Indexing Text with Approximate q-grams

Gonzalo Navarro [a,1], Erkki Sutinen [b] and Jorma Tarhio [c]

[a] *Department of Computer Science, University of Chile.* gnavarro@dcc.uchile.cl

[b] *Department of Computer Science, University of Joensuu.* sutinen@cs.joensuu.fi

[c] *Department of Computer Science and Engineering, Helsinki University of Technology.* tarhio@cs.hut.fi

Abstract

We present a new index for approximate string matching. The index collects text q-samples, that is, disjoint text substrings of length q, at fixed intervals and stores their positions. At search time, part of the text is filtered out by noticing that any occurrence of the pattern must be reflected in the presence of some text q-samples that match approximately inside the pattern. Hence the index points out the text areas that could contain occurrences and must be verified. The index parameters permit load balancing between filtering and verification work, and provide a compromise between the space requirement of the index and the error level for which the filtration is still efficient. We show experimentally that the index is competitive against others that take more space, being in fact the fastest choice for intermediate error levels, an area where no current index is useful.

Key words: Approximate string matching, text databases, q-gram indices.

1 Introduction and Related Work

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 is: Given a long text $T_{1 \ldots n}$ of length n, and a (comparatively short) pattern $P_{1 \ldots m}$ of length m, both sequences over an alphabet Σ of size σ, retrieve all the text substrings (or "occurrences") whose *edit distance* to the pattern is at most k. The *edit*

[1] Partially supported by Fondecyt Grant 1-020831.

distance between two strings A and B, $ed(A, B)$, is defined as the minimum number of character insertions, deletions and substitutions needed to convert A into B or vice versa. We define the "error level" as $\alpha = k/m$. Note that the problem is meaningful for $0 \leq \alpha < 1$, as otherwise the pattern matches everywhere.

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 [Sel80]. It is based on filling a matrix $C_{0\ldots m,0\ldots n}$, where $C_{i,j}$ is the minimum edit distance between $P_{1\ldots i}$ and a suffix of $T_{1\ldots j}$. Therefore all the text positions j such that $C_{m,j} \leq k$ are the endpoints of occurrences of P in T with at most k errors. The matrix is initialized at the borders with $C_{i,0} = i$ and $C_{0,j} = 0$, while its internal cells are filled using

$$C_{i,j} \quad = \quad \text{if} \;\; P_i = T_j \;\; \text{then} \;\; C_{i-1,j-1} \;\; \text{else} \;\; 1 + \min(C_{i-1,j}, C_{i-1,j-1}, C_{i,j-1})$$

which extends the previous alignment when the new characters match, and otherwise selects the best choice among the three alternatives of insertion, deletion and substitution. Fig. 1 shows an example. In an on-line searching only the previous column $C_{*,j-1}$ is needed to compute the new one $C_{*,j}$, so the space requirement is only $O(m)$.

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

Fig. 1. The dynamic programming matrix to search the pattern `"survey"` inside the text `"surgery"`. Bold entries indicate matching text positions when $k = 2$.

A number of algorithms improved later this result [Nav01]. The average lower bound of the on-line problem (proved and reached in [CM94]) is $\Omega(n(k + \log_\sigma m)/m)$, which is of course $\Omega(n)$ for constant m.

If the text is large even the fastest on-line algorithms are not practical, and preprocessing the text becomes necessary. However, just a decade ago, indexing text for approximate string matching was considered one of the main open problems in this area [WM92,BY92]. Despite some progress in recent years, the indexing schemes for this problem are still rather immature.

There are two types of indexing mechanisms for approximate string matching, which we call "word-retrieving" and "sequence-retrieving". Word retrieving indices [MW94,BYN00,ANZ97] are oriented to natural language text and information retrieval. They can retrieve every *word* whose edit distance to the pattern *word* is at most k. Hence, they are not able to recover from an error involving a separator, such as recovering the word `"flowers"` from the misspelled text `"flo wers"` or `"manyflowers"`, if we allow one error. These indices are more mature, but their restriction can be unacceptable in some applications, especially where there are no words (like in biological data such as DNA and proteins, and in multimedia data such as MIDI files), or where the concept of word is difficult to define (as in oriental languages like Chinese and Korean, and in agglutinating languages such as Finnish and German).

Our focus in this paper is sequence retrieving indices, which put no restrictions on the text, patterns or occurrences. Among these, we find three types of approaches.

Neighborhood Generation. This approach considers that the set of strings matching a pattern with k errors (called $U_k(P)$, the pattern "k-neighborhood") is finite, and therefore it can be enumerated and each string in $U_k(P)$ can be searched for using a data structure designed for *exact* searching. The data structures used have been the suffix tree [Knu73,AG85] and DAWG [Cro86,BBH$^+$85] of the text. These allow a recursive backtracking procedure for finding all the relevant text substrings (or suffix tree / DAWG nodes), instead of a brute-force enumeration and searching of all the strings in $U_k(P)$. The approaches [Gon92,JU91,Ukk93,Cob95] differ basically in the traversal procedure used on the data structure.

Those indices take $O(n)$ space and construction time. However, the constant is large. Even the most economical suffix tree implementations take 9 to 12 times the text size [GKS99]. The simpler search approaches [Gon92] can run over a suffix array [MM93,GBYS92], which still takes 4 times the text size. These high space requirements, together with the poor locality of reference of the search process, restrict the applicability of this approach to cases where the text and the large index fit in main memory.

With respect to search times, they are asymptotically independent on n, but exponential in k. The reason is that $|U_k(P)| = O((m\sigma)^k)$ holds [Ukk93]. Therefore, neighborhood generation is a promising alternative only for short patterns.

Reduction to Exact Searching. These indices are based on adapting online filters. Filters are fast algorithms that discard large parts of the text by checking for a necessary condition for matching (simpler than the exact condition). Most such filters are based on finding substrings of the pattern without errors, and checking for potential occurrences around those matches. The index is used to quickly find those pattern substrings without errors.

The main principle driving these indices is that, if two strings match with k errors and $k + s$ non-overlapping samples are extracted from one of them, then at least s of these must appear unaltered in the other. Some indices [Shi96,NBY98] use this principle by splitting the pattern into $k+s$ nonoverlapping pieces and searching these in the text, checking the text surrounding the areas where s pattern pieces appear at appropriate relative positions. These indices need to be able to find any text substring that matches a pattern piece, and are based on suffix trees/arrays or on indexing all the text q-grams (that is, substrings of length q). Related indices [JU91,HS94] are based on the intersections of two sets of q-grams: those in the pattern and those in its potential occurrence.

These indices can also be built in linear time and need $O(n)$ space. Depending on q they achieve different space-time tradeoffs. In general, filtration indices are much smaller than suffix trees (1 to 4 times the text size), although they only work well for low error levels α: Their search time is sublinear provided $\alpha = O(1/\log_\sigma n)$.

In another approach [ST96], the index stores the locations of only some text q-grams. These are collected at fixed intervals h, that is, one position out of h. These q-grams are called "q-samples". Then s is computed so that there are at least $k + s$ q-samples inside any occurrence. Thus, those text areas are checked where s pattern q-grams appear at appropriate relative positions. Using samples the index can take even less space than the text, although the acceptable error level is reduced even more.

Intermediate Partitioning. Somewhat between the previous approaches are [Mye94,NBY00], because they do not reduce the problem to exact but to approximate search of pattern pieces, and use a neighborhood generation approach to search for the pieces. The general principle is that if two strings match with at most k errors and j disjoint substrings are taken from one of them, then at least one of these appears in the other with $\lfloor k/j \rfloor$ errors. Hence, these indices split the pattern into j pieces, each piece is searched for in the index allowing $\lfloor k/j \rfloor$ errors, and the approximate matches of the pieces are extended to complete pattern occurrences. The existing indices differ in how j is selected (be it by indexing-time constraints [Mye94] or by optimization goals [NBY00]), and in the use of different data structures used to search for the pieces with a neighborhood generation approach. They achieve search time complexities of $O(n^\lambda)$, where $\lambda < 1$ for low enough error levels ($\alpha < 1 - e/\sqrt{\sigma}$, a limit probably impossible to surpass [BYN99]).

The idea of intermediate partitioning has given excellent results [NBY00] and it was shown to be an optimizing point between the extremes of neighborhood generation, that worsens as longer pieces are searched for, and reduction to exact searching, that worsens as shorter pieces are searched for. Moreover, intermediate partitioning permits handling higher error levels than those accepted by other filtering schemes. However, it has only been exploited in one direction: taking the pieces from the pattern. The other choice is to take text q-samples ensuring that at least j of them lie inside any occurrence of the pattern, and search for the pattern q-grams allowing $\lfloor k/j \rfloor$ errors in the index of text q-samples. This idea has been indeed proposed in [ST96] as an on-line filter, but it has never evolved into an indexing approach.

This is our main purpose. We first improve the filtering condition of [ST96] and then show how an index can be designed based upon this principle. We finally implement the index and experiment on it, both to obtain tuning recommendations and to compare it against others. In particular, our index turns out to be an excellent choice to search for relatively high error levels on DNA text, which makes it appealing for computational biology applications. We remark that the existing indices, even those based on intermediate partitioning, can handle only low error levels, so our index fills an important gap. It does so also conceptually: Table 1 shows how our contribution fills a hole in the possible alternatives attempted so far.

	Exact piece search	Approximate piece search
Pattern pieces	Split pattern into $k+s$ pieces [Shi96,NBY98]	Split pattern into j pieces [Mye94,NBY00]
Text pieces	Exact q-samples from the text [ST96]	Approximate q-samples from the text **(this work)**

Table 1
Different filtering approaches for approximate string matching.

The key of the success in handling higher error levels is that they are handled in a novel way: By adjusting parameters, we can do part of the dynamic programming already in the filtration phase, thus restricting the text area to be verified. In certain cases, this gives a better overall performance compared to the case where a weaker filtration mechanism results in a larger text area to be checked by dynamic programming. This is also the essential idea in the intermediate partitioning schemes [Mye94,NBY00], where the extremes of neighborhood generation and partitioning into exact searching do all the work in one part of the process.

Our index, however, has the advantage of taking little space. By selecting the interval h between the q-samples, the user can decide which of the two goals is more relevant: saving space with a larger h or better performance for higher error levels, using a smaller h.

An conference version of this paper appeared in [NSTT00].

2 The Filtration Condition

A filtration condition can be based on locating approximate matches of pattern q-grams in the text. In principle, this leads to a filtration tolerating higher error level as compared to the methods applying exact q-grams: An error breaking pattern q-gram u yields one error on it. Thus, the modified q-gram u' in an approximate match is no more an exact q-gram of the pattern, but an approximate q-gram of it. Hence, while u' cannot be used in a filtration scheme based on exact q-grams, it gives essential information for a filtration scheme based on approximate q-grams.

This is the idea we pursue in this section. We start with a lemma that is used to obtain a necessary condition for an approximate match.

Lemma 1 *Let A and B be two strings such that $ed(A,B) \leq k$. Let $A = A_1x_1A_2x_2 \ldots x_{j-1}A_j$, for strings A_i and x_i and for any $j \geq 1$. Then, at least one string A_i appears in B with at most $\lfloor k/j \rfloor$ errors.*

Proof: Since at most k edit operations (errors) are performed on A to convert it into B, and each such operation can affect only one piece, then at least one of the A_i's gets no more than $\lfloor k/j \rfloor$ of them. Otherwise, if each A_i appears inside B with at least $\lfloor k/j \rfloor + 1 > k/j$ errors, then the whole A needs strictly more than $j \cdot k/j = k$ errors to be converted into B. □

This shows that an approximate match for a pattern implies also the approximate match of some pattern pieces. It is worthwhile to note that it is possible that $j \cdot \lfloor k/j \rfloor < k$, so we are not only "distributing" the errors across pieces but also "removing" some of them. Fig. 2 illustrates.

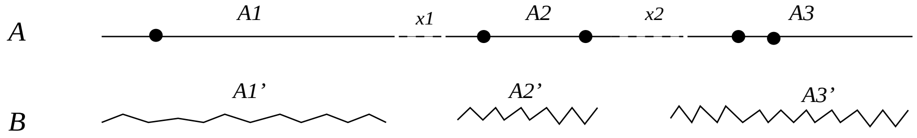


Fig. 2. Illustration of Lemma 1, where $k = 5$ and $j = 3$. At least one of the A_i's has at most $\lfloor 5/3 \rfloor = 1$ error (in this case A_1).

Lemma 1 is used by considering that the string B is the pattern and the string A is its occurrence in the text. Hence, we need to extract j pieces from each potential pattern occurrence in the text.

Given some q and $h \geq q$, we extract one text q-gram (called a "q-sample") each h text characters. Let us call d_r the q-samples, $d_1, d_2, \ldots, d_{\lfloor \frac{n}{h} \rfloor}$, where $d_r = T_{h(r-1)+1 \ldots h(r-1)+q}$.

We need to guarantee that there are at least j text samples inside any occurrence of P. An occurrence of P has minimum length $m-k$, and in the worst case it may start at the second position of a q-sample (thus not containing it), and hence it must extend by $h-1+(j-1)h+q$ characters in order to contain the j q-samples that follow to the right. The resulting condition is $jh+q-1 \leq m-k$. Note that h and q have to be known at indexing time, when m and k are unknown, so at search time we have to adjust j according to the rest. The condition on j is

$$1 \;\leq\; j \;\leq\; \left\lfloor \frac{m-k-q+1}{h} \right\rfloor \tag{1}$$

which shows that an index built using some q and h values is only useful for searches where $m-k \geq h+q-1$ (as otherwise there is no way to guarantee that occurrences contain at least one q-sample).

Fig. 3 illustrates the idea, pointing out another fact not discussed until now. If the pattern P matches a text area containing a *test sequence* of q-samples $D_r = d_{r-j+1} \ldots d_r$, then d_{r-j+i} must match inside a specific substring Q_i of P. These *pattern blocks* are overlapping substrings of P, namely $Q_i = P_{(i-1)h+1 \ldots ih+q-1+k}$. To see this, let us focus first in the case $k=0$. Consider an alignment of P against $T_{i+1 \ldots i+m}$. The first q-sample can be from $T_{i+1 \ldots i+q}$ to $T_{i+h \ldots i+h+q-1}$, so it can be aligned with $P_{1 \ldots q}$ to $P_{h \ldots h+q-1}$. Hence it must appear in $P_{1 \ldots h+q-1}$. In general, the i-th q-sample must appear in $P_{(i-1)h+1 \ldots ih+q-1}$. If we now consider the possibility of performing up to k insertions in T before the end of the i-th q-gram, we must extend the right end of Q_i by k positions, obtaining the result.

A *cumulative best match distance* is computed for each D_r, as the sum of the best distances of the involved consecutive text samples d_{r-j+i} inside the Q_i's. More formally, we compute for D_r

$$\sum_{1 \leq i \leq j} bed(d_{r-j+i}, Q_i),$$

where

$$bed(u, Q) = \min_{1 \leq i \leq i' \leq |Q|} ed(u, Q_{i \ldots i'}).$$

That is, $bed(u,Q)$ gives the best edit distance between u and a substring of Q. It is easy to see that the cumulative best match distance is a lower bound to the edit distance between P and any text occurrence containing the test sequence D_r. Thus, the text area corresponding to D_r is examined only if its cumulative best match distance does not exceed k.

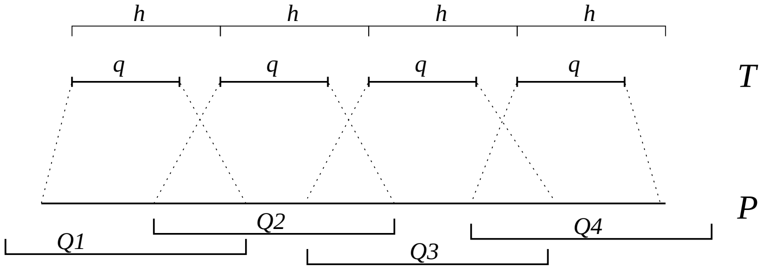


Fig. 3. Searching using q-samples, showing how the four relevant text samples at each position are aligned with the corresponding pattern blocks.

The algorithm works as follows. Each *counter* M_r, corresponding to the sequence $D_r = d_{r-j+1} \ldots d_r$, indicates the cumulative best match distance for D_r. Actually, for reasons that will be fully clear later, M_r is an underestimation of the actual best match distace. The counters are initialized to $M_r = j(e+1)$, were $q > e \geq \lfloor k/j \rfloor$ will be specified in Section 3. That is, we start by assuming that each text q-sample yields $e + 1$ errors, enough to disallow a match. Later, we concentrate only on those q-samples that can be found in the pattern blocks with at most e errors, and the others will be assumed to yield only $e + 1$ errors. This permits us focusing on few q-samples, at the cost of assuming that all the others have less errors than what they really have, thus unnecessarily verifying some text areas. (If we use $e + 1 = q$ then we will not make unnecessary verifications, as any q-sample matches with q errors.) Note that our bounds for e imply a further restriction on j: $q > k/j$.

Now, for each pattern block Q_i, we obtain its "q-gram e-environment", defined as

$$U_e^q(Q_i) = \{u \in \Sigma^q, bed(u, Q_i) \leq e\}$$

which is the set of q-grams that appear inside Q_i with at most e errors. Now, each $d_r \in U_e^q(Q_i)$ represents a text q-sample that matches inside pattern block Q_i. Therefore, we update all the counters

$$M_{r+j-i} \quad \leftarrow \quad M_{r+j-i} - (e+1) + bed(d_r, Q_i)$$

Finally, all the text areas whose counter $M_r \leq k$ are checked with dynamic programming. The text area corresponding to D_r is $T_{h(r-j)+1 \ldots h(r-1)+q}$. By considering that the occurrence can start up to $h - 1$ characters behind the first q-sample and be of length up to $m + k$, we have that the text area to verify when $M_r \leq k$ is $T_{h(r-j-1)+2 \ldots h(r-j-1)+m+k+1}$.

Of course, it is not necessary to maintain all the counters M_r, since they can implicitly be assumed to be initialized at $j(e + 1)$ until a text q-sample participating in D_r is found in some $U_e^q(Q_i)$.

Fig. 4 gives a simplified pseudocode for the indexing and search processes.

```
Indexing ($T_{1...n}$)
1.  Choose $q$ and $h$, $q \leq h$
2.  Initialize empty trie of $q$-samples
3.  For $r \in 1 \ldots \lfloor n/h \rfloor$
4.    Insert $d_r = T_{h(r-1)+1...h(r-1)+q}$ into the trie

Searching ($P_{1...m},\ k$)
1.  Choose $j$, $1 \leq j \leq \lfloor \frac{m-k-q+1}{h} \rfloor$
2.  Choose $e$, $\lfloor k/j \rfloor \leq e < q$
3.  For $r \in 1 \ldots \lfloor n/h \rfloor$
4.    $M_r \leftarrow j(e+1)$
5.  For $i \in 1 \ldots j$
6.    $Q_i \leftarrow P_{(i-1)h+1...ih+q-1+k}$
7.    For each trie $q$-gram $d_r$ such that $bed(d_r, Q_i) \leq e$
8.      $M_{r+j-i} \leftarrow M_{r+j-i} - (e+1) + bed(d_r, Q_i)$
9.  For $r \in 1 \ldots \lfloor n/h \rfloor$
10.   If $M_r \leq k$
11.     Run dynamic programming over $T_{h(r-j-1)+2...h(r-j-1)+m+k+1}$
```

Fig. 4. Indexing and searching pseudocode, at a conceptual level. In lines 1 and 2 of the search process it may happen that no suitable j or e value exists, in which case the index is not suitable for that (q, h, m, k) combination.

3 Finding Approximate q-Grams

In this section we focus on the problem of finding all the text q-samples that appear *inside* a given pattern block Q_i, that is, find all the indexes r such that $d_r \in U_e^q(Q_i)$. The first observation is that it is not necessary to generate the whole $U_e^q(Q_i)$, since we are interested only in the q-samples that appear in the text (more specifically, in their positions). So we actually generate

$$I_e^q(Q_i) = \{r \in 1 \ldots \lfloor n/h \rfloor, bed(d_r, Q_i) \leq e\}$$

The idea is to store all the different text q-samples in a trie data structure, where the leaves store the corresponding r values. A backtracking approach is used to find all the leaves of the trie that are relevant for a given pattern block Q_i, that is, those that match inside Q_i with at most e errors.

From now on we use $Q = Q_i$ and free variable i for other purposes. If considering a specific text q-sample $S = s_1 \ldots s_q$ (corresponding to some d_r), the problem is solved by the use of the dynamic programming algorithm explained in the Introduction, where the text is the pattern block Q and the pattern is the text q-sample S. That is, we fill a matrix $C_{0\ldots q,0\ldots|Q|}$ such that $C_{i,\ell}$ is the smallest edit distance between $S_{1\ldots i}$ and a suffix of $Q_{1\ldots \ell}$. When this matrix is filled, we have that the text q-sample S is relevant if and only if $C_{q,\ell} \leq e$ for some ℓ (in other words, S matches somewhere inside Q with at most e errors). In a trie traversal of the q-samples, the characters of S are obtained one by one as we descend by each branch, so this matrix will be filled row-wise rather than column-wise, which is the typical choice in on-line searching.

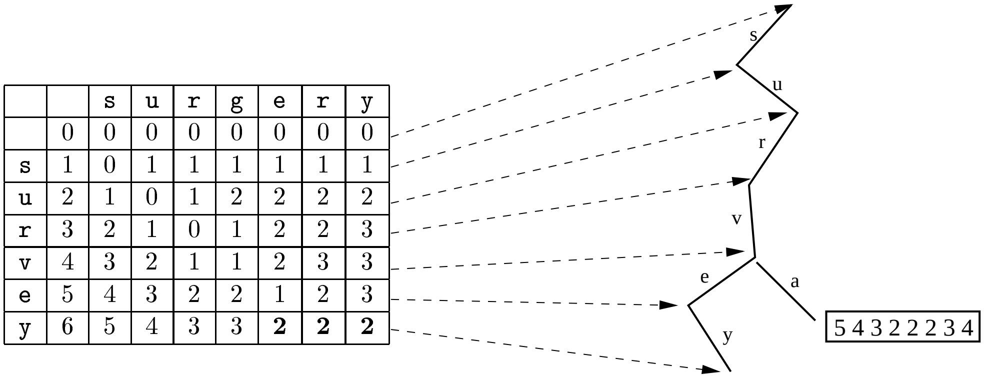
The algorithm works as follows. We perform an exhaustive search on the trie, starting at the root and entering into every children of each node. At each moment, if we are in a trie node representing a prefix S' of some text q-samples, we keep $C_{|S'|,\ell}$ for all ℓ, that is, the current row of the dynamic programming matrix. Upon entering into the children of the current node following an edge labeled with character c, a new row of C is computed from the current one using c as the next pattern character. When we reach the leaf nodes of the trie (at depth q) we check in the last row of C whether there is a cell with value at most e, in which case the corresponding text q-sample is processed. This means that we have all the text positions r of the q-sample, and we update all the corresponding counters as explained in Section 2. Note that since we only store the rows of the ancestors of the current node at each time, the total space requirement for the backtrack is just $O(|Q|q) = O(mq)$.

As we presented it, it seems that we traverse all the nodes of the trie. This is already better than on-line searching because all the common prefixes of different q-samples are processed only once. However, some further pruning can be done. As all the values from a row to the next are nondecreasing, we know that if all the values of a row are larger than e then this will also hold in descendant nodes. Therefore, at that point we can abandon that branch of the trie without actually considering its subtree.

Fig. 5 shows an example, using $Q =$ `"surgery"` and $S =$ `"survey"`. If $e = 1$ then the alternative path shown can be abandoned immediately since all its entries are larger than 2.

4 The Parameters of the Algorithm

The value of e has been left unspecified in the previous development. This is because there is a tradeoff involved. If we use a small e value, then the search for the e-environments will be faster, but as we have to assume that the text q-samples not found have only $e+1$ errors (which may underestimate the real number of errors they have), 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



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

Fig. 5. The dynamic programming algorithm run over the trie of text q-samples. We show just one path and one additional child.

verifications, in exchange for a higher cost to find the e-environments.

As the cost of this search grows exponentially with e, the minimal $e = \lfloor k/j \rfloor$ can be a good choice. With the minimal e the sequences D_{r-j+i} are assumed to have $j(\lfloor k/j \rfloor + 1)$ errors, which can get as low as $k+1$. In that particular case we can avoid the use of counters, since every text q-gram d_{r-j+i} found inside Q_i will trigger a verification in D_r.

It is interesting to consider the interplay between the different remaining parameters h, q and j. Eq. (1) relates these parameters, introducing also m and k in the condition. For a given query, j has a maximum acceptable value. As j grows, longer test sequences with less errors per sample are used, so the cost to find the relevant q-samples decreases but the amount of text verification increases.

So j and e permit finding the best compromise between both parts of the search. On the other hand, q and h determine the space usage of the index, which is in the worst case $O(\sigma^q + n/h)$ integers (one header per different q-sample and one integer per sampled text position). Having a smaller index restricts more the allowed j values and, indirectly, the e values.

Table 2 shows all the parameters of the index, together with their limits and usage.

5 Implementation-Independent Empirical Evaluation

In this section we empirically study the behavior of our index, focusing on the aspects that are independent on the actual implementation. Texts and patterns have been generated uniformly over Σ and independently of each other. We have used $n = 100,000$ and $m = 40$, and considered alphabet sizes

Name	Meaning	Limits	When it grows...
q	Length of text samples (chosen at indexing time).	$q \leq h$	Improves filtration but increases index space, $O(\sigma^q)$. It also puts limits on $m-k$, h, j, and e.
h	Distance between samples (chosen at indexing time)	$h \leq m-k-q+1$	Worsens filtration but decreases index space, $O(n/h)$. It also puts limits on q, $m-k$, and j.
j	Number of samples in occurrences (chosen at query time).	$\frac{k}{q} < j \leq \frac{m-k-q+1}{h}$	Permits reducing backtracking but increases verification. It also puts limits on e.
e	Errors presumed for samples (chosen at query time).	$k/j \leq e < q$	Increases backtracking but reduces verification.

Table 2
The parameters of our index.

$\sigma = 4$ and $\sigma = 20$ to simulate the cases of DNA and proteins, respectively. In this section we speak about "processed columns", referring to text positions that have to be processed by dynamic programming due to verifications.

Table 3 shows how the error level increases the number of processed columns, for alphabets of size 4 and 20. In each case we have used the maximum possible j according to Eq. (1) and the minimum $e = \lfloor k/j \rfloor$. The behavior in other alphabets is similar, but a bigger alphabet implies a higher tolerated error level. Note that some fluctuations in the number of processed columns are due to changes in the value of e.

For a given k value, it is possible to reduce the number of processed columns by changing h, q or j. Compare the results in Table 4 to those in Table 3.

Table 5 shows how our scheme allows to do part of the dynamic programming already in the filtration phase, by traversing the trie structure and evaluating minimum edit distances between q-samples and substrings of pattern blocks. This is based on increasing the value of e. Although the results seem promising at a first sight, one has to remember that a small portion of processed columns does not necessarily imply a shorter processing time. In fact, the optimal setting for e depends on several factors, such as the length of the text and the implementation of the trie.

The distance h between the q-samples is crucial to the space requirement of the index. Table 6 shows that a lower interval h, and thus, a larger index, yields a more efficient filtration, as indicated, for example, in the number of processed columns.

Since the index of the presented approach only stores non-overlapping q-

			$\sigma = 4$	$\sigma = 20$
k	j	e	Columns	Columns
$0 \ldots 3$	5	0	0.0	0.0
4	5	0	7.5	0.0
5	5	1	0.0	0.0
6	4	1	33.9	0.0
7	4	1	93.7	0.1
8	4	2	97.0	0.0
9	4	2	100.0	0.0
10	4	2	100.0	0.2
11	4	2	100.0	9.0
12	3	4	100.0	99.9
13	3	4	100.0	100.0

Table 3
Percentage of processed columns for $q = h = 6$.

k	h	q	j	e	Columns
6	7	7	4	1	6.0
7	8	8	3	2	44.2
8	8	8	3	2	95.6

Table 4
Percentage of processed columns for $\sigma = 4$, for different values of h, q and j.

e	Columns	Nodes
1	33.3	8,061
2	11.6	19,304
3	9.6	21,500
4	7.1	21,544
5	4.9	21,544
6	2.1	21,544

Table 5
Percentage of processed columns and number of traversed nodes of the q-sample trie for $\sigma = 4$, $k = 6$, $q = h = 6$, and $j = 4$, for different values of e.

samples, its space requirement is small, and can be kept below the size of the text [ST96]. This should be kept in mind when the performance is compared to other related approaches. Table 7 shows that the new approach works

h	j	Columns	Space
7	11	100.0	57%
6	8	99.8	66%
5	6	90.7	80%
4	5	14.2	100%
3	4	0.1	133%

Table 6
Percentage of processed columns for decreasing h, for $\sigma = 4$, $k = 5$, $q = 3$, and $e = 3$. Note that the parameter j has to be adjusted according to h. The space is indicated in terms of percentage of text size needed by the index, for large n.

for a small error level almost as efficiently as its competitor [NBY00] which, however, consumes more space (4 times the text space).

k	Ours	Interm.
4	0.0	1.0
5	0.3	1.0
6	5.3	1.1
7	30.2	1.2
8	81.1	22.9
9	99.5	23.6

Table 7
Percentage of processed columns for relatively low error levels. Our approach collects *non-overlapping* q-samples, and the intermediate partitioning approach [NBY00], denoted by "Interm", stores *all* the text pieces which need to be searched for. The parameters are as follows: $\sigma = 4$, $q = h = 6$, $j = 4$, $e = 6$.

Let us conclude by briefly discussing how the space consumption of our index depends on the sampling interval h. The standard implementation of a q-gram index (as well as a suffix array) stores all the locations of all the q-grams of the text. Since the number of q-grams in a text of length n is $n - q + 1$ and storing a position takes $\log n$ bits (without compression), the overall space consumption is $n \log n$ (q is small compared to n). Let us define a *space saving factor* v_r as the space requirement ratio between our method and the standard approach, that is,

$$v_r = \frac{\frac{n}{h} \log \frac{n}{h}}{n \log n} \approx \frac{1}{h} \text{ (for large } n\text{).}$$

Table 8 shows how the space saving factor improves with increasing h.

h	v_r
1	1.000
2	0.470
3	0.302
4	0.220
5	0.172
6	0.141
7	0.119
8	0.102
9	0.090
10	0.080

Table 8
Space saving factor v_r for $n = 100,000$.

6 Implementation and Experimental Evaluation

In this section we describe our actual index implementation and evaluate its performance. Our implementation is rather simple and in-memory, omitting several possible improvements. Our trie of q-grams is implemented as a tree with pointers. The children of a node are allocated in a single block of memory, that grows using a doubling scheme. The text positions of all the q-samples are stored as plain integers without compression, in a single chunk of memory (the construction makes two passes over the text to precompute the sizes to be allocated to each q-sample inside the large chunk). At search time, a hash table is used to store the counters M_r that are pointed out during the backtracking in the trie. The plain dynamic programming algorithm is used both to backtrack in the trie and for verifying text areas.

We have used 30 MB of DNA text from the human genome obtained from GenBank, and 30 MB of English text from Wall Street Journal articles of TREC-3 collection. As our index turned out to be competitive on DNA rather than on English, we will focus more on the former. The search patterns were extracted at random text positions. Our machine is an Intel Pentium IV of 2 GHz and 512 MB of RAM running Linux. The code is written in C and compiled using full optimization. The experiments were repeated enough times to ensure a percentual error below 2%, with 95% confidence[2]. We measure CPU times.

[2] That is, N times, such that $2\hat{\sigma}/\sqrt{N} \leq 0.02\hat{\mu}$, where $\hat{\sigma}$ and $\hat{\mu}$ are, respectively, the standard deviation and mean estimators.

Fig. 6 shows the space required by our index and the time necessary to build it, for $q = 7$ and $h = 7$, 9 and 11 on DNA. As we can see, the index takes about half the space of the text and is built in at most 1 sec/MB. The space looks a bit sublinear because the overhead of the trie of different q-samples is more significant for smaller texts.

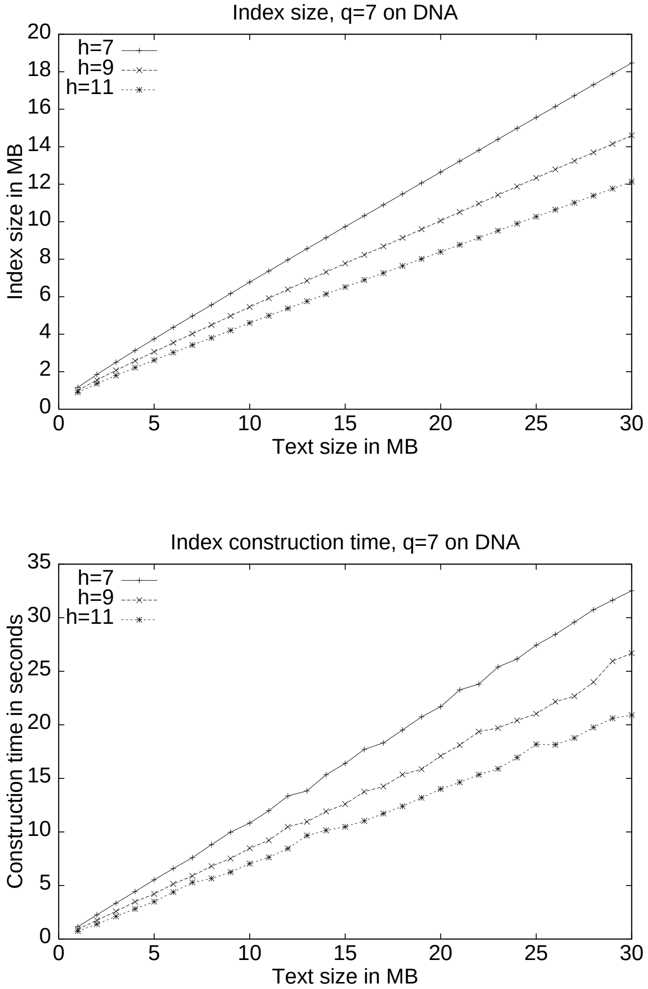


Fig. 6. On top, space used by our index. On the bottom, construction time.

We consider now the optimal choice of parameters e and j. Since there are two parameters to tune at the same time, it is interesting to obtain a simple recommendation that works well in most cases, so as to simplify the usage of the index. We performed extensive experiments on English and DNA texts, for $5 \leq m \leq 60$ and $0 \leq k \leq 0.4m$, and several combinations of q and h values. Fortunately, it turns out that the rule of thumb that almost always gives the best choice (and always a reasonable one) is rather simple and independent of many factors: Use $e = 1$ and j as large as possible according to Eq. (1). This works as long as j satisfies $\lfloor k/j \rfloor \leq e = 1$, otherwise we have no choice but using a larger e.

Using the above rule for j and e, we compare our implementation (*Approx-*

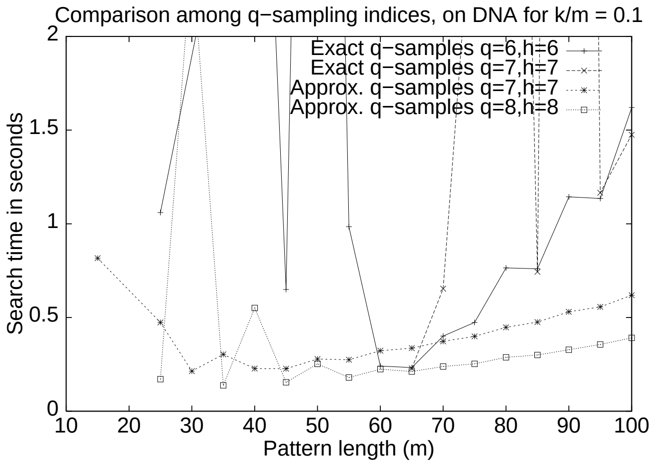
imate q-samples) against its competing approaches. *Exact q-samples* is our implementation of [ST96], reusing most of our code. *Exact partitioning* is our implementation of [Shi96,NBY98], over a trie of all the text q-grams. *Intermediate partitioning* is our implementation of [NBY00], based on backtracking over a suffix array. We have not used the optimized code of the authors, but reimplemented the indices under our standards (for example, using dynamic programming for backtracking and verification). The reason is that those optimizations can be made over all the indices and we want to compare them under similar conditions. Finally, we show, as a reference point, the time needed by a sequential scan of the text using plain dynamic programming (*Sequential*). All the indices used were tuned to their optimal choice of parameters.

Fig. 7 shows how our index compares against the only other index able of using that little space: exact q-samples [ST96]. Although that index works better than ours on English text, on DNA it rarely performs well, and our index is always better. This shows that our index is more resistant than its predecessor to higher error levels, as expected (the same α value is harder to handle when the alphabet is smaller).

Only for this comparison we have considered patterns of length up to 100. The large peaks in the figures are due to the integer nature of the problem. For example, with exact q-samples, $m = 40$, $q = 6$, $h = 6$, we allow $k = 4$ and hence it is enough that an area contains $s = \lfloor (m-k-q+1)/h \rfloor - k = 1$ pattern sample to verify it [ST96]. With $m = 45$, however, still $k = 4$ and we need at least $s = 2$ pattern samples in the area to verify it, so the performance improves greatly. With $m = 50$, k raises to 5 and then $s = 1$ again. Note also that some (m, k, q, h) combinations are not possible because they require verification when $s = 0$ q-samples match, for example $m = 50$, $k = 5$, $q = 7$, $h = 7$. The important point is to see how the lowest lines are due to exact q-samples on English text and to our index on DNA.

Finally, Figs. 8 and 9 compare our index against other indices that take 4 times the text size, on DNA. We consider $k/m = 0.10 \ldots 0.35$, which covers basically all the cases of interest in DNA searching. As it can be seen, our index is not competitive for low error levels or for very short patterns, but it becomes the only existing choice for $m \geq 30$ and $k/m \geq 0.3$.

It can be argued that we are comparing our index against plain dynamic programming, while faster sequential algorithms could beat it in the area where our algorithm is the only choice. However, this area corresponds to a high error level, where there are very few choices for sequential algorithms [Nav01] (for example, filtering algorithms do not work). In fact, the only possible speedup is to replace the dynamic programming by a bit-parallel simulation [Mye99]. However, as we explain in the next section, the same speedup could be obtained in our index by using that technique, both for traversing the trie and



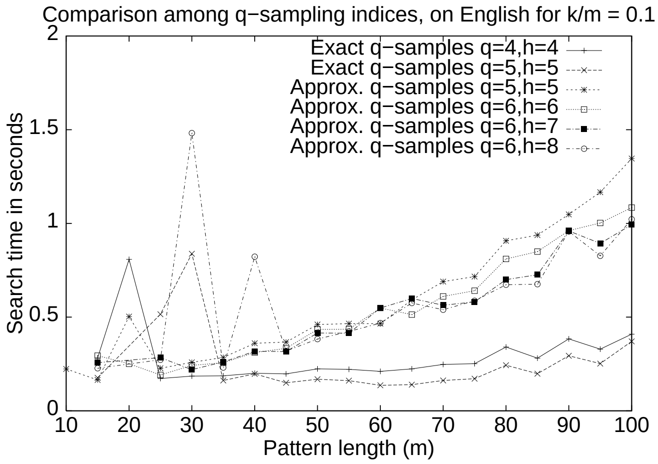
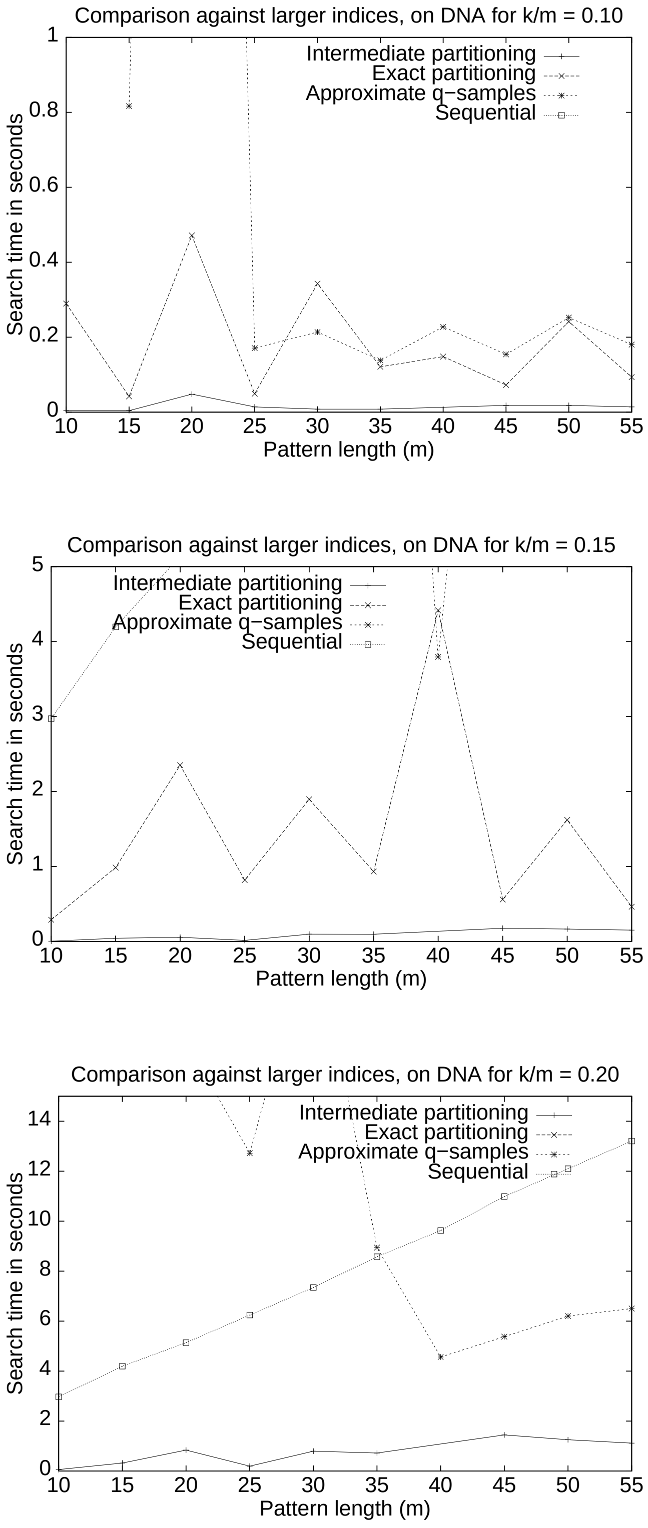


Fig. 7. Comparison against exact q-samples index, for $k/m = 0.1$. We show DNA on top and English on the bottom. Times for sequential scanning are above 3 seconds, out of the plot range.

for text verification. Hence all the results would downscale similarly and their relative speeds would not change.

7 Conclusions

We have introduced a static pattern matching scheme which is based on locating approximate matches of the pattern substrings among the q-samples of the text. The mechanism breaks the fixed division of pattern matching into two phases, filtration and checking, where dynamic programming belongs only to the last phase. In our approach, it is possible to share dynamic programming between these phases by setting appropriate parameters. This is an important feature, since it makes it possible to tune the algorithm according to the particular problem instance. In some cases, saving space is a critical issue,


Comparison against larger indices, on DNA for k/m = 0.10

Comparison against larger indices, on DNA for k/m = 0.15

Comparison against larger indices, on DNA for k/m = 0.20

Fig. 8. Comparison against other indices, for small k/m values. Some search times fall out of the plot range.

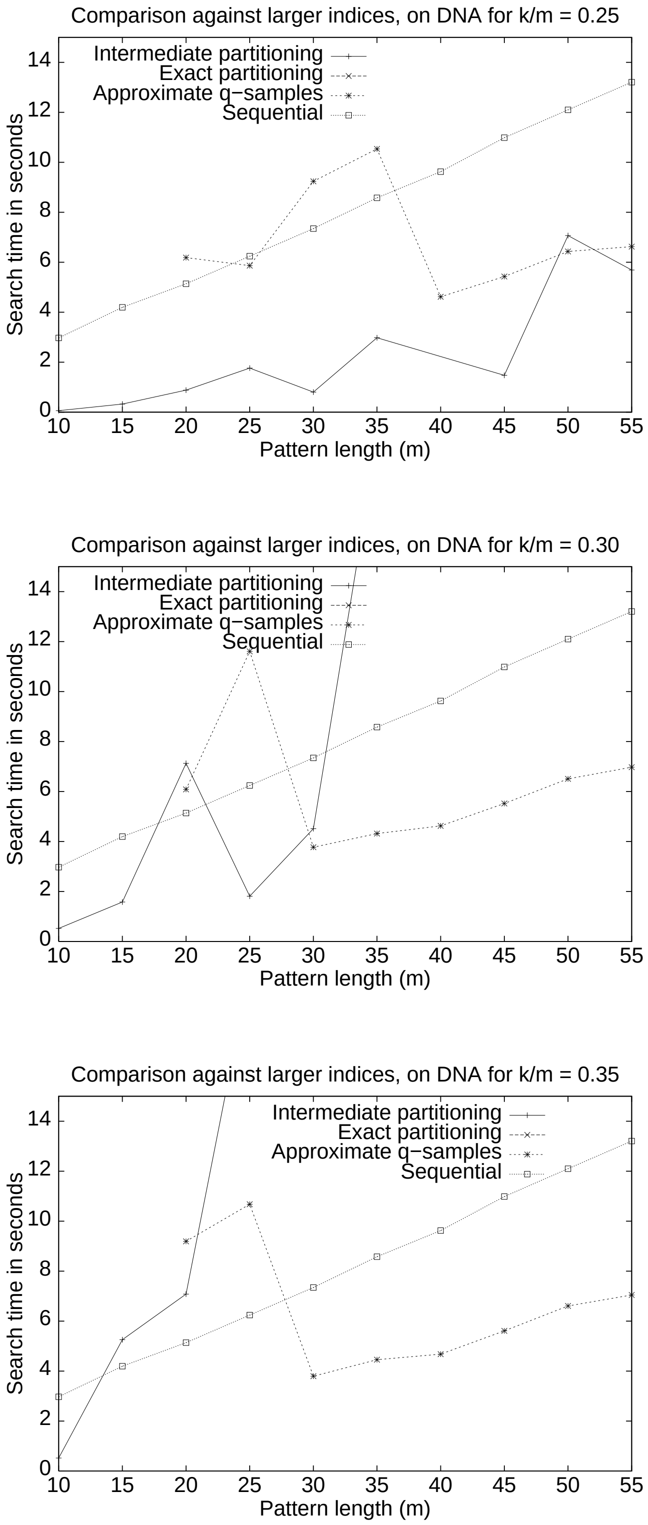


Fig. 9. Comparison against other indices, for medium k/m values. Some search times fall out of the plot range.

whereas a high error level requires a denser index.

Our experimental results demonstrated that we not only filled a conceptual gap with respect to the combinations of approaches attempted so far, but also an efficiency gap, providing an index that works well on high error levels. The index worked particularly well on DNA, which makes it appealing for computational biology applications.

Several optimizations can be performed over our implementation. The most obvious is to use bit-parallel algorithms to process the dynamic programming matrices used during backtracking and at verification time. The most promising algorithm for this task is that of Myers [Mye99]. This algorithm is conceived to run column by column, while our backtracking phase works row by row. However, this change is rather simple because the update formula is symmetric and the matrix can be transposed using the same formula. The only important change is that now the current "column" is initialized with zeros, and that the first cell "column" j has value j instead of zero. Both issues have been already addressed [HN02].

Acknowledgements

We thank Jani Tanninen (Univ. of Joensuu) and Javier Bustos (Univ. of Chile) for the experimental results. We also thank the comments of an anonymous referee, that improved the readability of the paper.

References

[AG85] A. Apostolico and Z. Galil. *Combinatorial Algorithms on Words.* Springer-Verlag, New York, 1985.

[ANZ97] M. Araújo, G. Navarro, and N. Ziviani. Large text searching allowing errors. In *Proc. WSP'97*, pages 2–20. Carleton University Press, 1997.

[BBH$^+$85] A. Blumer, J. Blumer, D. Haussler, A. Ehrenfeucht, M. Chen, and J. Seiferas. The samllest automaton recognizing the subwords of a text. *Theoretical Computer Science*, 40:31–55, 1985.

[BY92] R. Baeza-Yates. Text retrieval: Theory and practice. In *12th IFIP World Computer Congress*, volume I, pages 465–476. Elsevier Science, September 1992.

[BYN99] R. Baeza-Yates and G. Navarro. Faster approximate string matching. *Algorithmica*, 23(2):127–158, 1999.

[BYN00] R. Baeza-Yates and G. Navarro. Block-addressing indices for approximate text retrieval. *J. of the American Society for Information Science (JASIS)*, 51(1):69–82, January 2000.

[CM94] W. Chang and T. Marr. Approximate string matching and local similarity. In *Proc. CPM'94*, LNCS 807, pages 259–273, 1994.

[Cob95] A. Cobbs. Fast approximate matching using suffix trees. In *Proc. CPM'95*, pages 41–54, 1995. LNCS 937.

[Cro86] M. Crochemore. Transducers and repetitions. *Theoretical Computer Science*, 45:63–86, 1986.

[GBYS92] 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.

[GKS99] R. Giegerich, S. Kurtz, and J. Stoye. Efficient implementation of lazy suffix trees. In *Proc. WAE'99*, LNCS 1668, pages 30–42, 1999.

[Gon92] G. Gonnet. A tutorial introduction to Computational Biochemistry using Darwin. Technical report, Informatik E.T.H., Zuerich, Switzerland, 1992.

[HN02] H. Hyyrö and G. Navarro. Faster bit-parallel approximate string matching. In *Proc. 13th Combinatorial Pattern Matching (CPM'2002)*, LNCS 2373, pages 203–224, 2002.

[HS94] N. Holsti and E. Sutinen. Approximate string matching using q-gram places. In *Proc. 7th Finnish Symposium on Computer Science*, pages 23–32. University of Joensuu, 1994.

[JU91] P. Jokinen and E. Ukkonen. Two algorithms for approximate string matching in static texts. In *Proc. of MFCS'91*, volume 16, pages 240–248, 1991.

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

[MM93] U. Manber and E. Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, pages 935–948, 1993.

[MW94] U. Manber and S. Wu. GLIMPSE: A tool to search through entire file systems. In *Proc. USENIX Technical Conference*, pages 23–32, Winter 1994.

[Mye94] E. Myers. A sublinear algorithm for approximate keyword searching. *Algorithmica*, 12(4/5):345–374, Oct/Nov 1994.

[Mye99] G. Myers. A fast bit-vector algorithm for approximate string matching based on dynamic progamming. *Journal of the ACM*, 46(3):395–415, 1999.

[Nav01] G. Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31–88, 2001.

[NBY98] 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`.

[NBY00] G. Navarro and R. Baeza-Yates. A hybrid indexing method for approximate string matching. *Journal of Discrete Algorithms (JDA)*, 1(1):205–239, 2000.

[NSTT00] G. Navarro, E. Sutinen, J. Tanninen, and J. Tarhio. Indexing text with approximate q-grams. In *Proc. 11th Annual Symposium on Combinatorial Pattern Matching (CPM'2000)*, LNCS 1848, pages 350–363, 2000.

[Sel80] P. Sellers. The theory and computation of evolutionary distances: pattern recognition. *J. of Algorithms*, 1:359–373, 1980.

[Shi96] F. Shi. Fast approximate string matching with q-blocks sequences. In *Proc. WSP'96*, pages 257–271. Carleton University Press, 1996.

[ST96] E. Sutinen and J. Tarhio. Filtration with q-samples in approximate string matching. In *Proc. CPM'96*, LNCS 1075, pages 50–61, 1996.

[Ukk93] E. Ukkonen. Approximate string matching over suffix trees. In *Proc. CPM'93*, pages 228–242, 1993.

[WM92] S. Wu and U. Manber. Fast text searching allowing errors. *Comm. of the ACM*, 35(10):83–91, October 1992.