Indexing Methods for Approximate String Matching

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

Abstract

Indexing for approximate text searching is a novel problem that has received significant attention because of its applications in signal processing, computational biology, and text retrieval, to name a few. We classify most indexing methods in a taxonomy that helps understand their essential features. We show that the existing methods, rather than being completely different as they are often regarded to be, form a range of solutions, whose optimum is usually found somewhere in between the two extremes.

1 Introduction

Approximate string matching is about finding a pattern in a text where the pattern, the text, or both have suffered some kind of undesirable corruption. This has a number of applications, such as retrieving musical passages similar to a sample, finding DNA subsequences after possible mutations, or searching text under the presence of typing or spelling errors.

The problem of *approximate string matching* is formally stated as follows: given a long text $T_{1...n}$ of length n and a comparatively short pattern $P_{1...m}$ of length m, both sequences over an alphabet Σ of size σ, find the text positions that match the pattern with at most k "errors".

Among the many existing error models we focus on the popular *Levenshtein* or *edit distance*, where an error is a character insertion, deletion or substitution. That is, the distance $d(x, y)$ between two strings x and y is the minimum number of such errors needed to convert one into the other, and we seek for text substrings that are at distance k or less from the pattern. Most of the techniques can be easily adapted to other error models. We use $\alpha = k/m$ as the error ratio, so $0 < \alpha < 1$.

There are numerous solutions to the *on-line* version of the problem, where the pattern is preprocessed but the text is not [15]. They range from the classical $O(mn)$ worst-case time to the optimal $O((k + \log_\sigma m)n/m)$ average case time. Although very fast on-line algorithms exist, many applications handle so large texts that no on-line algorithm can provide acceptable performance.

An alternative approach when the text is large and searched frequently is to preprocess it: build a data structure on the text (an *index*) beforehand and use it to speed up searches. Many such *indexing methods* have been developed for *exact* string matching [1], but only one decade ago doing the same for *approximate* string matching was an open problem [2].

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

*Dept. of Computer Science, University of Chile. Supported in part by Fondecyt grant 1990627.
†Dept. of Computer Science, University of Joensuu, Finland.
‡Dept. of Computer Science and Engineering, Helsinki University of Technology, Finland.

During the last decade, several proposals to index a text to speed up approximate searches have been presented. No attempt has been made up to now to show them under a common light. This is our purpose. We classify the existing approaches along two dimensions: data structure and search method.

Four different data structures are used in the literature. They all serve roughly the same purposes but present different space/time tradeoffs. We mention them from more to less powerful and space demanding. *Suffix trees* permit searching for any substring of the text. *Suffix arrays* permit the same operations but are slightly slower. Q*-grams* permit searching for any text substring not longer than q. Q*-samples* permit the same but only for some text substrings.

On the other hand, there are three search approaches. *Neighborhood generation* generates and searches for, using an index, all the strings that are at distance k or less from the pattern (their neighborhood). *Partitioning into exact searching* selects pattern substrings that must appear unaltered in any approximate occurrence, uses the index to search for those substrings, and checks the text areas surrounding them. Assuming that the errors occur in the pattern or in the text leads to radically different approaches. *Intermediate partitioning* extracts substrings from the pattern that are searched for allowing fewer errors using neighborhood generation. Again we can consider that errors occur in the pattern or in the text.

Table 1 illustrates this classification and places the existing schemes in context.

Data Structure	Search Approach				
	Neighborhood Generation	Partitioning into Exact Searching		Intermediate Partitioning	
		Errors in Text	Errors in Pattern	Errors in Text	Errors in Pattern
Suffix Tree	[10] Jokinen & Ukkonen 91 [24] Ukkonen 93 [5] Cobbs 95		[19] Shi 96		
Suffix Array	[7] Gonnet 88				[17] Navarro & Baeza-Yates 99
Q-grams	n/a	[10] Jokinen & Ukkonen 91 [9] Holsti & Sutinen 94	[16] Navarro & Baeza-Yates 97		[14] Myers 90
Q-samples	n/a	[21] Sutinen & Tarhio 96	n/a	[18] Navarro et al. 2000	n/a

Table 1: Taxonomy of indexes for approximate text searching. A "n/a" means that the data structure is unsuitable to implement that search approach because not enough information is maintained.

2 Basic Concepts

2.1 Suffix Trees

Suffix trees [1] are widely used data structures for text processing. Any position i in a text T defines automatically a *suffix* of T, namely $T_{i...}$. A *suffix trie* is a trie data structure built over all the suffixes of T. Each leaf node points to a suffix. Each internal node represents a unique substring of T that appears more than once. Every substring of T can be found by traversing a path from the root, possibly continuing the search directly in the text if a leaf is reached. In practice a *suffix tree*, obtained by compressing unary trie paths, is preferred because it yields $O(n)$ space and $O(n)$ construction time [12, 25] and offers the same functionality. Figure 1 illustrates a suffix trie.

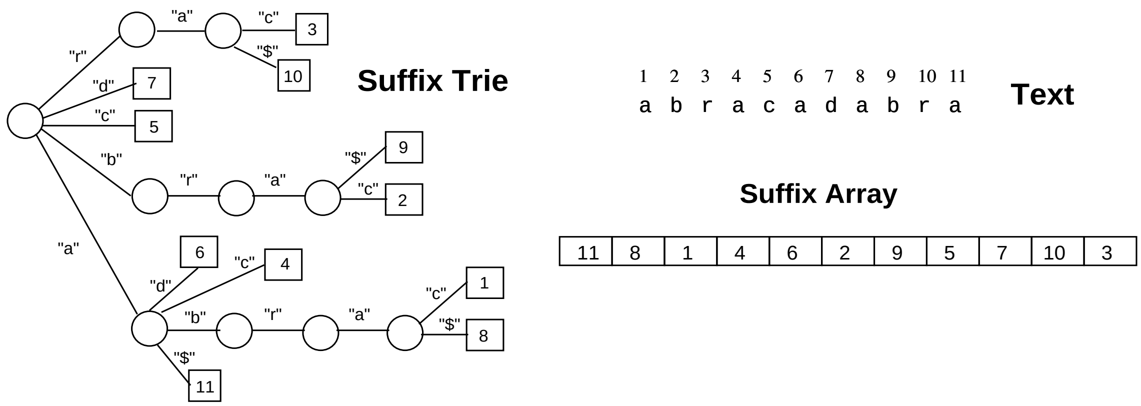


Figure 1: The suffix trie and suffix array for a sample text. The “$” is a special marker to denote the end of the text and is lexicographically smaller than the other characters.

To search for a simple pattern in the suffix trie, we just enter it driven by the letters of the pattern, reporting all the suffix start points in the subtree of the node we arrive at, if any. E.g., consider searching for `"abr"` in the example. So the search time is the optimal $O(m)$. A weak point of the suffix tree is its large space requirement, worsened by the absence of practical schemes to manage it in secondary memory. Among the many attempts to reduce this space, the best practical implementations still require about 9 times the text size [6] and do not handle well secondary memory.

2.2 Suffix Arrays

The suffix array [11, 8] is a weak version of the suffix tree, which requires much less space (one pointer per text position, i.e., about 4 times the text size) and poses a small penalty over the search time.

If the leaves of the suffix tree are traversed in left-to-right order, all the text suffixes are retrieved in lexicographical order. A suffix array is simply such an ordered array containing all the pointers to the text suffixes. Figure 1 illustrates this.

The suffix array can be built directly in $O(n \log n)$ worst case time and $O(n \log \log n)$ average time [11]. For secondary memory, a more practical $O(n^2 \log M \ / \ M)$ time algorithm [8] is preferable, where M is the amount of main memory available.

Suffix arrays can simulate by binary searching almost every algorithm on suffix trees, at an $O(\log n)$ time penalty factor. This is because each suffix subtree corresponds to a suffix array interval, so moving to a child node is equivalent to reducing the current suffix array interval by doing two binary searches. For instance, exact searching for a pattern takes $O(m \log n)$ time using this approach.

2.3 Q-gram and Q-sample Indexes

Yet a weaker (and less space demanding) scheme is to limit the length of the strings that can be directly found in the index. A q-gram index allows retrieval of text strings of length at most q.

In a q-gram index, every different text q-gram (substring of length q) is stored. For each q-gram, all its positions in the text (called *occurrences*) are stored in increasing text order.

An even less space demanding alternative is a q-sample index, where only *some* text q-grams (called text q-samples) are stored, and therefore not any text q-gram can be found. The text q-samples, unlike the text q-grams, do not overlap, and there may even be some space between each pair of samples. This severely restricted index is attractive for its low space requirements, and it still permits searching for long strings, as we will see later.

A q-gram or q-sample index can be built in linear time, although for large texts a more practical $O(n \log(n/M))$ time algorithm can be used. Depending on q the index takes from 0.5 to 3 times the text size for reasonable retrieval performance.

2.4 Computing Edit Distance

The basic algorithm to compute the edit distance between two strings x and y is based on dynamic programming (see [15]). To compute $d(x, y)$ a matrix $C_{0\ldots|x|,0\ldots|y|}$ is filled, where $C_{j,i} = d(x_{1\ldots j}, y_{1\ldots i})$. This is computed as $C_{0,0} = 0$ and

$$C_{j,i} \;\; = \;\; \min(C_{j-1,i-1} + \delta(x_j, y_i), C_{j-1,i} + 1, C_{j,i-1} + 1)$$

where $\delta(a, b)$ is zero for $a = b$ and 1 otherwise, and $C_{-1,i} = C_{j,-1} = \infty$. The minimization accounts for the three allowed operations: substitutions, deletions and insertions. At the end, $C_{|x|,|y|} = d(x, y)$. The matrix is filled, e.g., column-wise to guarantee that necessary cells are already computed. The table in Figure 2 (left) illustrates this algorithm to compute $d($`"survey"`, `"surgery"`$)$.

The algorithm is $O(|x||y|)$ time in the worst and average case. The space required is only $O(|x|)$ in a column-wise processing because only the previous column must be stored to compute the new one.

3 Neighborhood Generation

3.1 The Neighborhood of the Pattern

The number of strings that match a pattern P with at most k errors is finite, as the length of any such string cannot exceed $m + k$. We call this set of strings the "k-neighborhood" of P, and denote it $U_k(P) = \{x \in \Sigma^*,\ d(x, P) \leq k\}$.

The idea of this approach is, in essence, to generate all the strings in $U_k(P)$ and use an index to search for their text occurrences (without errors). Each such string can be searched for separately, as in [14], or a more sophisticated technique can be used (see next).

The main problem with this approach is that $U_k(P)$ is quite large. Good bounds [23, 14] show an exponential growth in k, e.g., $|U_k(P)| = O(m^k \sigma^k)$ [23]. So this approach works well for small m and k.

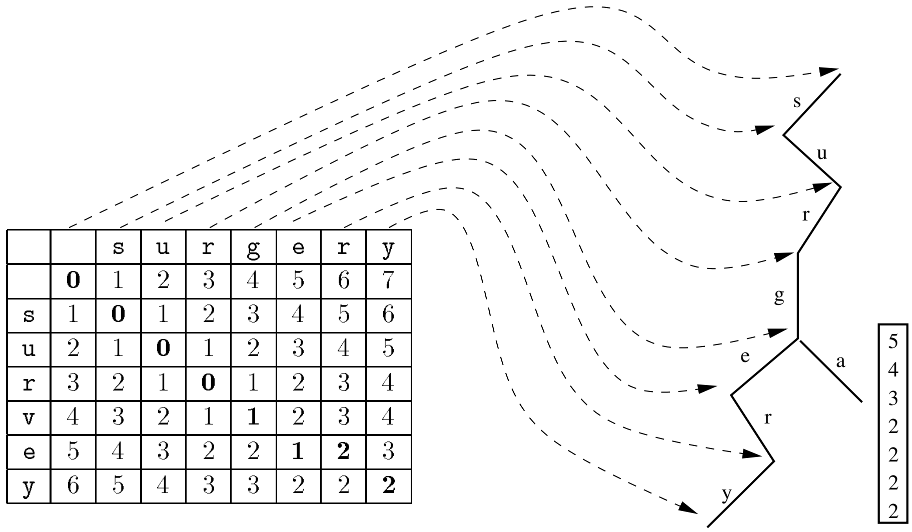
3.2 Backtracking

The suffix tree or array can be used to find all the strings in $U_k(P)$ that are present in the text [7, 24]. Since every substring of the text (i.e., every potential occurrence) can be found by traversing the suffix tree from the root, 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 cannot start a string in $U_k(P)$.

We explain the algorithm on a suffix trie. We compute the edit distance between our pattern $x = P$ and every text string y that labels a path from the root to a trie node N. We start at the root with the initial column $C_{j,root} = j$ (Section 2.4 with $i = 0$) and recursively descend by every branch of the trie. For each edge traversed we compute a new column from the previous assuming that the new character of y is that labeling the edge just traversed.

Two cases may occur at node N: (a) We may find that $C_{m,N} \leq k$, which means that $y \in U_k(P)$, and hence we report all the leaves of the current subtree as answers. (b) We may find that $C_{j,N} > k$ for every j, which means that y is not a prefix of any string in $U_k(P)$ and hence we can abandon this branch of the trie. If none of these two cases occur, we continue descending by every branch. If we arrive at a leaf node, we continue the algorithm of Section 2.4 over the text suffix pointed to.

Figure 2 illustrates the process over the path that spells out the string `"surgery"`. The matrix can be seen now as a stack (that grows to the right). With $k = 2$ the backtracking ends indeed after reading `"surge"` since that string matches the pattern (case (a)). If we had instead $k = 1$ the search would have been pruned (case (b)) after considering `"surger"`, and in the alternative path shown, after considering `"surga"`, since in both cases no entry of the matrix is ≤ 1.



		s	u	r	g	e	r	y
	0	1	2	3	4	5	6	7
s	1	**0**	1	2	3	4	5	6
u	2	1	**0**	1	2	3	4	5
r	3	2	1	**0**	1	2	3	4
v	4	3	2	1	**1**	2	3	4
e	5	4	3	2	2	**1**	**2**	3
y	6	5	4	3	3	2	2	**2**

Figure 2: The dynamic programming algorithm run over the suffix trie. We show only one path and one additional link.

Some improvements to this algorithm [10, 25, 5] avoid processing some redundant nodes at the cost of a more complex node processing, but their practicality has not been established. This method has been used also to compare a whole text against another one or against itself [3].

4 Partitioning into Exact Search

Each approximate occurrence of a pattern contains some pattern substrings that match without errors. Hence, we can derive sufficient conditions for an approximate match based on exact matching of one or more carefully selected pattern pieces. These pieces are searched for without errors, and the text areas surrounding their occurrences are verified for an approximate occurrence of the complete pattern. This technique is called "filtration" [15].

In indexed searching, some kind of index is used to quickly locate the *exact* occurrences of the selected pattern pieces, and a classical on-line algorithm is used for verification. A general limitation of filtration methods is that there is always a maximum error ratio α up to where they are useful, as for larger error levels the text areas to verify cover almost all the text.

A general lemma is useful to unify the many existing variants.

Lemma 1: Let A and B be two strings such that $d(A, B) \leq k$. Let $A = A_1 x_1 A_2 x_2 ... x_{k+s-1} A_{k+s}$, for strings A_i and x_i and for any $s \geq 1$. Then, at least s strings $A_{i_1} \ldots A_{i_s}$ appear in B. Moreover, their relative distances inside B cannot differ from those in A by more than k.

This is clear if we consider the sequence of at most k edit operations that convert A into B. As each edit operation can affect at most one of the A_i's, at least s of them must remain unaltered. The extra requirement on relative distances follows by considering that k edit operations cannot produce misalignments larger than k.

Two main branches of algorithms based on the lemma exist, differing essentially in whether the errors are assumed to occur in the pattern or in the text.

4.1 Errors in the Pattern

This technique is based on the application of Lemma 1 under the setting $P = A$, $x_i = \varepsilon$. That is, the pattern is split in $k+s$ pieces, and hence s of the pieces must appear inside any occurrence. Therefore, the $k+s$ pieces are searched for in the text and the text areas where s of those pieces appear under the stated distance requirements are verified for a complete match.

Using the data structures of Section 2 the time to search for the pieces in the index is $O(m)$ or $O(m \log n)$, but the checking time dominates. The case $s = 1$, proposed in [16], shows an average time to check the candidates of $O(m^2kn/\sigma^{m/(k+1)})$. The case $s > 1$ is proposed in [19] without any analysis. It is not clear which is better. If s grows, the pieces get shorter and hence there are more matches to check, but on the other hand, forcing s pieces to match makes the filter stricter [19].

Note that, since we cannot know where the pattern pieces can be found in the text, all the text positions must be searchable. The technique described next, instead, works on a q-sample index. A disadvantage of this smaller index is that it tolerates lower error ratios.

4.2 Errors in the Text

Assume now that the errors occur in the text, i.e., A is an occurrence of P in T. We extract substrings of length q at fixed text intervals of length $h \geq q$.

Those q-samples correspond to the A_i's of Lemma 1, and the space between q-samples to the x_i's. What the lemma ensures is that, inside any occurrence of P containing $k+s$ text q-samples, at least s of them appear in P at about the same positions ($\pm k$). Now, for the lemma to hold, we need to ensure that any occurrence of P in T contains at least $k+s$ text q-samples, i.e., $h \leq \lfloor (m-k-q+1)/(k+s) \rfloor$.

At search time, all the $m-q+1$ (overlapping) pattern q-grams are extracted and searched for in the index of text q-samples. When s pattern q-grams match in the text at the proper distances, the text area is verified for a complete match. This idea is presented in [21], and earlier versions in [10, 9, 22].

Let us discuss the best value of q. We want it to be small to avoid a very large set of different q-samples. We want it to be large to minimize the amount of verification. Some analyses [20] show that $q = \Theta(\log_\sigma n)$ is the optimal value. On the other hand, little has been said about the best s value, except that a larger s may trigger fewer verifications.

5 Intermediate Partitioning

We present now an approach that lies between the two previous ones. We filter the search by looking for pattern pieces, but those pieces are large and still may appear with errors in the occurrences. However, they appear with *fewer* errors, and therefore we use neighborhood generation to search for them. A new lemma is useful here.

Lemma 2: Let A and B be two strings such that $d(A, B) \leq k$. Let $A = A_1x_1A_2x_2...x_{j-1}A_j$, for strings A_i and x_i and for any $j \geq 1$. Let k_i be any set of nonnegative numbers such that $\sum_{i=1}^{j} k_i \geq k-j+1$. Then, at least one string A_i appears with at most k_i errors in B.

The proof is easy: if every A_i needs more than k_i errors to match in B, then the total distance cannot be less than $(k-j+1)+j = k+1$. Note that in particular we can choose $k_i = \lfloor k/j \rfloor$ for every i.

5.1 Errors in the Pattern

Search approaches based on this method have been proposed in [14, 17]. Split the pattern in j pieces, for some j that we discuss soon. Use neighborhood generation to find the text positions where those pieces appear, allowing

$\lfloor k/j \rfloor$ errors. Then, for each such text position, check with an on-line algorithm the surrounding text. The main question is now which j value to use.

In [14], the pattern is partitioned because they use a q-gram index, so they use the minimum j that gives short enough pieces (they are of length m/j). In [17] the index can search for pieces of any length, and the partitioning is done in order to optimize the search time.

Consider the evolution of the search time as j moves from 1 (neighborhood generation) to $k+1$ (partitioning into exact search). We search for j pieces of length m/j with k/j errors, so the error level α stays about the same for the subpatterns. As j moves to 1, the cost to search for the neighborhood of the pieces grows exponentially with their length, as shown in Section 3.1. As j moves to $k+1$ this cost decreases, reaching even $O(m)$ when $j = k+1$. So, to find the pieces, a larger j is better.

There is, however, the cost to verify the occurrences too. Consider a pattern that is split in j pieces, for increasing j. Start with $j = 2$. Lemma 2 states that every occurrence of the pattern involves an occurrence of at least one of its two halves (with $k/2$ errors), although there may be occurrences of the halves that yield no occurrences of the pattern. Consider now halving the halves ($j = 4$), so we have four pieces now (call them "quarters"). Each occurrence of one of the halves involves an occurrence of at least one quarter (with $k/4$ errors), but there may be many quarter occurrences that yield no occurrences of a pattern half. This shows that, as we partition the pattern in more pieces, more occurrences are triggered. Hence, the verification cost grows from zero at $j = 1$ to its maximum at $j = k+1$. The tradeoff is illustrated in Figure 3.

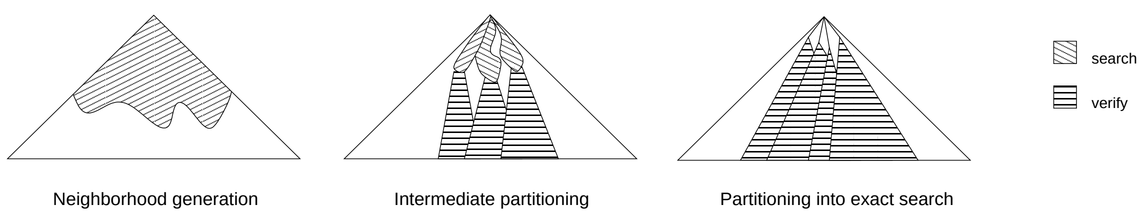


Figure 3: Intermediate partitioning can be seen as a tradeoff between neighborhood generation and partitioning into exact search.

In [17] it is shown that the optimal j is $\Theta(m/\log_\sigma n)$, yielding a time complexity of $O(n^\lambda)$, for $0 \leq \lambda \leq 1$. This is sublinear ($\lambda < 1$) for $\alpha < 1 - e/\sqrt{\sigma}$, a well known limit for any filtration approach [15] (although the e is pessimistic and is replaced by 1 in practice). Interestingly, the same results are obtained in [14] by setting $q = \Theta(\log_\sigma n)$. The experiments in [17] show that this intermediate approach is by far superior to both extremes.

5.2 Errors in the Text

This time we consider an occurrence containing a sequence of j q-samples, which must be chosen at steps of $h \leq \lfloor (m-k-q+1)/j \rfloor$. By Lemma 2, one of the q-samples must appear in the pattern with $\lfloor k/j \rfloor$ errors at most. Moreover, if *every* q-sample i appears in the pattern block $Q_i = P_{hi-k..hi+q-1+k}$ with k_i errors, then it must hold that $\sum k_i \leq k$.

This method [21, 18] searches every block Q_i in the index of q-samples using backtracking, so as to find the least number of errors to match each text q-sample *inside* Q_i, using a slight modification to the algorithm of Section 3.2. If a zone of consecutive samples is found whose errors add up to at most k, the area is verified with an on-line algorithm.

To permit efficient neighborhood searching, we need to limit the maximum error level allowed. Permitting q errors may be too expensive, as every text q-sample will be considered. Rather, we choose $q > e \geq \lfloor k/j \rfloor$ and assume that every text q-sample indeed matches with $e+1$ errors. We search the pattern blocks permitting only

e errors. Every q-sample found with $k_i \leq e$ errors changes its estimation from $e+1$ to k_i, otherwise it stays at the optimistic bound $e+1$.

There is a tradeoff here. If we use a small e value, then the search of the e-neighborhoods will be cheaper, but as we have to assume that the text q-samples not found have $e+1$ errors, some unnecessary verifications will be carried out. On the other hand, using larger e values gives more exact estimates of the actual number of errors of each text q-sample and hence reduces unnecessary verifications in exchange for a higher cost to search the e-environments.

Not enough work has been done on obtaining the optimal e. In [18] it is mentioned that, as the cost of the search grows exponentially with e, the minimal $e = \lfloor k/j \rfloor$ can be a good choice. It is also shown experimentally that the scheme tolerates higher error levels than the corresponding partitioning into exact search.

6 Conclusions

We have considered indexing mechanisms for approximate string matching, a novel and difficult problem arising in several areas. We have classified the different approaches using two coordinates: the supporting data structure and the search approach. We have shown that the most promising alternatives are those that look for an optimum balance point between exhaustively searching for neighborhoods of pattern pieces and the strictness of the filtration produced by splitting the pattern into pieces.

A separate issue not covered in this paper is indexing schemes for approximate word matching on natural language text. This is a much more mature problem with well-established solutions.

Radically innovative ideas are welcome in this area. One such idea is an approximation algorithm with worst-case performance guarantees [13]. By setting error thresholds, a certain "fuzziness" is tolerated in the results that are produced, so approximation algorithms should be acceptable for most applications. Another novel approach that is starting to receive attention (e.g., [4]) is to use the edit distance to structure the text as a metric space, so our problem is reduced to a metric-space search. More development is necessary to establish how competitive these ideas could be in practice.

References

[1] A. Apostolico and Z. Galil. *Combinatorial Algorithms on Words*. Springer-Verlag, 1985.

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

[3] R. Baeza-Yates and G. Gonnet. A fast algorithm on average for all-against-all sequence matching. In *Proc. 6th Symp. on String Processing and Information Retrieval (SPIRE'99)*. IEEE CS Press, 1999. Previous version unpublished, Dept. of Computer Science, Univ. of Chile, 1990.

[4] E. Chávez and G. Navarro. A metric index for approximate string matching. In *Proc. 5th Symp. on Latin American Theoretical Informatics (LATIN)*, 2002. Cancun, Mexico. To appear.

[5] A. Cobbs. Fast approximate matching using suffix trees. In *Proc. 6th Ann. Symp. on Combinatorial Pattern Matching (CPM'95)*, LNCS 807, pages 41–54, 1995.

[6] R. Giegerich, S. Kurtz, and J. Stoye. Efficient implementation of lazy suffix trees. In *Proc. 3rd Workshop on Algorithm Engineering (WAE'99)*, LNCS 1668, pages 30–42, 1999.

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

[8] 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.

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

[10] P. Jokinen and E. Ukkonen. Two algorithms for approximate string matching in static texts. In *Proc. 2nd Ann. Symp. on Mathematical Foundations of Computer Science (MFCS'91)*, pages 240–248, 1991.

[11] U. Manber and E. Myers. Suffix arrays: a new method for on-line string searches. *SIAM J. on Computing*, 22(5):935–948, 1993.

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

[13] S. Muthukrishnan and C. Sahinalp. Approximate nearest neighbors and sequence comparisons with block operations. In *Proc. ACM Symp. on the Theory of Computing*, pages 416–424, 2000.

[14] E. Myers. A sublinear algorithm for approximate keyword searching. *Algorithmica*, 12(4/5):345–374, 1994. Earlier version in Tech. report TR-90-25, Dept. of CS, Univ. of Arizona, 1990.

[15] G. Navarro. A guided tour to approximate string matching. *ACM Comp. Surv.*, 33(1):31–88, 2001.

[16] G. Navarro and R. Baeza-Yates. A practical q-gram index for text retrieval allowing errors. *CLEI Electronic Journal*, 1(2), 1998. `http://www.clei.cl`. Earlier version in *Proc. CLEI'97*.

[17] G. Navarro and R. Baeza-Yates. A hybrid indexing method for approximate string matching. *J. of Discrete Algorithms*, 1(1):205–239, 2000. Hermes Science Publishing. Earlier version in *CPM'99*.

[18] G. Navarro, E. Sutinen, J. Tanninen, and J. Tarhio. Indexing text with approximate q-grams. In *Proc. 11th Ann. Symp. on Combinatorial Pattern Matching (CPM'2000)*, LNCS 1848, pages 350–363, 2000.

[19] F. Shi. Fast approximate string matching with q-blocks sequences. In *Proc. 3rd South American Workshop on String Processing (WSP'96)*, pages 257–271. Carleton University Press, 1996.

[20] E. Sutinen and J. Tarhio. On using q-gram locations in approximate string matching. In *Proc. 3rd European Symp. on Algorithms (ESA'95)*, LNCS 979, pages 327–340, 1995.

[21] E. Sutinen and J. Tarhio. Filtration with q-samples in approximate string matching. In *Proc. 7th Ann. Symp. on Combinatorial Pattern Matching (CPM'96)*, LNCS 1075, pages 50–61, 1996.

[22] T. Takaoka. Approximate pattern matching with samples. In *Proc. 5th Int'l. Symp. on Algorithms and Computation (ISAAC'94)*, LNCS 834, pages 234–242, 1994.

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

[24] E. Ukkonen. Approximate string matching over suffix trees. In *Proc. 4th Ann. Symp. on Combinatorial Pattern Matching (CPM'93)*, LNCS 684, pages 228–242, 1993.

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