SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 1(1), 1–4 (JANUARY 1988)

A Comparison of Approximate String Matching Algorithms

PETTERI JOKINEN, JORMA TARHIO, AND ESKO UKKONEN

Department of Computer Science, P.O. Box 26 (Teollisuuskatu 23), FIN-00014 University of Helsinki, Finland
(email: tarhio@cs.helsinki.fi)

SUMMARY

Experimental comparison of the running time of approximate string matching algorithms for the k differences problem is presented. Given a pattern string, a text string, and integer k, the task is to find all approximate occurrences of the pattern in the text with at most k differences (insertions, deletions, changes). We consider seven algorithms based on different approaches including dynamic programming, Boyer-Moore string matching, suffix automata, and the distribution of characters. It turns out that none of the algorithms is the best for all values of the problem parameters, and the speed differences between the methods can be considerable.

KEY WORDS String matching Edit distance k differences problem

INTRODUCTION

We consider the k *differences problem*, a version of the approximate string matching problem. Given two strings, *text* $T = t_1 t_2 \ldots t_n$ and *pattern* $P = p_1 p_2 \ldots p_m$ and integer k, the task is to find the end points of all approximate occurrences of P in T. An approximate occurrence means a substring P' of T such that at most k editing operations (insertions, deletions, changes) are needed to convert P' to P.

There are several algorithms proposed for this problem, see e.g. the survey of Galil and Giancarlo.[1] The problem can be solved in time $O(mn)$ by dynamic programming.[2,3] A very simple improvement giving $O(kn)$ expected time solution for random strings is described by Ukkonen.[3] Later, Landau and Vishkin,[4,5] Galil and Park,[6] Ukkonen and Wood[7] give different algorithms that consist of preprocessing the pattern in time $O(m^2)$ (or $O(m)$) and scanning the text in worst-case time $O(kn)$. Tarhio and Ukkonen[8,9] present an algorithm which is based on the Boyer-Moore approach and works in sublinear average time. There are also several other efficient solutions[10-17], and some[11-14] of them work in sublinear average time. Currently $O(kn)$ is the best worst-case bound known if the preprocessing time is allowed to be at most $O(m^2)$.

There are also fast algorithms[9,17-20] for the k mismatches problem, which is a reduced form of k differences problem so that a change is the only editing operation allowed.

It is clear that with such a multitude of different solutions to the same problem it is difficult to select a proper method for each particular approximate string matching task. The theoretical analyses given in the literature are helpful but it is important that the theory is completed with experimental comparisons extensive enough.

CCC 0038–0644/88/010001–04

Received 1 March 1988
Revised 25 March 1988

We will present an experimental comparison* of the running times of seven algorithms for the k differences problem. The tested algorithms are: two dynamic programming methods,[2, 3] Galil-Park algorithm,[6] Ukkonen-Wood algorithm,[7] an algorithm counting the distribution of characters,[18] approximate Boyer-Moore algorithm,[9] and an algorithm based on maximal matches between the pattern and the text.[10] (The last algorithm[10] is very similar to the linear algorithm of Chang and Lawler,[11] although they have been invented independently.) We give brief descriptions of the algorithms as well as an Ada code for their central parts. As our emphasis is in the experiments, the reader is advised to consult the original references for more detailed descriptions of the methods.

The paper is organized as follows. At first, the framework based on edit distance is introduced. Then the seven algorithms are presented. Finally, the comparison of the algorithms is represented and its results are summarized.

THE K DIFFERENCES PROBLEM

We use the concept of edit distance[21, 22] to measure the goodness of approximate occurrences of a pattern. The *edit distance* between two strings, A and B in alphabet Σ, can be defined as the minimum number of editing steps needed to convert A to B. Each editing step is a rewriting step of the form $a \to \varepsilon$ (a deletion), $\varepsilon \to b$ (an insertion), or $a \to b$ (a change) where a, b are in Σ and ε is the empty string.

The *k differences* problem is, given pattern $P = p_1 p_2 \dots p_m$ and text $T = t_1 t_2 \dots t_n$ in alphabet Σ of size c, and integer k, to find all such j that the edit distance (i.e., the number of differences) between P and some substring of T ending at t_j is at most k. The basic solution of the problem is the following dynamic programming method:[2, 3] Let D be an $m + 1$ by $n + 1$ table such that $D(i, j)$ is the minimum edit distance between $p_1 p_2 \dots p_i$ and any substring of T ending at t_j. Then

$$D(0, j) = 0, \quad 0 \le j \le n;$$

$$D(i, j) = \min \begin{cases} D(i-1, j) + 1 \\ D(i-1, j-1) + \quad \text{if } p_i = t_j \text{ then 0 else 1} \\ D(i, j-1) + 1 \end{cases}$$

Table D can be evaluated column-by-column in time $O(mn)$. Whenever $D(m, j)$ is found to be at most k for some j, there is an approximate occurrence of P ending at t_j with edit distance $D(m, j) \le k$. Hence j is a solution to the k differences problem.

In Fig. 1 there is an example of table D for T = `bcbacbbb` and P = `cacd`. The pattern occurs at positions 5 and 6 of the text with at most 2 differences.

All the algorithms presented work within this model, but they utilize different approaches in restricting the number of entries that are necessary to evaluate in table D. Some of the algorithms work in two phases: scanning and checking. The scanning phase searches for potential occurrences of the pattern, and the checking phase verifies if the suggested occurrences are good or not. The checking is always done using dynamic programming.

* The comparison was carried out in 1991. Some of the newer methods will likely be faster than the tested algorithms for certain values of problem parameters.

		0	1	2	3	4	5	6	7	8
			b	c	b	a	c	b	b	b
0		0	0	0	0	0	0	0	0	0
1	c	1	1	0	1	1	0	1	1	1
2	a	2	2	1	1	1	1	1	2	2
3	c	3	3	2	2	2	1	2	2	3
4	d	4	4	3	3	3	2	2	3	3

Figure 1. Table D.

ALGORITHMS

Dynamic programming

We consider two different versions of dynamic programming for the k differences problem. In the previous section we introduced the trivial solution which computes all entries of table D. The code of this algorithm is straight-forward,[2, 21] and we do not present it here. In the following, we refer to this solution as Algorithm DP.

Diagonal h of D for $h = -m, \ldots, n$, consists of all $D(i, j)$ such that $j - i = h$. Considering computation along diagonals gives a simple way to limit unnecessary computation. It is easy to show that entries on every diagonal h are monotonically increasing.[22] Therefore the computation along a diagonal can be stopped, when the threshold value of $k + 1$ is reached, because the rest of the entries on that diagonal will be greater than k. This idea leads to Algorithm EDP (Enhanced Dynamic Programming) working in average time[3] $O(kn)$. Algorithm EDP is shown in Fig. 2.

In algorithm EDP, the text and the pattern are stored in tables T and P. Table D is evaluated a column at a time. The entries of the current column are stored in table h, and the value of $D(i-1, j-1)$ is temporarily stored in variable C. A work space of $O(m)$ is enough, because every $D(i, j)$ depends only on entries $D(i-1, j)$, $D(i, j-1)$, and $D(i-1, j-1)$. Variable *Top* tells the row where the topmost diagonal still under the threshold value $k+1$ intersects the current column. On line 12 an approximate occurrence is reported, when row m is reached.

Galil-Park

The $O(kn)$ algorithm presented by Galil and Park[6] is based on the diagonalwise monotonicity of the entries of table D. It also uses so-called reference triples that represent matching substrings of the pattern and the text. This approach was used already by Landau and Vishkin.[4] The algorithm evaluates a modified form of table D. The core of the algorithm is shown in Fig. 3 as Algorithm GP.

In preprocessing of pattern P (procedure call *Prefixes*(P) on line 2), upper triangular table *Prefix*(i, j), $1 \leq i < j \leq m$, is computed where *Prefix*(i, j) is the length of the longest common prefix of $p_i \ldots p_m$ and $p_j \ldots p_m$.

Reference triple (u, v, w) consists of start position u, end position v, and diagonal w such that substring $t_u \ldots t_v$ matches substring $p_{u-w} \ldots p_{v-w}$ and $t_{v+1} \neq p_{v+1-w}$. Algorithm GP manipulates several triples; the components of the r^{th} triple are presented as $U(r)$, $V(r)$, and $W(r)$.

```
1   begin
2       Top := k + 1;
3       for I in 0 .. m loop H(I) := I; end loop;
4       for J in 1 .. n loop
5           C := 0;
6           for I in 1 .. Top loop
7               if P(I) = T(J) then E :=C;
8               else E := Min((H(I – 1), H(I), C)) + 1; end if;
9               C := H(I); H(I) := E;
10          end loop;
11          while H(Top) > k loop Top := Top – 1; end loop;
12          if Top = m then Report_Match(J);
13          else Top := Top + 1; end if;
14      end loop;
15  end;
```

Figure 2. Algorithm EDP.

For diagonal d and integer e, let $C(e, d)$ be the largest column j such that $D(j - d, j) = e$. In other words, the entries of value e on diagonal d of D end at column $C(e, d)$. Now $C(e, d) = Col + Jump(Col + 1 - d, Col + 1)$ holds where

$$Col = \max\{C(e-1, d-1) + 1, C(e-1, d) + 1, C(e-1, d+1)\}$$

and *Jump*(i, j) is the length of the longest common prefix of $p_i \ldots p_m$ and $t_j \ldots t_n$ for all i, j.

Let *C-diagonal* g consist of entries $C(e, d)$ such that $e + d = g$. For every C-diagonal Algorithm GP performs an iteration that evaluates it from two previous C-diagonals (lines 7–38). The evaluation of each entry starts with evaluating the *Col* value (line 11). The rest of the loop (lines 12–35) effectively finds the value *Jump*$(Col + 1 - d, Col + 1)$ using the reference triples and table *Prefix*. A new C-value is stored on line 24.

The algorithm maintains an ordered sequence of reference triples. The sequence is updated on lines 28–35. Procedure *Within*(d) called on line 14 tests if text position d is within some interval of the k first reference triples in the sequence. In the positive case, variable R is updated to express the index of the reference triple whose interval contains text position d.

A match is reported on line 26.

Instead of the whole C defined above, table C of the algorithm contains only three successive C-diagonals. The use of this buffer of three diagonals is organized with variables $B1$, $B2$, and $B3$.

Ukkonen-Wood

Another $O(kn)$ algorithm, given by Ukkonen and Wood,[7] has an overall structure identical to the algorithm of Galil and Park. However, no reference triples are used. Instead, to find the necessary values *Jump*(i, j), the text is scanned with a modified suffix automaton for

```
1   begin
2       Prefixes(P);
3       for I in –1 .. k loop
4           C(I, 1) := –Infinity; C(I, 2) := –1;
5       end loop;
6       B1:=0; B2:=1; B3:=2;
7       for J in 0 .. n – m + k loop
8           C(–1, B1) := J; R := 0;
9           for E in 0 .. k loop
10              H := J – E;
11              Col := Max((C(E–1, B2) + 1, C(E–1, B3) + 1, C(E–1, B1)));
12              Se := Col + 1; Found := false;
13              while not Found loop
14                  if Within(Col + 1) then
15                      F := V(R) – Col; G := Prefix(Col+1–H, Col+1–W(R));
16                      if F = G then Col := Col + F;
17                      else Col := Col + Min(F, G); Found := true; end if;
18                  else
19                      if Col – H < m and then P(Col+1–H) = T(Col+1) then
20                          Col := Col + 1;
21                      else Found := true; end if;
22                  end if;
23              end loop;
24              C(E, B1) := Min(Col, m+H);
25              if C(E, B1) = H + m and then C(E–1, B2) < m + H then
26                  Report_Match((H + m));
27              end if;
28              if V(E) >= C(E, B1) then
29                  if E = 0 then U(E) := J + 1;
30                  else U(E) := Max(U(E), V(E–1) + 1); end if;
31              else
32                  V(E) := C(E, B1); W(E) := H;
33                  if E = 0 then U(E) := J + 1;
34                  else U(E) := Max(Se, V(E–1) + 1); end if;
35              end if;
36          end loop;
37          B := B1; B1 := B3; B3 := B2; B2 := B;
38      end loop;
39  end;
```

Figure 3. Algorithm GP.

```
1   begin
2       Prefixes(P);
3       for I in – 1 .. k loop
4           C(I, 1) := –Infinity; C(I, 2) := –1;
5       end loop;
6       B1 := 0; B2 := 1; B3 := 2;
7       for J in 0 .. n – m + k loop
8           C(–1, B1) := J;
9           for E in 0 .. k loop
10              H := J – E;
11              Col := Max((C(E – 1, B2) + 1,
12              C(E – 1, B3) + 1, C(E – 1, B1)));
13              C(E, B1) := Col + Jump(Col + 1 – H, Col + 1);
14              if C(E, B1) = H + m and then C(E – 1, B2) < m + H then
15                  Report_Match(H + m);
16              end if;
17          end loop;
18          B := B1; B1 := B3; B3 := B2; B2 := B;
19      end loop;
20  end;
```

Figure 4. Algorithm UW.

pattern P. The core of the resulting method, called Algorithm UW, is shown in Fig. 4.

Table *Prefix* is as in Algorithm GP. Procedure call *Jump*$(Col + 1 - H, Col + 1)$ on line 13 returns the *Jump* value, as in Algorithm GP. The value is evaluated as

$$\textit{Jump}(i, j) = \min(\textit{Prefix}(i, Mn(j)),\ \textit{Maxprefix}(j))$$

where *Maxprefix*(j) equals the length of the longest common prefix of $t_j \ldots t_n$ and $p_{Mn(j)} \ldots p_m$, and $1 \leq Mm(j) \leq m$ is such that the length of this common prefix is maximal. For each text position j, the values of *Maxprefix*(j) and $Mm(j)$ are produced in a left-to-right scan over T by a suffix automaton for P.

The suffix automaton is constructed during the preprocessing phase of Algorithm UW. The construction is a modification of the suffix automata constructions of Crochemore[23] and Blumer et al.[24]

Speed-up heuristic based on the distribution of characters

Grossi and Luccio[18] present an algorithm for the k mismatches problem, where the change is the only editing operation allowed. The key idea is to search for substrings of the text whose distribution of characters differs from the distribution of characters in the pattern at most as much as it is possible under k differences. In the following we present how the same approach can also be applied to the k differences problem.

```
1   begin
2       for I in 1 .. m loop C(P(I)) := C(P(I)) + 1; end loop;
3       for J in 1 .. n loop
4           X := T(J); EnQueue(Q, X); C(X) := C(X) – 1;
5           if C(X) < 0 then Z := Z + 1; end if;
6           while Z > k loop
7               DeQueue(Q, X);
8               if C(X) < 0 then Z := Z – 1; end if;
9               C(X) := C(X) + 1;
10          end loop;
11          if Size(Q) = m then
12              Mark(J – m + 1); DeQueue(Q, X);
13              if C(X) < 0 then Z := Z – 1; end if;
14              C(X) := C(X) + 1;
15          end if;
16      end loop;
17      EDP(m);
18  end;
```

Figure 5. Algorithm DC.

Algorithm DC in Fig. 5 works in two main phases: scanning and checking. The scanning phase (lines 3–16) scans over the text and marks the parts that may contain approximate occurrences of P. This is done by marking on line 12 some diagonals of D.

The checking phase (line 17) evaluates all marked diagonals using Algorithm EDP restricted to the marked diagonals. Whenever EDP refers to an entry outside the diagonals, the entry can be taken to be ∞. Parameter x of call *EDP*(x) tells how many columns should be evaluated for one marked diagonal. The minimum value m for x is applicable for DC.

The scanning phase is almost identical to the original algorithm.[18] It maintains queue Q, which corresponds to a substring of the text with at most m characters. If $f(x)$ and $q(x)$ are frequencies of character x in the pattern and in Q, variable Z has the value

$$\sum_{x \text{ in } Q} \max(q(x) - f(x),\ 0).$$

The value of Z is computed together with table C which maintains the difference $f(x) - q(x)$ for every x. When Z is greater than k, we know for sure that no approximate occurrence of the pattern with at most k differences can contain substring held by Q. This is because the value of Z is always at most as large as the number of differences between P and the substring of T that corresponds to the current Q. Items are inserted to Q until $Z > k$ or Q contains m characters. In the latter case a potential approximate occurrence has been found, which is marked on line 12.

```
1   function Maxprefix(I: positive) return integer is
2   begin
3       State := Initial_State; R := I; D:= 0;
4       while State.Go_To(T(R)) /= null loop
5           State := State.Go_To(T(R));
6           R := R + 1; D := D + 1;
7       end loop;
8       return D;
9   end Maxprefix;
10
11  begin
12      Create; I := –1; S := 0;
13      while I < n loop
14          S := S + 1; Bound := I + 1 + Maxprefix(I + 2);
15          while I < Bound loop
16              I := I + 1; H(I) := S;
17          end loop;
18      end loop;
19      for I in 1 .. n – m + k + 1 loop
20          if H(I + m – k – 1) – H(I) – 1 <= k then Mark(I); end if;
21      end loop;
22      EDP(m + k);
23  end;
```

Figure 6. Algorithm MM.

Speed-up heuristic based on maximal matches

Our next algorithm, described and analyzed in detail by Ukkonen,[10] has the same two phases (scanning and checking) as Algorithm DC. The scanning part again marks the potential approximate occurrences of P. Now they are found using a heuristic measure that tries to take into account the relative order of the characters. Note that the order is totally ignored in the character distribution heuristic of Algorithm DC. That heuristic gives the same estimate of the number of differences for all permutations of P and all permutations of the corresponding text portion.

The heuristic is based on the observation that the smaller is the number of the differences between P and its occurrence P', the longer portions of P and P' must have exact matches. Whenever there are long exact matches between P and a substring of T, then the substring is a potential occurrence of P.

To make this precise, let again for $1 \leq j \leq n$,

$$\mathit{Maxprefix}(j) = \max_{1 \leq i \leq m} \{\text{length of the longest common prefix of } t_j \ldots t_n \text{ and } p_i \ldots p_m\},$$

and let $\mathit{Maxprefix}(n + 1) = 0$. Let $\mathit{Bound}(1) = \mathit{Maxprefix}(1)$ and let $H(i) = 1$ for $1 \leq i \leq$

Bound(1). Generally, if *Bound*$(s) = B$ and $B < n$, then

$$H(i) = s + 1 \text{ for } B + 1 \leq i \leq B + 1 + \textit{Maxprefix}(B + 2),$$

and

$$\textit{Bound}(s + 1) = B + 1 + \textit{Maxprefix}(B + 2).$$

Informally, assume that we try match P against T in a greedy fashion by taking as long matches as possible. Starting from p_1 and t_1, determine the longest common prefix of the suffixes of P and $t_1 \ldots t_n$. When a mismatch is found at t_j, we continue by searching for the longest common prefix of the suffixes of P and $t_{j+1} \ldots t_n$, and so on. Then $H(i)$ tells the number of discontinuation positions in $t_1 \ldots t_i$, i.e., how many times a new longest common prefix is applied or there is no common prefix at all.

It is easy to show that there can be an approximate occurrence of P with $\leq k$ differences starting from t_i only if

$$H(i + m - k - 1) - H(i) - 1 \leq k.$$

The scanning phase finds values $H(i)$, $1 \leq i \leq n$, and marks for checking the indexes i that satisfy the above condition. The resulting method, called Algorithm MM, is given in Figures 6 and 7.

The computation of values $H(i)$ reduces to values *Maxprefix*(i) (line 14 of Fig. 6). These are found by scanning T with a similar but somewhat simpler suffix automaton for P as was used in algorithm UW. The construction of the automaton, essentially from Crochemore,[23] is given in Fig. 7. The automaton consists of the initial state, other states (of type Node), and goto and fail transitions between them. Value *Maxprefix*(i) is found simply by scanning T from t_i with the automaton using only goto transitions (lines 1–9 of Fig. 6). When the first t_j is encountered such that there is no matching goto step for t_j, then *Maxprefix*$(i) = j - i$.

Lines 13–17 of Algorithm MM compute the $H(i)$ values, and line 20 tests and marks the appropriate entries. This takes linear time. The final checking is again done by EDP on line 22.

Boyer-Moore approach

The characteristic feature of the Boyer-Moore algorithm[25] for exact matching of string patterns is the right-to-left scan over the pattern. At each alignment of the pattern with the text, characters of the text below the pattern are examined from right to left, starting by comparing the rightmost character of the pattern with the character currently below it in the text. Between alignments, the pattern is shifted to the right along the text.

Tarhio and Ukkonen[9] have developed an approximate string matching algorithm which is a generalization of the Horspool version[26] of the Boyer-Moore algorithm. A slightly modified version called Algorithm ABM is presented in Fig. 8. The algorithm has scanning and checking phases. The scanning phase based on a Boyer-Moore idea repeatedly applies two operations: mark and shift. Checking is done by EDP as in the case of algorithms DC and MM. In the case of ABM, $2m + k$ columns have to be checked for every marked diagonal.

Let us consider an alignment of pattern P with text T. In deciding whether a diagonal corresponding to this alignment should be marked, we search for bad characters. Let p_i be above t_j. Character t_j in T is bad (for this alignment), if the distance from i to the closest occurrence of t_j in P is more than k (when $k = 0$, a bad character is simply a mismatch). It can be shown that no approximate occurrence of P in T may contain more than k bad characters.

```
1   function Create
2   begin
3       Root := new Node; Root.Depth := 0; Top := Root;
4       for I in 1 .. m loop
5           New_Node := new Node; Next := Top;
6           while Next /= Root and Next.Go_To(P(I)) = null loop
7               Next.Go_To(P(I)) := New_Node;
8               Next := Next.Fail;
9           end loop;
10          if Next.Go_To(P(I)) = null then
11              Root.Go_To(P(I)) := New_Node;
12              New_Node.Fail := Root;
13          elsif Next.Depth + 1 = Next.Go_To(P(I)).Depth then
14              New_Node.Fail := Next.Go_To(P(I));
15          else
16              R := new Node;
17              R.Fail := Next.Go_To(P(I)).Fail;
18              R.Go_To := Next.Go_To(P(I)).Go_To;
19              R.Depth := Next.Depth + 1; New_Node.Fail := R;
20              Next.Go_To(P(I)).Fail := R;
21              loop
22                  Next.Go_To(P(I)) := R; Next := Next.Fail;
23                  exit when Next = null or else
24                  Next.Go_To(P(I)).Depth < R.Depth;
25              end loop;
26          end if;
27          New_Node.Depth := Top.Depth + 1; Top := New_Node;
28      end loop;
29      Initial_State := Root;
30  end Create;
```

Figure 7. Preprocessing of P for Algorithm MM.

So at every alignment of the pattern we search for bad characters proceeding from right to left until $k + 1$ of them were found or the whole pattern was traversed. In the former case there is no approximate occurrence at the current alignment and in the latter case a potential approximate occurrence has been found.

For determining the length of the shift, i.e. what is the next potential diagonal after h for marking, we search for the first diagonal after h, where at least one of the characters $t_{h+m}, t_{h+m-1}, \ldots, t_{h+m-k}$ matches with the corresponding character of P. This can be done with a precomputed $(k + 1) \times c$ table d defined for each $i = m - k, \ldots, m$, and for each a in S such that

$$d[i, a] = \min\{s \mid s = m \text{ or } (1 \leq s < m \text{ and } p_{i-s} = a)\}.$$

```
1   begin
2       Initialize;
3       J := m;
4       while J <= n + k loop
5           R := J; I := m;
6           Bad_Chars := 0; Shift := m;
7           while I > 0 and Bad_Chars <= k loop
8               if I >= m – k then
9                   Shift := Min(Shift, d(I, T(R)));
10              end if;
11              if Bad(T(R), I) then
12                  Bad_Chars := Bad_Chars + 1;
13              end if;
14              I := I – 1; R := R – 1;
15          end loop;
16          if Bad_Chars <= k then
17              Mark(J – m – k + 1);
18              J := J + m + 1;
19          else J := J + Max(Min(k + 1, I + 1), Shift);
20          end if;
21      end loop;
22      EDP(2 * m + k);
23  end;
```

Figure 8. Algorithm ABM.

In Algorithm ABM the minimum of $d[m-k, t_{h+m-k}], \ldots, d[m, t_{h+m}]$ is computed to variable *Shift* simultaneously with the inspection of the bad characters. There are two cases in determining the final value of shift. If a diagonal is marked, the length of the shift is $m+1$ (on line 18), which ensures that no text position is reinspected after marking (the value of *Shift* is not used in this case). This heuristic is correct as $2m+k$ columns of D are checked by EDP (line 22), starting from the first column that crosses a marked diagonal. If the diagonal is not marked, the length of the shift is the maximum of the value of variable *Shift* and the minimum of $k+1$ and $I+1$, where variable I tells the number of positions not inspected at the current alignment.

In order to be able to handle correctly an approximate occurrence of P in the end of text, it is assumed that k additional dummy characters have been appended to the text.

For finding the bad characters fast we use precomputed table *Bad*(i, a), $1 \leq i \leq m$, $a \in \Sigma$, such that *Bad*(i, a) = true, if and only if a does not appear in $p_{i-k} \ldots p_{i+k}$, where $p_j = \varepsilon$ for $j < 1$ and $j > m$.

The computation of tables *Bad* and d from P and S is straightforward. An efficient implementation for computing d is given in Fig. 9, where we assume alphabet Σ consists of codes $1, \ldots, c$. This is a corrected form of the original preprocessing algorithm.[9]

```
1   begin
2       for J in 1 .. c loop
3           ready(J) := m + 1;
4       end loop;
5       for J in 1 .. c loop
6           for I in m–k .. k loop
7               d(I, J) := m;
8           end loop;
9       end loop;
10      for I in reverse 1 .. m–1 loop
11          for J in reverse Max(I+1, m–k) .. ready(P(I))–1 loop
12              P(I)) := J – I;
13          end loop;
14          ready(P(I)) := Max(I+1, m–k)
15      end loop;
16  end
```

Figure 9. Computation of table d for Algorithm ABM.

COMPARISON

We performed an extensive test program on all seven algorithms DP, EDP, GP, DC, UW, MM, and ABM described in the previous sections. The results are presented in Table I. In our tests, we used random patterns of varying lengths and random texts of length 100,000 characters over alphabets of different sizes. Besides random texts we ran a test series on a text of English fiction. The tests were run on a Vaxstation 3100 under VMS. In order to decrease random variation, the figures of Table I are averages of ten runs with different patterns. All times are total execution times including preprocessing, scanning, and checking. The best value on each row is set in boldface.

All patterns and texts were generated independently, because if the patterns had been selected from the texts, the scanning speeds of some algorithms would have depended on the length of the text. In the case of the fiction text, a 100,000 characters long portion was used as a text, and the patterns were selected from another portion starting from a character preceded by a space.

We tested the effect of m in a test series with varying $m = 8, 16, 32, 64, 128$ and fixed $k = 4$. We examined the effect of k in two test series. In one series we evaluated the effect of absolute error with parameters $k = 0, 1, 2, 4, 6, 8$ and with fixed $m = 10$, and in the other series we tested fixed relative error of $k = m/8$ with $m = 8, 16, 32, 64, 128$. All tests were performed for alphabet sizes $c = |\Sigma| = 2, 4, 10, 30, 90$ and for the English text.

The algorithms with two phases (scanning and checking) were implemented so that they perform two subsequent passes over T. An alternative way would be to merge the two phases into a one-pass algorithm. In practice this is more desirable form of the algorithms as then we need a buffer of length $O(m)$ only to save the relevant part of T. Therefore we also tried (e.g. for ABM) that type of implementations. In our tests the approach of two passes was about 10 percent faster, but, on the other hand, it consumes more space.

Table I. Execution times (in units of 10 milliseconds) of the algorithms (n = 100,000). Value E of c denotes the English text.

c	m	k	DP	GP	UW	EDP	DC	MM	ABM
2	8	4	**778**	2648	2970	807	1157	1129	1095
2	16	4	1440	2847	2957	**1352**	1691	1716	1677
2	32	4	2803	2836	2996	**1479**	1819	1876	1823
2	64	4	5565	2775	2757	1493	1878	**155**	1854
2	128	4	11108	2794	2899	1379	1573	**144**	1695
2	10	0	919	596	917	343	495	370	**305**
2	10	1	920	1149	1407	596	**760**	867	880
2	10	2	924	1716	1913	**809**	1077	1112	1080
2	10	4	**949**	2771	3044	988	1356	1314	1280
2	10	6	**951**	3654	4138	991	1361	1320	1300
2	10	8	**950**	4537	5203	989	1359	1318	1313
2	8	1	751	1114	1356	**559**	750	845	848
2	16	2	1433	1681	1914	**861**	1175	1165	1142
2	32	4	2802	2795	2994	**1426**	1741	1819	1764
2	64	8	5564	4993	4625	**2653**	3281	3220	3166
2	128	16	11102	9946	8903	**5102**	6038	6047	5829
4	8	4	**906**	2652	2876	923	1283	1261	1227
4	16	4	1708	2675	2959	**1047**	1279	1387	1320
4	32	4	3347	2702	3026	1072	1257	**244**	1378
4	64	4	6650	2619	2811	1051	933	**147**	1507
4	128	4	13296	2694	2940	1047	597	**148**	1694
4	10	0	1077	536	935	263	213	164	**101**
4	10	1	1087	1063	1435	457	**393**	488	471
4	10	2	1087	1585	1921	**631**	754	916	897
4	10	4	1093	2691	2928	**1019**	1379	1358	1322
4	10	6	**1114**	3695	4014	1150	1538	1505	1479
4	10	8	**1117**	4617	5134	1156	1548	1514	1505
4	8	1	877	1027	1371	444	**405**	673	468
4	16	2	1707	1567	1951	644	681	**564**	978
4	32	4	3341	2699	3033	1065	1207	**239**	1392
4	64	8	6663	4791	4655	1858	2150	**150**	2291
4	128	16	13342	9624	8959	3653	4375	**149**	4313
10	8	4	988	2437	2758	**860**	1060	1202	1166
10	16	4	1917	2445	2952	864	**678**	1129	896
10	32	4	3774	2478	3044	861	306	**157**	444
10	64	4	7520	2402	2848	860	197	**156**	440
10	128	4	15024	2500	2953	863	190	**156**	432
10	10	0	1218	506	910	237	187	165	**49**
10	10	1	1221	987	1411	383	195	175	**134**
10	10	2	1220	1479	1897	542	**264**	664	279
10	10	4	1222	2479	2875	**869**	958	1203	1185
10	10	6	1225	3515	3858	**1175**	1541	1544	1520
10	10	8	**1250**	4539	4968	1293	1698	1676	1652
10	8	1	990	955	1338	383	200	287	**133**
10	16	2	1918	1447	1944	539	205	**163**	232
10	32	4	3768	2489	3045	865	301	**157**	484
10	64	8	7524	4448	4684	1510	777	**157**	1737
10	128	16	15035	8999	8964	2886	3028	**156**	3400

c	m	k	DP	GP	UW	EDP	DC	MM	ABM
30	8	4	1031	2325	2692	782	**333**	1128	464
30	16	4	1999	2333	2897	781	**189**	287	192
30	32	4	3960	2393	3012	779	188	**167**	179
30	64	4	7906	2315	2841	777	189	**165**	194
30	128	4	15794	2415	2958	779	188	**166**	188
30	10	0	1275	500	855	232	188	176	**36**
30	10	1	1276	965	1351	363	188	176	**73**
30	10	2	1275	1442	1842	500	188	226	**117**
30	10	4	1274	2373	2808	779	**229**	1112	287
30	10	6	1273	3336	3776	1069	**788**	1440	1079
30	10	8	**1287**	4328	4798	1321	1682	1714	1683
30	8	1	1031	936	1282	363	188	183	**86**
30	16	2	1997	1405	1893	500	187	171	**94**
30	32	4	3958	2395	3014	779	188	**167**	185
30	64	8	7903	4210	4665	1357	188	**165**	315
30	128	16	15807	8546	8977	2534	190	**165**	486
90	8	4	1044	2300	2657	750	193	1097	**189**
90	16	4	2023	2305	2852	749	188	183	**110**
90	32	4	4026	2371	2957	747	188	179	**81**
90	64	4	8045	2298	2771	748	188	176	**72**
90	128	4	16089	2396	2940	749	189	176	**74**
90	10	0	1291	497	826	230	187	185	**33**
90	10	1	1290	960	1318	357	188	184	**59**
90	10	2	1289	1432	1815	486	187	186	**88**
90	10	4	1290	2346	2776	748	188	1084	**149**
90	10	6	1290	3285	3746	1015	**230**	1389	290
90	10	8	1293	4225	4722	1285	**928**	1682	1162
90	8	1	1045	932	1252	357	188	186	**73**
90	16	2	2021	1393	1848	485	187	182	**61**
90	32	4	4027	2373	2960	749	188	179	**82**
90	64	8	8039	4114	4589	1281	189	**176**	185
90	128	16	16091	8225	8959	2377	189	**175**	292
E	8	4	1012	2370	2730	817	**755**	1158	967
E	16	4	1964	2393	2923	836	**250**	883	359
E	32	4	3882	2423	3022	805	188	**164**	273
E	64	4	7724	2359	2838	823	188	**164**	292
E	128	4	15443	2460	2972	832	189	**168**	311
E	10	0	1251	500	888	234	188	171	**43**
E	10	1	1253	972	1381	370	189	185	**101**
E	10	2	1253	1456	1874	515	203	493	**183**
E	10	4	1246	2431	2850	834	**626**	1166	902
E	10	6	1256	3430	3825	**1137**	1344	1506	1492
E	10	8	**1273**	4445	4891	1314	1718	1707	1685
E	8	1	1014	943	1313	371	190	246	**111**
E	16	2	1968	1418	1919	512	188	169	**151**
E	32	4	3868	2453	3023	843	189	**164**	286
E	64	8	7755	4370	4689	1473	192	**164**	430
E	128	16	15472	8799	8957	2753	206	**167**	639

The tests show that algorithms DP, GP, and UW are considerably slower than the other four algorithms and they can almost never win the best of the four. Algorithm DP is the best if m is very small or k is close to m; the advantage of DP in such cases is, however, only marginal both in theory and in practice.

The behavior of the $O(kn)$ algorithms GP and UW is mutually rather similar, mostly GP being the faster one in our experiments. Both GP and UW show $O(kn)$ behavior but, unfortunately, with relative high overhead. Therefore they are often slower than the trivial solution DP.

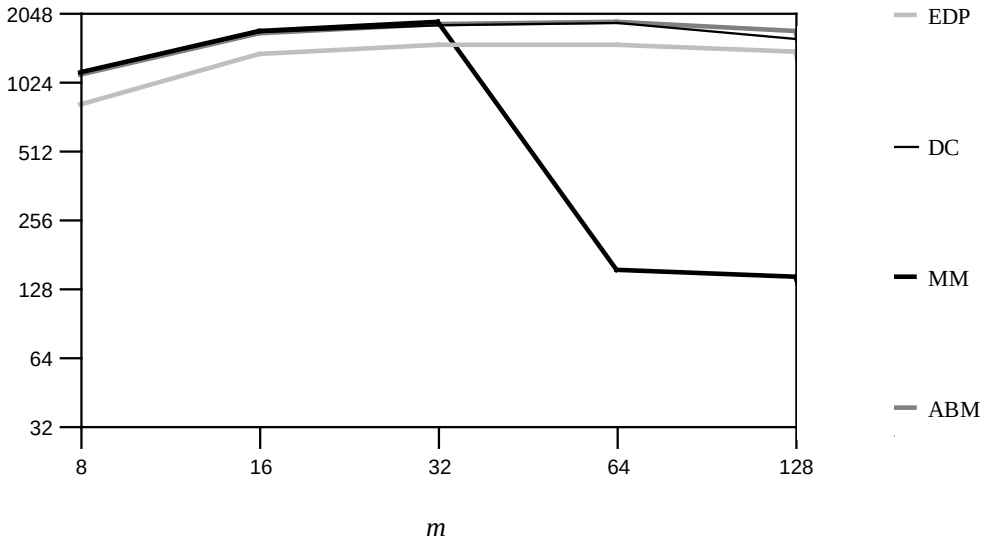

Figure 10. Execution times for c = 2 and k = 4.

Because algorithms EDP, DC, MM, and ABM were better than the others, we studied relations of their execution times more carefully. Figure 10 shows the total execution times of EDP, DC, MM, and ABM when $k = 4$ and m varies for alphabet size $c = 2$. Algorithm EDP is the best for $m \leq 32$, MM for $m > 32$. Figure 11 shows the corresponding times when $k = m/8$ for alphabet size $c = 10$. Now ABM is the best for $k < 2$ and MM for $k \leq 2$. In Figure 12 $m = 10$ and k varies for alphabet size $c = 30$. Algorithm ABM is the best for $k \leq 3$ and DC for $3 < k \leq 7$ and EDP for $k > 7$. Figure 13 presents the situation where $k = 4$ and m varies for alphabet size $c = 90$. This time ABM is clearly the best for all tested $m \leq 8$.

Figure 14 is based on a larger test series. It shows roughly the fastest algorithm for different values of m ($\leq$ 64) and k in the cases of the binary text and the English text. In the case of the binary alphabet, DC is only slightly slower than EDP for small values of k.

Based on similar classification in all relevant alphabets, it is possible to build a hybrid algorithm of EDP, DC, MM, and ABM, which selects on the basis of values c, m, and k which method to use. Because DC, MM, and ABM already contain EDP as their subroutine, it would be easy to incorporate such a selection between the advanced method and EDP to make the total algorithm faster. Particularly the variation of the execution times for ABM can

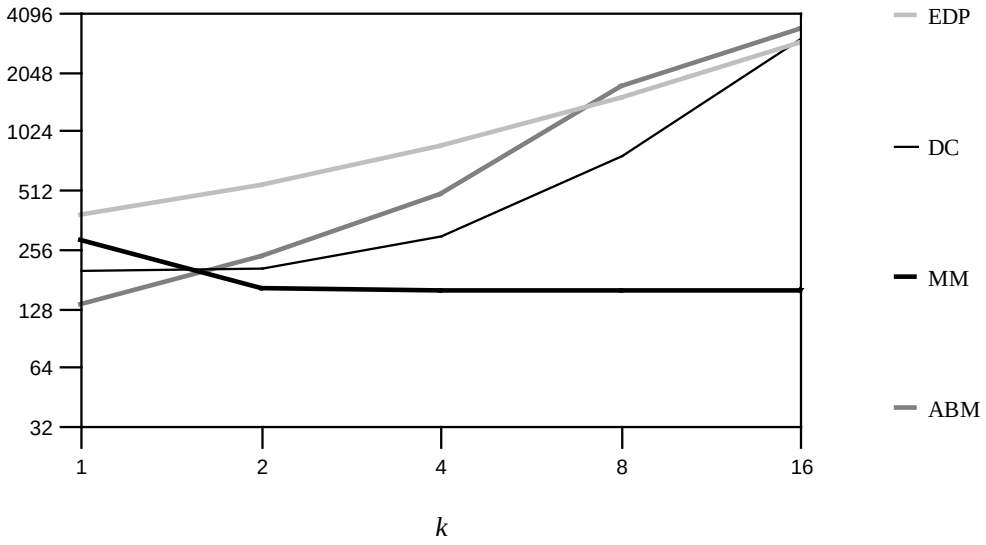

Figure 11. Execution times for c = 10 and $k = m / 8$.

be decreased by widening the belt of diagonals that are evaluated for each marking and by correspondingly increasing the length of the shift.

In order to get some impression of the behavior of the algorithms on other computer systems, we tested EDP and ABM programmed in Pascal on Sun 4/260S. The execution times of EDP and ABM on Sun (shown in Table II for some parameter values) were on the average 68 per cent and 60 per cent, respectively, of the corresponding times on Vaxstation. Though ABM seems to work relatively slightly better on Sun than on Vaxstation, the results were in general very similar.

CONCLUSIONS

The string matching algorithms are algorithmic miniatures whose relative speeds are sensitive on implementation details and on the processor used. We reported experimental comparison (on a Vaxstation 3100) of seven different algorithms for the k differences problem. The speed of the algorithms varied in a wide range, depending on the algorithm itself and on the problem parameters k, m and c. Typically a scanning speed of 20,000 – 200,000 text characters per second was obtained.

Algorithms DP, GP, and UW turned out to be uniformly slower than the remaining methods EDP, DC, MM, and ABM. Among these, no single method is always the fastest. Rather, we found the unsatisfactory situation that depending on k, m, and c, any algorithm can be the best. The algorithms also show quite unstable behavior. In certain cases the fastest of the four methods was observed to be even 30 times faster than the slowest one. Hence the choice of the proper algorithm can be very significant decision in an application where long texts have to be scanned.

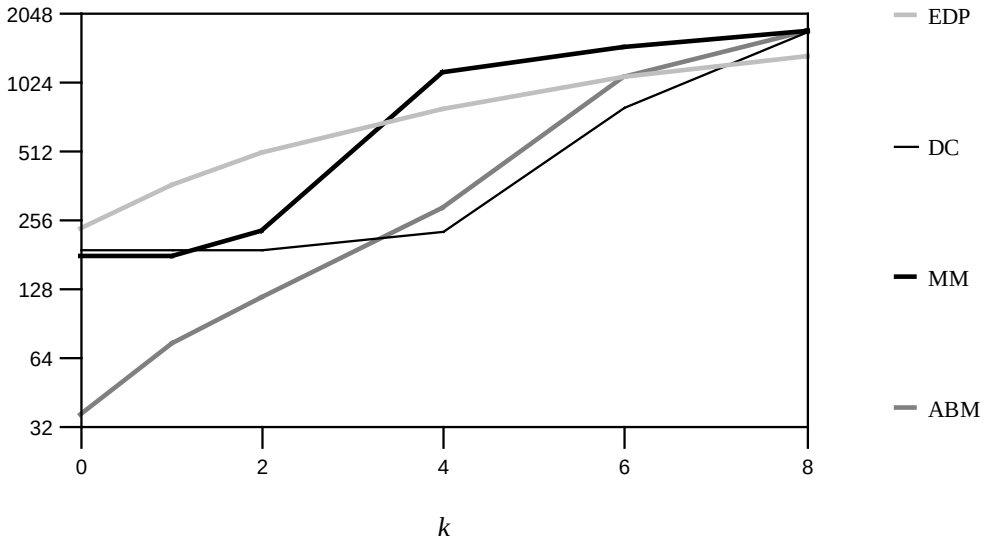

Figure 12. Execution times for c = 30 and m = 10.

Roughly summarized, our results on EDP, DC, MM, and ABM suggest that EDP is the best method for the binary alphabet and also for other alphabets when k is relatively large. Algorithm ABM is the best method for very small k ($k = 1, 2$) in which cases it achieves very high speed. In the remaining cases the choice is between algorithms MM and DC. When m is large and k not too large relative to m, algorithm MM shows the highest speed, even for the binary alphabet.

Table II. Execution times (in units of 10 milliseconds) of EDP and ABM on Sun 4/260S.

c	m	k	EDP	ABM
2	10	0	217	147
2	10	1	419	542
2	10	2	552	653
2	10	4	702	803
2	10	6	707	816
2	10	8	707	820
30	10	0	149	27
30	10	1	237	44
30	10	2	331	69
30	10	4	527	146
30	10	6	723	568
30	10	8	893	1001

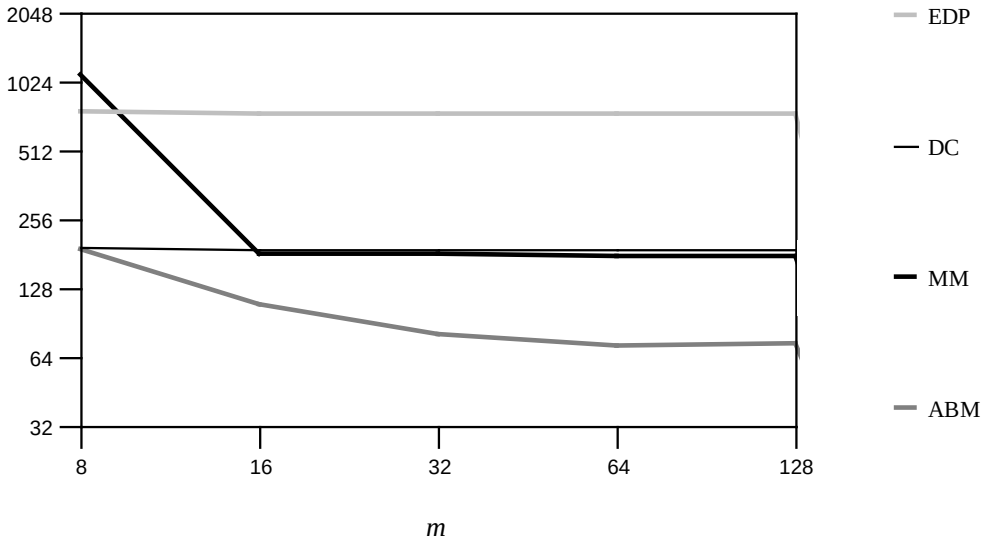

Figure 13. Execution times for c = 90 and k = 4.

ACKNOWLEDGEMENTS

The financial support of the Academy of Finland and the Alexander von Humboldt Foundation is gratefully acknowledged.

REFERENCES

1. Z. Galil and R. Giancarlo, 'Data structures and algorithms for approximate string matching', *Journal of Complexity*, **4**, 33–72 (1988).
2. P. Sellers, 'The theory and computation of evolutionary distances: Pattern recognition', *Journal of Algorithms*, **1**, 359–372 (1980).
3. E. Ukkonen, 'Finding approximate patterns in strings', *Journal of Algorithms*, **6**, 132–137 (1985).
4. G. Landau and U. Vishkin, 'Fast string matching with k differences', *Journal of Computer and System Sciences*, **37**, 63–78 (1988).
5. G. Landau and U. Vishkin, 'Fast parallel and serial approximate string matching', *Journal of Algorithms*, **10**, 157–169 (1989).
6. Z. Galil and K. Park, 'An improved algorithm for approximate string matching', *SIAM Journal on Computing*, **19**, 989–999 (1990).
7. E. Ukkonen and D. Wood, 'Approximate string matching with suffix automata', *Algorithmica*, **10**, 353–364 (1993).
8. J. Tarhio and E. Ukkonen, 'Boyer-Moore approach to approximate string matching', J. Gilbert and R. Karlson (eds.), *SWAT90, 2nd Scandinavian Workshop on Algorithm Theory*, Lecture Notes in Computer Science 447, Berlin, 1990, pp. 348–359. Springer-Verlag.
9. J. Tarhio and E. Ukkonen, 'Approximate Boyer-Moore string matching', *SIAM Journal on Computing*, **22**, 243–260 (1993).
10. E. Ukkonen, 'Approximate string matching with q-grams and maximal matches', *Theoretical Computer Science*, **92**, 191–211 (1992).
11. W. Chang and E. Lawler, 'Sublinear approximate string matching and biological applications', *Algorithmica*,

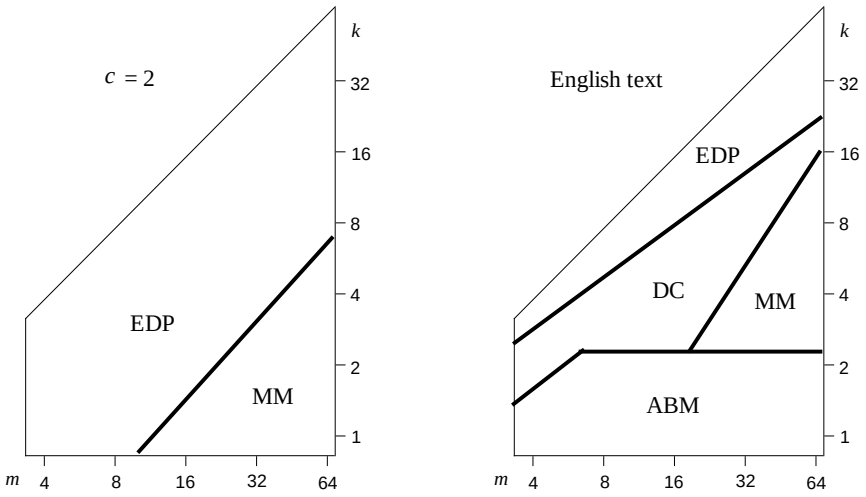

Figure 14. Rough classification.

12, 327–344 (1994).
12. T. Takaoka, 'Approximate pattern matching with samples', *Proceedings of ISAAC '94*, Lecture Notes in Computer Science 834, Berlin, 1994, pp. 234–242. Springer-Verlag.
13. E. Sutinen and J. Tarhio, 'On using q-gram locations in approximate string matching', P. Spirakis (ed.), *Proc. 3rd Annual European Symposium on Algorithms ESA '95*, Lecture Notes in Computer Science 979, Berlin, 1995, pp. 327–340. Springer.
14. W. Chang and T. Marr, 'Approximate string matching and local similarity', M. Crochemore and D. Gusfield (eds.), *Combinatorial Pattern Matching, Proceedings of 5th Annual Symposium*, Lecture Notes in Computer Science 807, Berlin, 1994, pp. 259–273. Springer-Verlag.
15. W. Chang and J. Lampe, 'Theoretical and empirical comparisons of approximate string matching algorithms', A. Apostolico et al. (ed.), *Combinatorial Pattern Matching, Proceedings of Third Annual Symposium*, Lecture Notes in Computer Science 644, Berlin, 1992, pp. 175–184. Springer-Verlag.
16. S. Wu and U. Manber, 'Fast text searching allowing errors', *Communications of ACM*, **35**, 83–91 (1992).
17. R. Baeza-Yates and C. Perleberg, 'Fast and practical approximate string matching algorithms', A. Apostolico et al. (ed.), *Combinatorial Pattern Matching, Proceedings of Third Annual Symposium*, Lecture Notes in Computer Science 644, Berlin, 1992, pp. 185–192. Springer-Verlag.
18. R. Grossi and F. Luccio, 'Simple and efficient string matching with k mismatches', *Information Processing Letters*, **33**, 113–120 (1989).
19. R. Baeza-Yates and G. Gonnet, 'A new approach to text searching', *Communications of ACM*, **35**, 74–82 (1992).
20. P. Pevzner and M. Waterman, 'Multiple filtration and approximate pattern matching', *Algorithmica*, **13**, 135–154 (1995).
21. R. Wagner and M. Fischer, 'The string-to-string correction problem', *Journal of the ACM*, **21**, 168–173 (1975).
22. E. Ukkonen, 'Algorithms for approximate string matching', *Information Control*, **64**, 100–118 (1985).
23. M. Crochemore, 'String matching with constraints', *13th Symposium on Mathematical Foundations of Computer Science*, Lecture Notes in Computer Science 324, Berlin, 1988, pp. 44–58. Springer-Verlag.
24. A. Blumer, J. Blumer, D. Haussler, A. Ehrenfeucht, M. T. Chen, and J. Seiferas, 'The smallest automaton recognizing the subwords of a text', *Theoretical Computer Science*, **40**, 31–55 (1985).
25. R. Boyer and S. Moore, 'A fast string searching algorithm', *Communcations of the ACM*, **20**, 762–772 (1977).
26. N. Horspool, 'Practical fast searching in strings', *Software – Practice and Experience*, **10**, 501–506 (1980).