Indexing Text with Approximate q-grams

Gonzalo Navarro[1]*, Erkki Sutinen[2], Jani Tanninen[2], and Jorma Tarhio[2]

[1] Dept. of Computer Science, University of Chile. gnavarro@dcc.uchile.cl
[2] Dept. of Computer Science, University of Joensuu, Finland.
{sutinen,jtanni,tarhio}@cs.joensuu.fi

Abstract. We present a new index for approximate string matching. The index collects text q-samples, i.e. 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. We show experimentally that the parameterization mechanism of the related filtration scheme provides a compromise between the space requirement of the index and the error level for which the filtration is still efficient.

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 is: 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, retrieve all the text substrings (or "occurrences") whose *edit distance* to the pattern is at most k. The *edit distance* between two strings A and B, $ed(A, B)$, is defined as the minimum number of character insertions, deletions and replacements needed to convert A into B or vice versa. We define the "error level" as $\alpha = k/m$.

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 [23]. It is based in filling a matrix $C_{0..m,0..n}$, where $C_{i,j}$ is the minimum edit distance between $P_{1..i}$ and a suffix of $T_{1..j}$. 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

* Work developed during postdoctoral stay at the University of Helsinki, partially supported by the Academy of Finland and Fundación Andes. Also supported by Fondecyt grant 1-000929.

and replacement. Figure 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 [20]. The lower bound of the on-line problem (proved and reached in [7]) is $O(n(k + \log_\sigma m)/m)$, 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 few years ago, indexing text for approximate string matching was considered one of the main open problems in this area [27, 3]. Despite some progress in the last 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 indexes [18, 5, 2] are more 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", if we allow one error. These indexes are more mature, but their restriction can be unacceptable in some applications, especially where there are no words (as in DNA), where the concept of word is difficult to define (as in oriental languages) or in agglutinating languages (as Finnish).

Our focus in this paper is sequence retrieving indexes, which put no restrictions on the patterns and their 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 using a data structure designed for *exact* searching. The data structures used have been the suffix tree [16, 1] and DAWG [9, 6] of the text. These data structures 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 [12, 15, 26, 8] differ basically in the traversal procedure used on the data structure.

Those indexes take $O(n)$ space and construction time, but their construction is not optimized for secondary memory and is very inefficient in this case (see, however, [10]). Moreover, the structure is very inefficient in space requirements, since it takes 12 to 70 times the text size (see, e.g. [11]). The simpler search approaches [12] can run over a suffix array [17, 13], which takes 4 times the text size. With respect to search times, they are asymptotically independent on n, but exponential in m or k. The reason is that $|U_k(P)| = O(\min(3^m, (m\sigma)^k)$ [26]. Therefore, neighborhood generation is a promising alternative for short patterns only.

Reduction to Exact Searching. These indexes are based on adapting on-line filtering algorithms. Filters are fast algorithms that discard large parts of the text checking for a necessary condition (simpler than the matching 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 indexes 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 indexes [24, 21] use this principle by splitting the pattern in $k+s$ nonoverlapping pieces and searching these in the text, checking the text surrounding the areas where s pattern pieces appear at reasonable distances. These indexes need to be able to find any text substring that matches a pattern piece, and are based on suffix trees or indexing all the text q-grams (i.e. substrings of length q).

In another approach [25], the index stores the locations of all the text q-grams with a fixed interval h; these q-grams are called "q-samples". The distance h between samples 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 reasonable distances among each other. Related indexes [15, 14] are based on the intersections of two sets of q-grams: that in the pattern and that in its potential occurrence.

These indexes can also be built in linear time and need $O(n)$ space. Depending on q they achieve different space-time tradeoffs. In general, filtration indexes are much smaller than suffix trees (1 to 10 times the text size), although they work well for low error levels α: their search times are sublinear provided $\alpha = O(1/\log_\sigma n)$. A particularly interesting index with respect to space requirements is [25], because it does not index *all* the text q-grams. Rather, the q-samples selected are disjoint and there can be even some space among them. Using this technique the index can take even less space than the text, although the acceptable error level is reduced.

Intermediate Partitioning. Somewhat between the previous approaches are [19, 22], because they do not reduce the search to exact but to approximate search of pattern pieces, and use a neighborhood generating approach to search 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 indexes split the pattern in j pieces, each piece is searched in the index allowing $\lfloor k/j \rfloor$ errors and the approximate matches of the pieces are extended to complete pattern occurrences. The existing indexes differ in how j is selected (be it by indexing-time constraints [19] or by optimization goals [22]), and in the use of different data structures used to search the pieces with a neighborhood generating 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 shown to be probably impossible to surpass in [4]).

The idea of intermediate partitioning has given excellent results [22] and was shown to be an optimizing point between the extremes of neighborhood generating (that worsens as longer pieces are searched) and reduction to exact searching (that worsens as shorter pieces are searched). 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 match of the pattern, and search the pattern q-grams allowing $\lfloor k/j \rfloor$ errors in the index of text q-samples. This idea has been indeed proposed in [25] 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 [25] and then show how an index can be designed based upon this principle. We finally implement the index and show how it performs. The index has the advantage of taking little space and being an alternative tradeoff between neighborhood generation and reduction to exact searching. By selecting the interval h between the q-samples, the user is able to decide which of the two goals is more rlevant: saving space by a higher h or better performance for higher error levels, ensured by a lower h.

In particular, the scheme allows us to handle the problem posed by high error levels 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.

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 $ed(A,B) \le k$. Let $A = A_1 x_1 A_2 x_2 ... x_{j-1} A_j$, for strings A_i and x_i and for any $j \ge 1$. 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, at least one of the A_i's get no more than $\lfloor k/j \rfloor$ of them. Or put in another way, if each A_i appears inside B with not less than $\lfloor k/j \rfloor + 1 > k/j$ 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. Figure 2 illustrates.

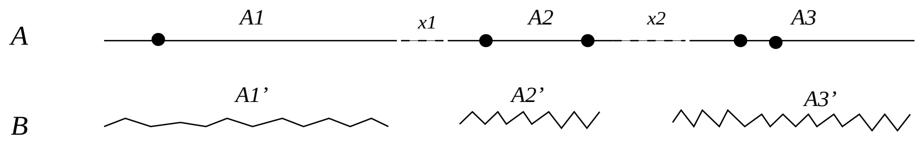


Fig. 2. Illustration of Lemma 1, where $k = 5$ and $j = 3$. At least one of the A_i's has at most one error (in this case A_1).

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 $h \ge q$, 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 $d_r = T_{h(r-1)+1..h(r-1)+q}$.

We need to guarantee that there are at least j text samples inside any occurrence of P. As an occurrence of P has minimal length $m - k$, the resulting condition on h is

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

(note that h has to be known at indexing time, when m and k are unknown, but in fact one can use a fixed h and adjust j at query time).

Figure 3 illustrates the idea, pointing out another fact not discussed until now. If the pattern P matches in 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 $Q_i = P_{(i-1)h+1...ih+q-1+k}$.

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

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

where

$$bed(u, Q) = \min_{Q' < Q} ed(u, Q')$$

(where $<$ denotes substring of). That is, $bed(u, Q)$ gives the best edit distance between u and a substring of Q. The text area corresponding to D_r is examined only if its cumulative best match distance is at most k.

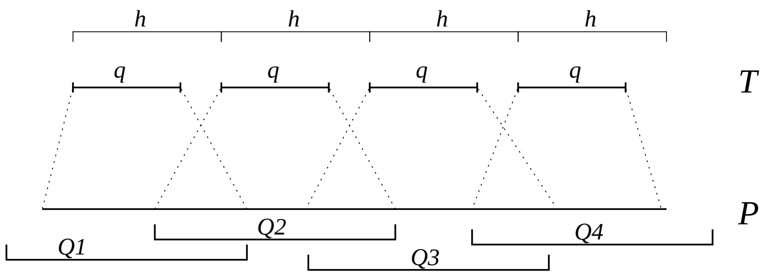


Fig. 3. Searching using q-samples, showing how the four relevant text samples at each position are aligned with the corresponding pattern blocks.

The algorithm works as follows. Each *counter* M_r, corresponding to the sequence $D_r = d_{r-j+1} \dots d_r$, indicates the number of errors produced by D_r. The counters are initialized to $M_r = j(e+1)$, were $q \ge e \ge \lfloor k/j \rfloor$ is unspecified by now. That is, we start by assuming that each text q-sample yields enough errors to disallow a match. Later, we can concentrate only on those that can be found in the pattern with at most e errors.

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

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

which is the set of possible 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 $M_r \le k$ are checked with dynamic programming. 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 $U_e^q(Q_i)$.

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 r such that $d_r \in U_e^q(Q_i)$. The first observation is that it is not necessary to generate all $U_e^q(Q_i)$, since we are interested only in the text q-samples (more specifically, in their positions). Rather, we generate

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

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, i.e. those that match inside Q_i with at most e errors.

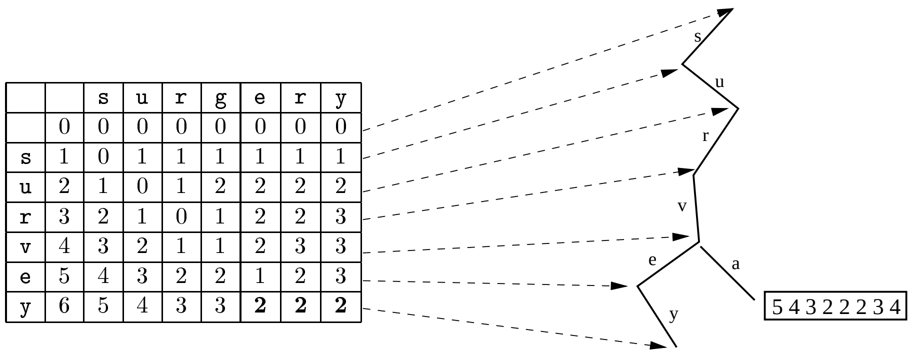
From now on we use $Q = Q_i$ and use i for other purposes. If considering a specific text q-sample $S = s_1 \ldots s_q$ (corresponding to some d_r), the problem is solved by the use of the dynamic programming algorithm explained in the Introduction, where the text is the pattern block Q and the pattern is the text q-sample S. That is, we fill a matrix $C_{0..q,0..|Q|}$ such that $C_{i,\ell}$ is the smallest edit distance between $S_{1..i}$ and a suffix of $Q_{1..\ell}$. When this matrix is filled, we have that the text q-sample S is relevant if and only if $C_{q,\ell} \le e$ 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, so this matrix will be filled row-wise instead of the typical on-line column-wise filling.

The algorithm works as follows. We perform an exhaustive search on the trie, starting at the root and entering into all the 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, i.e. the current row of the dynamic programming matrix. Upon entering into the children of the current node following an edge labeled with the letter c, a new row of C is computed from the current one using c as the next pattern letter. 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 reported. 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. However, some 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 keep true in descendant nodes. Therefore, at that point we can abandon that branch of the trie without actually considering its subtree.

Figure 4 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.

An alternative way to consider the problem is to model the search with a non-deterministic automaton (NFA). Consider the NFA for $e = 2$ errors shown in Figure 5. It is built for a fixed pattern block Q and is fed with the characters



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

of a text q-gram S. Every row denotes the number of errors seen (the first row zero, the second row one, etc.). Every column represents matching a prefix of S. Horizontal arrows represent matching a character (i.e. if the characters of S and Q match, we advance in S and in Q). All the others increment the number of errors (move to the next row): vertical arrows insert a character in S (we advance in Q but not in S), solid diagonal arrows replace a character (we advance in Q and S), and dashed diagonal arrows delete a character from S (they are ε-transitions, since we advance in S without advancing in Q). The initial set of ε-transitions allow a match of S to start anywhere inside Q. The q-gram prefix S' of S matches inside Q as long as there is an active state after considering all the characters of S'.

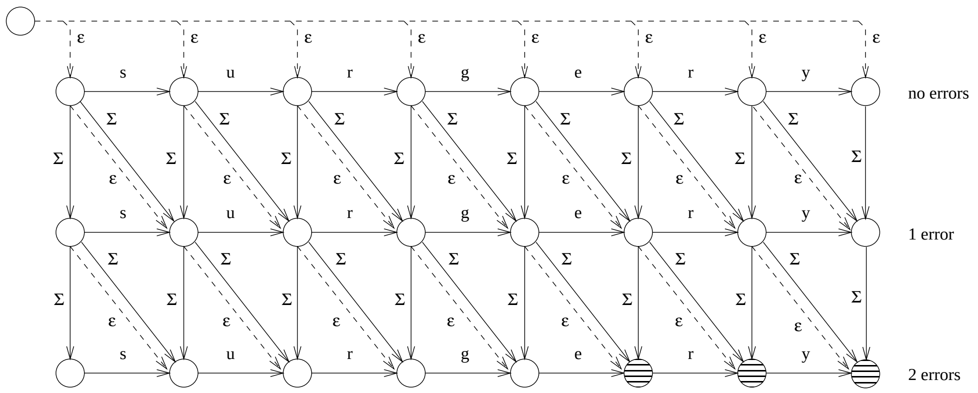


Fig. 5. An NFA for approximate string matching inside the pattern block Q = "**surgery**" with two errors. The shaded states are those active after considering the text q-sample "**survey**".

In [4, 22] it is shown that this NFA can be simulated using bit-parallelism, mapping each state to a bit in a computer word and updating all the states of a single computer word in $O(1)$ operation. The total time needed is $O(|Q|e/w)$ per node of the trie, where w is the number of bits in the computer word (cf. the dynamic programming $O(|Q|)$ time per trie node). The only change necessary to the simulation technique used in [22] is to start with all the states active to account for the initial ε-transitions, absent in [22]. Checking that there is an active state in the automaton is easily done (so the branch of the trie can be abandoned if there are no more active states). Finally, checking the exact number of errors of a match (i.e. finding the smallest row with an active state) is easily done in $O(e)$ time using bit masks. We use this simulation in our implementation.

4 The Parameters of the Problem

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 of the e-environments will be cheaper, 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 it has), so 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.

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. Equation (1) relates these parameters, introducing also m and k in the condition. In general m and k are not known at index construction time, while h and q have to be determined at that moment. Therefore, j must be adjusted at search time in order to make Eq. (1) hold. For a given query, j has a maximum acceptable value. As j grows, longer test sequences with less errors per piece 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 $O(\sigma^q + n/h)$. Having a smaller index puts restrictions in the allowed j values and, indirectly, on e.

5 Experimental Results

In the following experiments, the texts have been generated according to the symmetric Bernoulli model where each character occurs at the same probability, independently of other characters, like its predecessors.

Table 1 shows how the error level increases the number of processed columns, in cases of random text in 4- and 20-character alphabets. 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 the change in the value of e.

			$\sigma = 4$	$\sigma = 20$
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 1. Processed columns (in per cent) for $m = 40$, $q = h = 6$, and $n = 100,000$.

Altering the number of q-samples in test sequences D_r, i.e., the value of j, is related to changes in the values of h and q. This phenomenon lets us also to achieve more efficient filtration for higher error levels. Compare the results in Table 2 to those in Table 1.

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

Table 2. Processed columns (in per cent) for $\sigma = 4$, $m = 40$, and $n = 100,000$, for different values of h, q and j.

Table 3 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 the 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, like the length of the text and the implementation of the trie.

e	Columns	Traversed 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 3. Processed columns (in per cent) and the number of traversed nodes of the q-sample trie for $\sigma = 4$, $m = 40$, $k = 6$, $q = h = 6$, $j = 4$, and $n = 100,000$, for different values of e.

The distance h between the q-samples is crucial for the space requirement of the index. Table 4 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.

h	j	Columns
7	11	100.0
6	8	99.8
5	6	90.7
4	5	14.2
3	4	0.1

Table 4. Processed columns (in per cent) with a decreasing h, for $\sigma = 4$, $m = 40$, $k = 5$, $q = 3$, $e = 3$, $n = 100,000$. Note that the parameter j has to be adjusted according to h.

Since the index of the presented approach only stores non-overlapping q-samples, its space requirement is small, and can be kept below the size of the text [25]. This should be kept in mind when the performance is compared to other related approaches. Table 5 shows that the new approach works for a small error level almost as efficiently as its competitor [22] which, however, consumes more space; in fact, four times as much as the text does. It is obvious that an index which stores only a fraction of text portions cannot compete with one with more information on the text.

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 stores all the locations of all the q-grams of the text. Since the number

k	Alg. A	Alg. B
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 5. Processed columns (in per cent) for relatively low error levels. The new approach, denoted as A, collects *non-overlapping* q-samples, and an intermediate partitioning approach [22], denoted by B, stores *all* the text pieces which need to be searched for. The parameters are as follows: $\sigma = 4$, $m = 40$, $q = h = 6$ for algorithm A, $j = 4$, $e = 6$, and $n = 100,000$.

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, i.e.

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

Table 6 shows how the space saving factor decreases with an 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 6. Space saving factor v_r for $n = 100,000$.

6 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, whereas a high error level requires a more dense index. At the moment, the presented approach presumes non-overlapping q-samples ($h \geq q$). However, this is a question of parameterization. In the future, we will evaluate the impact of these parameters in different environments and problem instances, and enhance the scheme to allow also overlapping q-samples.

References

1. A. Apostolico and Z. Galil. *Combinatorial Algorithms on Words.* Springer-Verlag, New York, 1985.
2. M. Araújo, G. Navarro, and N. Ziviani. Large text searching allowing errors. In *Proc. WSP'97*, pages 2–20. Carleton University Press, 1997.
3. R. Baeza-Yates. Text retrieval: Theory and practice. In *12th IFIP World Computer Congress*, volume I, pages 465–476. Elsevier Science, September 1992.
4. R. Baeza-Yates and G. Navarro. Faster approximate string matching. *Algorithmica*, 23(2):127–158, 1999.
5. 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.
6. 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.
7. W. Chang and T. Marr. Approximate string matching and local similarity. In *Proc. CPM'94*, LNCS 807, pages 259–273, 1994.
8. A. Cobbs. Fast approximate matching using suffix trees. In *Proc. CPM'95*, pages 41–54, 1995. LNCS 937.
9. M. Crochemore. Transducers and repetitions. *Theoretical Computer Science*, 45:63–86, 1986.
10. M. Farach, P. Ferragina, and S. Muthukrishnan. Overcoming the memory bottleneck in suffix tree construction. In *Proc. SODA'98*, pages 174–183, 1998.
11. R. Giegerich, S. Kurtz, and J. Stoye. Efficient implementation of lazy suffix trees. In *Proc. WAE'99*, LNCS 1668, pages 30–42, 1999.
12. G. Gonnet. A tutorial introduction to Computational Biochemistry using Darwin. Technical report, Informatik E.T.H., Zuerich, Switzerland, 1992.
13. 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.
14. 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.
15. 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.
16. D. Knuth. *The Art of Computer Programming*, volume 3: Sorting and Searching. Addison-Wesley, 1973.

17. U. Manber and E. Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, pages 935–948, 1993.
18. U. Manber and S. Wu. GLIMPSE: A tool to search through entire file systems. In *Proc. USENIX Technical Conference*, pages 23–32, Winter 1994.
19. E. Myers. A sublinear algorithm for approximate keyword searching. *Algorithmica*, 12(4/5):345–374, Oct/Nov 1994.
20. G. Navarro. A guided tour to approximate string matching. Technical Report TR/DCC-99-5, Dept. of Computer Science, Univ. of Chile, 1999. To appear in *ACM Computing Surveys*. `ftp://ftp.dcc.uchile.cl/pub/users/gnavarro/-survasm.ps.gz`.
21. 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`.
22. G. Navarro and R. Baeza-Yates. A new indexing method for approximate string matching. In *Proc. CPM'99*, LNCS 1645, pages 163–186, 1999.
23. P. Sellers. The theory and computation of evolutionary distances: pattern recognition. *J. of Algorithms*, 1:359–373, 1980.
24. F. Shi. Fast approximate string matching with q-blocks sequences. In *Proc. WSP'96*, pages 257–271. Carleton University Press, 1996.
25. E. Sutinen and J. Tarhio. Filtration with q-samples in approximate string matching. In *Proc. CPM'96*, LNCS 1075, pages 50–61, 1996.
26. E. Ukkonen. Approximate string matching over suffix trees. In *Proc. CPM'93*, pages 228–242, 1993.
27. S. Wu and U. Manber. Fast text searching allowing errors. *Comm. of the ACM*, 35(10):83–91, October 1992.