, its simplicity makes it suitable for a number of enhancements. For example, at an additional $O(\log n)$ query time factor, it can be run over suffix arrays [24, 14]. A suffix array can be built in $O(n \log n)$ time if it fits in main memory, and in $O(n^2 \log n/M)$ otherwise (where M is the size of main memory). It takes up only $4n$ bytes.

We propose now a variation on this indexing scheme, which is only possible because of the mentioned simplicity. Instead of using dynamic programming over the suffix tree, we use [5]. This on-line algorithm uses bit parallelism to simulate an automaton that recognizes the approximate pattern, and achieves linear time for small patterns in the worst case. If the pattern is long, the automaton is partitioned as in [5]. This technique can be seen as a particular case of general automaton searching over a trie [3]. However, in this case the automaton is nondeterministic and converting it to deterministic is not practical, since it tends to generate large automata.

For lack of space we do not include the analysis of this technique in this extended abstract version. The analysis shows the limit on α up to where the number of inspected nodes is sublinear in n. If $\log_\sigma n \leq m$, it is $\alpha < (1 - e/\sqrt{\sigma})(\log_\sigma n)/m$. Otherwise, it is $\alpha < (\log_\sigma n)/m - 1$ or $\alpha < 1 - e/\sqrt{\sigma}$. The cost to inspect a node is $O(1)$ for small patterns (i.e. $(m-k)(k+2) \leq w$), while in general it is $O(k(m-k)/w)$. In the original scheme [13], the cost to inspect a node is $O(m)$.

4.2 Partitioning into Exact Search

This kind of algorithms use basically a traditional index, capable of exact retrieval only. This makes them suitable for integration with other information retrieval systems. They manage to partition the approximate problem into smaller exact searching problems.

4.2.1 Exact q-Grams

Jokinen and Ukkonen [17] observed that if an approximate match of P with at most k errors ends at position j of text T, then at least $m+1-(k+1)q$ q-grams of P occur in $T[j-m+1, j]$. $T[j-m+1, j]$ includes $m-q+1$ q-grams, of which at most kq get broken in k edit operations. They divide T into two layers of consecutive, non-overlapping blocks of length $2(m-1)$. Then, the number of pattern q-grams in each block is counted. For each block with at least $m+1-(k+1)q$ pattern q-grams, the respective text area is examined using dynamic programming.

Holsti and Sutinen [16] apply the filtration condition of [17], strengthened by using the fact that a preserved pattern q-gram cannot move more than k positions from its original position. This method does not use the previous block-oriented scheme, but a window-oriented approach: each occurrence of a pattern q-gram in T marks a corresponding window where an occurrence might be located.

Sutinen and Tarhio [33] give another way to utilize the relative order of the preserved q-grams. The idea is based on observing a sequence of q-samples, i.e. non-overlapping q-grams of the text at fixed periods of length h. Supposing that an occurrence always includes at least $k+s$ q-samples, a precondition implying $h = \lfloor \frac{m-k-q+1}{k+s} \rfloor$, they require that at most s of the q-samples may not occur in P, and the preserved q-samples occur approximately in the same locations in P and its occurrence. This condition can be verified by utilizing pattern blocks: the pattern P is been divided into $k+s$ slightly (of order k) overlapping blocks; the text scanning phase evaluates, for each sequence of $k+s$ q-samples, the blocks of the preserved q-grams. This is used in an indexing scheme [34], whose main contribution is saving space: only every h-th q-sample of text T is stored into the index. At first sight, this seems to result in a different index for each m and k, but they can adjust s in the formula of h so that the index, precomputed according to a fixed h, can be applied.

This index takes space $O(n/h \ \log(n/h))$, is built in $O(n)$ time and queried in $O(nm/\sigma^q \ (k + \log(m)/h))$ time. Compression can be used to reduce the index space: in [18] a Lempel-Ziv-related technique obtains $O(n/\log n)$ size at no query time penalty. Another choice is to point to blocks rather than exact text positions, using sequential searching on the matching blocks. This has been implemented in a system called Grampse [23], which has been shown faster than Glimpse for sequences of words (no errors) and is currently being adapted to handle approximate searching too.

4.2.2 Searching Every Matching String

This technique is proposed by Myers [28]. It uses an index where every sequence of the text up to a given length ℓ is stored, together with the list of its positions in the text. To search for a pattern of length $\leq \ell - k$, all the maximal strings whose edit distance to the pattern is at most k are generated, and each one is searched. Later, the lists are merged.

Longer patterns are split in many pieces of the required length. The partition proceeds in a binary fashion, such that at level i the pattern is left split in 2^i parts of similar length. The partitioning lemma of Section 2 shows that those parts can be searched with $\lfloor k/2^i \rfloor$ errors, provided every occurrence of every piece is verified for a complete match. Instead of simply verifying each of the substrings occurrences, the algorithm goes up level by level in the partition process, obtaining the occurrences of that level by combining those of their two children in the next level. When the pattern is long, this is better than the simple verification of each substring.

The length of the strings stored in the index is made small enough to store them as computer integers. This allows to build the index in $O(n)$ time (assuming it fits in main memory), and very quickly in practice. The strings must also be short to avoid an explosive number of them generated at search time. The space used by the index is mainly that of storing the positions of all sequences of the text.

Query complexity is shown to be $O(kn^{pow(\alpha)} \log n)$ on average, where $pow(\alpha)$ is a concave function of α satisfying $pow(0) = 0$. This is sublinear where $pow(\alpha) < 1$, which restricts the error ratios up to which the scheme is efficient. This maximum useful α increases with the alphabet size. For example, the formula shows that $\alpha_{\max}$ is 0.33 for $\sigma = 4$ and 0.56 for $\sigma = 20$. Experiments confirm those estimations, and show an improvement of various orders of magnitude over on-line algorithms.

4.2.3 Optimal Partitioning

Baeza-Yates and Navarro [6] propose a simple and practical scheme based on the partitioning lemma stated in Section 2, using the case $j = k + 1$. The pattern is split in $k + 1$ pieces, each piece is searched with no errors in an index, and each candidate is verified for a complete occurrence. This has been also used for on-line approximate search [8, 5].

Hence, the index stores all the text strings up to a given length. The length is selected to make the index not very large and still have good selectivity (e.g. 3 to 5 letters can be used for natural language). Longer pieces of the search pattern are pruned.

Since in natural language a simple equal-length partition may give very bad results depending on the resulting pieces, an $O(m^2(\log n + k))$ dynamic programming algorithm is used to select the best partition. This is defined as the one that minimizes the total number of text positions to verify. The information required to exactly predict the number of verifications is available from the index. This is also useful to give the user early feedback of the precision and the time cost of the posed query.

It is possible to reduce space requirements at the cost of more expensive querying times, by pointing to blocks instead of exact occurrences. However, this has not worked well in practice.

It is shown that the search time is $O(n^\beta)$ for $\alpha < \ln \sigma / ((1 - \beta) \ln n + 3 \ln m)$. The index is built in linear time, and depending on the length of the sequences it takes from $1.5n$ to $3n$ bytes.

Recently, Shi [32] studied the possibility of splitting the pattern in $k + s$ parts instead of $k + 1$ ($s > 1$), with no optimization. Although much less verifications are triggered, the search phase is more complex, and therefore experimental results are needed to comment on the real performance improvement. Moreover, it is possible that the improvement in the numbe of verifications holds only for long patterns, which is not the typical case in text retrieval.

4.3 General Partitioning

We include in this section the algorithms which are intermediate between the two approaches shown before. Basically, they partition the pattern, but not up to the point in which the problem is reduced to exact matching. The result is a set of subpatterns that are to be searched with less errors. Their occurrences are to be verified to check for complete matches. We describe here the previous algorithms of this kind, and propose a new one, based on [5].

4.3.1 Approximate q-Grams

A filtration condition can be based on locating approximate matches of pattern q-grams in the text. This leads to a filtration tolerating higher error ratio as compared to the methods applying exact q-grams: a single error in the occurrence still qualifies as a match of the q-gram.

There are few filtration schemes utilizing approximate q-grams. Chang and Marr [10] have developed a dynamic algorithm, which is based on best match distances between the scanned text q-grams and the searched pattern. The algorithm scans consecutive, non-overlapping text q-grams and maintains a cumulative sum of best match distances. If a sufficient number of q-grams have been scanned before the cumulative sum exceeding k, the corresponding area is checked by an accurate method.

Sutinen and Tarhio [34] apply the best match distance in a stricter way than [10]. In addition, they do not scan all but only every h-th q-sample of the text. Here, h depends on the number r of q-samples whose best match distances to the corresponding pattern blocks are evaluated at the same time: $q \leq h \leq \lfloor \frac{m-k-q+1}{r} \rfloor$. The corresponding text area is examined only if the cumulative best match distance for r consecutive q-samples is at most k.

An index of q-grams can be used in two ways in this case: either *inclusively*, to locate all the q-samples which are close enough to the pattern q-grams, or *exclusively*, to find out the q-samples with a sufficient distance from the original pattern. Since the filtration is based on the cumulative resemblance of the text q-samples with the corresponding blocks, the goal in the inclusive approach is to find as few q-samples as possible, while the goal of the exclusive approach is the opposite. The query time complexity of this scheme is $O(nm \log(k+q)/(q^2 w))$.

4.3.2 Pattern Partitioning

We present here a new algorithm based on the partitioning lemma given in Section 2. It is a static variation of an on-line version presented in [5].

We divide the pattern in j pieces, such that each piece can be searched with the simple (non-partitioned) automaton of [5]. Then we search in the suffix tree of the text the j pieces using the algorithm we proposed in Section 4.1.2. Finally, we collect all the nodes found and verify their text positions for an occurrence of the complete pattern.

In [5], it is shown that $j = O((m-k)/\sqrt{w})$ and that the number of verifications is sublinear in n for $\alpha_1 < 1 - e/\sqrt{\sigma}\ m^{f(w,\alpha_1)}$, where $f(w,\alpha) = j/(m-k) = 1 + \sqrt{1 + w\alpha/(1-\alpha)}/w$. In practice, that means a moderate error ratio (e.g. near 0.5 for $m = 20$, $\sigma = 30$).

Since we perform j searches of the same kind of Section 4.1.2, the same analysis holds considering

that each node is visited j times at $O(1)$ cost, and therefore the cost of inspecting each node is $O((m-k)/\sqrt{w})$. To achieve search sublinearity, apart from the considerations of Section 4.1.2, we pose also the requirement that the total number of verifications must be sublinear (no verifications are required in Section 4.1.2). Thus, we also require $\alpha < \alpha_1$.

5 Discussion and Future Work

We have pointed out the importance of indexing schemes for approximate text retrieval, and discussed all the techniques we are aware of, presenting also their theoretical analysis. We focus more on sequence-oriented indexes, since word-oriented ones solve only a simplification of the problem. We arranged all the approaches in a taxonomy, and proposed and analyzed two new indexing schemes.

Since comparing the different approaches based only on their theoretical analysis is difficult and probably not accurate, our aim for the final version is to complete this work with an extensive experimental comparison among all the sequence-oriented indexes. This comparison will be carried our for different kinds of text databases (e.g. random, DNA or proteins, natural language, etc.) and will measure: index construction times, index space overhead, query execution times and tolerance to errors (i.e. the maximum α value under which the scheme works well). This data will provide us information to compare the different approaches in practice and to find out which is the best option depending on the different circumstances.

References

[1] M. Araújo, G. Navarro, and N. Ziviani. Large text searching allowing errors. Technical report, Dept. of CS, Univ. Federal de Minas Gerais, Brazil, 1996.

[2] R. Baeza-Yates. Text retrieval: Theory and practice. In *12th IFIP World Computer Congress*, volume I, pages 465–476. Elsevier Science, Sep 1992.

[3] R. Baeza-Yates and G. Gonnet. Fast text searching for regular expressions or automaton searching on a trie. *J. of the ACM*, 43, 1996.

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

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

[6] R. Baeza-Yates and G. Navarro. Practical indices for approximate string matching. Technical report, Dept. of CS, Univ. of Chile, 1996.

[7] R. Baeza-Yates and G. Navarro. Block-addressing indices for approximate text retrieval. Submitted for publication, 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] W. Chang and T. Marr. Approximate string matching and local similarity. In *Proc. CPM'94*, pages 259–273, 1994.

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

[12] Z. Galil and K. Park. An improved algorithm for approximate string matching. *SIAM J. of Computing*, 19(6):989–999, 1990.

[13] G. Gonnet. A tutorial introduction to Computational Biochemistry using Darwin. Technical report, Informatik E.T.H., Zuerich, Switzerland, 1992. DFS over the suffix tree.

[14] G. Gonnet, R. Baeza-Yates, and T. Snider. *Information Retrieval: Data Structures and Algorithms*, chapter 3: New indices for text: Pat trees and Pat arrays, pages 66–82. Prentice-Hall, 1992.

[15] J. Heaps. *Information Retrieval - Computational and Theoretical Aspects.* Academic Press, NY, 1978.

[16] N. Holsti and E. Sutinen. Approximate string matching using q-gram places. In *Proc. 7th Finnish Symposium on Computer Science*, pages 23–32. Univ. of Joensuu, 1994.

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

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

[19] 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.

[20] J. Kim and J. Shawe-Taylor. An approximate string-matching algorithm. *Theoretical Computer Science*, 92:107–117, 1992.

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

[22] G. Landau and U. Vishkin. Fast string matching with k differences. *J. of Computer Systems Science*, 37:63–78, 1988.

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

[24] 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.

[25] 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.

[26] E. McCreight. A space-economical suffix tree construction algorithm. *J. of the ACM*, 23(2):262–272, Apr 1976.

[27] E. Moura, G. Navarro, and N. Ziviani. Indexing compressed text. Technical report, Dept. of CS, Univ. Federal de Minas Gerais, Brazil, 1996.

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

[29] G. Navarro. Approximate string matching by counting. Submitted for publication., 1996.

[30] S. Needleman and C. Wunsch. A general method applicable to the search for similarities in the amino acid sequences of two proteins. *J. of Molecular Biology*, 48:444–453, 1970.

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

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

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

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

[35] J. Tarhio and E. Ukkonen. Approximate Boyer-Moore string matching. *SIAM Journal on Computing*, 22(2):243–260, 1993.

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

[37] E. Ukkonen. Approximate string matching with q-grams and maximal matches. *Theoretical Computer Science*, 1:191–211, 1992.

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

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

[40] I. Witten, A. Moffat, and T. Bell. *Managing Gigabytes*. Van Nostrand Reinhold, New York, 1994.

[41] A. Wright. Approximate string matching using within-word parallelism. *Software Practice and Experience*, 24(4):337–362, Apr 1994.

[42] S. Wu and U. Manber. Agrep – a fast approximate pattern-matching tool. In *Proc. USENIX*, pages 153–162, 1992.

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

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