An Improved Algorithm for Approximate String Matching

Zvi Galil
Kunsoo Park

CUCS-468-89

An Improved Algorithm for Approximate String Matching *

Zvi Galil [1,2]
Kunsoo Park [1]

January 1989

Abstract: Given a text string, a pattern string, and an integer k, a new algorithm for finding all occurrences of the pattern string in the text string with at most k differences is presented. Both its theoretical and practical variants improve the known algorithms.

* Work supported in part by NSF Grants CCR-86-05353 and CCR-88-14977
[1] Department of Computer Science, Columbia University, New York, NY 10027
[2] Department of Computer Science, Tel-Aviv University, Tel-Aviv, Israel

1. Introduction

The *edit distance* between a text string $x = x_1 x_2 \ldots x_n$ and a pattern string $y = y_1 y_2 \ldots y_m$ over an alphabet is the minimum number of *differences* between them. A *difference* is one of the following.

(1) A character of the pattern corresponds to a different character of the text.
(2) A character of the text corresponds to no character in the pattern.
(3) A character of the pattern corresponds to no character in the text.

An *edit operation* is an operation which corrects a difference. *Change*, *insertion*, and *deletion* are the edit operations corresponding to the three types of differences. The *edit sequence* between the pattern and the text is the sequence of edit operations for converting the pattern to the text which realizes the edit distance. Algorithms for finding the edit distance and the edit sequence were given in [9] and [8].

In this paper we are interested in a more general problem; that is to find all occurrences of the pattern in the text with at most k differences ($k \leq m \leq n$), which is called the *string matching with k differences*. Closely related is the *string matching with k mismatches* in which only the difference of type (1) is allowed. Together these two problems are called *approximate string matching*.

For the problem of string matching with k differences Landau and Vishkin provided two algorithms [4] and [5]. Their first algorithm consists of text processing of time bound $O(k^2 n)$ and preprocessing of the pattern which has a practical variant and a theoretical variant depending on the use of a suffix tree and the lowest common ancestor algorithm. Their second algorithm consists of text processing of time bound $O(kn)$ and preprocessing of both text and pattern by using a suffix tree and the lowest common ancestor algorithm, which made it less suitable for practical use. The two algorithms are incomparable for general alphabets. We present a new algorithm whose practical and theoretical variants improve both [4] and [5]. The algorithm consists of text processing of time bound $O(kn)$ and preprocessing of the pattern which has practical and theoretical variants as Landau and Vishkin's first algorithm does. The time bounds of the algorithms are summarized in Figure 1, where $\tilde{m}$ is the minimum between m and the size of the alphabet. See [2] for a survey of approximate string matching.

Algorithm	Practical	Theoretical
[4]	$O(k^2 n + m^2)$	$O(k^2 n + m \log \tilde{m})$
[5]	—	$O(kn + n \log \tilde{m})$
New	$O(kn + m^2)$	$O(kn + m \log \tilde{m})$

Figure 1. The time bounds of the algorithms

2. $O(mn)$ Algorithms

The ith character of a string x is denoted by x_i. A substring of x from the ith through the jth characters is denoted by $x_i \ldots x_j$. If the minimum number of differences between the pattern y and any substring of the text x ending at x_j is less than k, we say that y *occurs at position j of x with at most k differences*. The problem of string matching with k differences is defined as follows: Given a text x of length n, a pattern y of length m, and an integer k ($k \leq m \leq n$), find all positions

```
Algorithm MN1
for i ← 0 to m do D(i,0) ← i;
for j ← 0 to n do D(0,j) ← 0;
for j ← 1 to n do
    for i ← 1 to m do
        row ← D(i − 1,j) + 1;
        col ← D(i,j − 1) + 1;
        if y_i = x_j then diag ← D(i − 1,j − 1);
        else diag ← D(i − 1,j − 1) + 1;
        D(i,j) ← min(row,col,diag);
     end for
end for
```

Figure 2. The algorithm MN1

of x where y occurs with at most k differences. Variations of [9] and [8] solve the string matching with k differences in time $O(mn)$ as follows. See also [4, 5].

Let $D(i,j)$, $0 \leq i \leq m$ and $0 \leq j \leq n$, be the minimum number of differences between $y_1 \ldots y_i$ and any substring of x ending at x_j. The differences between $y_1 \ldots y_i$ and $x_h \ldots x_j$ for some h, $1 \leq h \leq j$ are either

(i) differences between $y_1 \ldots y_{i-1}$ and $x_h \ldots x_j$ + a difference of type (3) at y_i, or
(ii) differences between $y_1 \ldots y_i$ and $x_h \ldots x_{j-1}$ + a difference of type (2) at x_j, or
(iii) differences between $y_1 \ldots y_{i-1}$ and $x_h \ldots x_{j-1}$ + the difference between y_i and x_j.

Thus, $D(i,j)$ is determined by the three entries $D(i-1,j)$, $D(i,j-1)$, and $D(i-1,j-1)$. $D(i,0) = i$ for $0 \leq i \leq m$ because $y_1 \ldots y_i$ differs from the empty text by i differences of type (3). $D(0,j) = 0$ for $0 \leq j \leq n$ because the empty pattern occurs at any position of the text. $D(m,j) \leq k$ if and only if the pattern occurs at position j of the text with at most k differences. Figure 2 shows the dynamic programming algorithm MN1 which is a variation of Wagner and Fischer's algorithm [9]. It fills in table D column by column. Since there are $O(mn)$ entries and each entry takes constant time to be filled in, algorithm MN1 takes time $O(mn)$.

Example 1. Let x = abbdadcbc, y = adbbc, and $k = 2$. Figure 3 shows table $D(i,j)$, $0 \leq i \leq 5$ and $0 \leq j \leq 9$. The pattern occurs at positions 3, 4, 7, 8, and 9 of the text with at most 2 differences.

Lemma 1 [8]. For every $D(i,j)$, $1 \leq i \leq m$ and $1 \leq j \leq n$,
$D(i,j) = D(i-1,j-1)$ or $D(i,j) = D(i-1,j-1) + 1$.

Let *D-diagonal* d be the entries of table $D(i,j)$ such that $j - i = d$. Lemma 1 suggests a more compact way to store the information of table D. For each D-diagonal we store only the positions where the value increases. For a D-diagonal d and a difference 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 D-diagonal d end at column $C(e,d)$. Note that $C(e,d) - d$ is the row of the last entry on D-diagonal d whose value is e. Let *C-diagonal* c be the entries of table $C(e,d)$ such that $e + d = c$. The definition of $C(e,d)$ implies that the minimum number of differences between $y_1 \ldots y_{C(e,d)-d}$ and any substring of the

	D	0	1 a	2 b	3 b	4 d	5 a	6 d	7 c	8 b	9 c
0		0	0	0	0	0	0	0	0	0	0
1	a	1	0	1	1	1	0	1	1	1	1
2	d	2	1	1	2	1	1	0	1	2	2
3	b	3	2	1	1	2	2	1	1	1	2
4	b	4	3	2	1	2	3	2	2	1	2
5	c	5	4	3	2	2	3	3	2	2	1

Figure 3. Table $D(i,j)$

text ending at $x_{C(e,d)}$ is e, and $y_{C(e,d)+1-d} \neq x_{C(e,d)+1}$. $C(e,d) = m+d$ for some $e \leq k$ if and only if the pattern occurs at position $m+d$ of the text with at most k differences.

Example 2. Consider x = abbdadcbc, y = adbbc, and $k = 2$ again. Figure 4 shows table C, where columns are D-diagonals and rows are differences. For D-diagonal $d = -2, -1, 2, 3$, and 4, $C(2,d) = 5+d$. Thus, the pattern occurs at positions 3, 4, 7, 8, and 9 of the text with at most 2 differences.

	C	d −3	−2	−1	0	1	2	3	4	5	6	7
	−1			−∞	−1	0	1	2	3	4	5	6
e	0		−∞	−1	1	1	2	3	6	5	6	
	1	−∞	−1	3	3	2	4	6	9	8		
	2	−1	3	4	4	4	7	8	9			

Figure 4. Table $C(e,d)$

The computation of $C(e,d)$ starts from an entry of D-diagonal d whose value is e. In table D the entries of value $e-1$ reach column $C(e-1,d-1)$ on D-diagonal $d-1$, column $C(e-1,d)$ on D-diagonal d, and column $C(e-1,d+1)$ on D-diagonal $d+1$. Let col be the maximum of $C(e-1,d-1)+1$, $C(e-1,d)+1$, and $C(e-1,d+1)$. $D(col-d,col)$ gets value e from one of the last entries of value $e-1$ on D-diagonals $d-1$, d, and $d+1$. The entries of value e on D-diagonal d continue until there is a mismatch between the pattern and the text on the D-diagonal. $C(e,d)$ is the column of the last entry on D-diagonal d whose value is e. For D-diagonal $d \geq 0$, the initial value of the D-diagonal is 0 at column d (i.e., $D(0,d) = 0$), so we assign $d-1$ to $C(-1,d)$ which indicates that imaginary entries of value -1 end at column $d-1$. Since the initial value of D-diagonal d, $-(k+1) \leq d \leq -1$, is $|d|$ at column 0, we assign -1 to $C(|d|-1,d)$. We also assign $-\infty$ to $C(|d|-2,d)$, $-(k+1) \leq d \leq -1$, so that they are properly initialized.

There are three types of D-diagonals with respect to table C.

(i) For $-k \leq d \leq -1$, we compute $d+k+1$ entries $C(e,d)$, $|d| \leq e \leq k$, because D-diagonal d starts with value $|d|$.

(ii) For $0 \leq d \leq n-m$, we compute $k+1$ entries $C(e,d)$, $0 \leq e \leq k$.

```
Algorithm MN2
// initialization //
for d ← 0 to n − m + k + 1 do C(−1, d) ← d − 1;
for d ← −(k + 1) to −1 do
    C(|d| − 1, d) ← −1;
    C(|d| − 2, d) ← −∞;
end for
for c ← 0 to n − m + k do
    for e ← 0 to k do
        d ← c − e;
        col ← max(C(e − 1, d − 1) + 1, C(e − 1, d) + 1, C(e − 1, d + 1));
        while col < n and col − d < m and y_{col+1−d} = x_{col+1} do
            col ← col + 1;
        end while
        C(e, d) ← min(col, m + d);
    end for
end for
```

Figure 5. The algorithm MN2

(iii) For $n-m+1 \leq d \leq n-m+k$, we compute $(n-m+k)-d+1$ entries $C(e,d), 0 \leq e \leq (n-m+k)-d$ because D-diagonal $n-m$ is the last D-diagonal for which we want to compute C, and D-diagonal d may affect the values of D-diagonal $n-m$ by the difference of type (3).

Thus the shape of table C is a parallelogram. Figure 5 shows the dynamic programming algorithm MN2 which is a variation of Ukkonen's algorithm [8]. It proceeds C-diagonal by C-diagonal. $col+1$ and $col+1-d$ are the postions of the text and the pattern where the characters are compared.

Lemma 2. The characters of the text which are compared with the pattern in the computation of C-diagonal c are at most $x_{c+1}, \ldots, x_{c+m}$ ($x_{c+1}, \ldots, x_n$ if $c+m > n$).

Proof. We show that in order to compute $C(e, c-e)$ for $0 \leq e \leq k$, at most $x_{c+1}, \ldots, x_{m+c-e}$ are compared with the pattern. To compute $C(0,c)$, we start with the comparison of y_1 and x_{c+1}, and we may continue up to the comparison of y_m and x_{c+m}, which is the last one on D-diagonal c. Thus we compare at most $x_{c+1}, \ldots, x_{c+m}$ with the pattern for $C(0,c)$. To compute $C(e, c-e)$ for $0 < e \leq k$, the first position of the text to be compared is greater than or equal to $c+1$: the computation of $C(0, c-e)$ starts at text position $c-e+1$, and there is at least one entry of each value e', $0 \leq e' < e$, on D-diagonal $c-e$. The entries of value e on D-diagonal $c-e$ may continue to $D(m, m+c-e)$, the last entry on D-diagonal $c-e$. Thus, at most $x_{c+1}, \ldots, x_{m+c-e}$ are compared for $C(e, c-e)$. If any position of the text is greater than n, the last position to be considered should be n. •

Lemma 3. During the computation of C-diagonal c,

1. the positions of the characters of the text which are actually compared with the pattern are nondecreasing, and
2. the repetitions of text positions occur at most k times.

Proof. Let j be the text position where a mismatch occurred in the computation of $C(e, c-e)$ for $0 \leq e < k$; i.e., $C(e, c-e) = j-1$. We show that the first position of the text to be considered

for $C(e+1, c-e-1)$ is at least j. At the beginning of the computation of $C(e+1, c-e-1)$, $col \geq C(e, c-e)$. The first position of the text $col+1$ satisfies the following:

$$col + 1 \geq C(e, c-e) + 1 = j.$$

Since the repetition of a text position occurs only at the first comparison for $C(e, c-e)$, $1 \leq e \leq k$, there are at most k repetitions. •

By lemma 2 and lemma 3 the computation of each C-diagonal takes time $O(m)$. Since there are $n - m + k + 1$ C-diagonals, algorithm MN2 takes time $O(mn)$.

3. The New Algorithm

The algorithm consists of preprocessing of the pattern followed by processing of the text. In the preprocessing we build an upper triangular table *Prefix*(i, j), $1 \leq i < j \leq m$, where *Prefix*(i, j) is the length of the longest common prefix of $y_i \ldots y_m$ and $y_j \ldots y_m$. This table is used for the comparison of two substrings of the pattern during the text processing. The details of the preprocessing will be discussed in the next section.

The text processing is based on the second algorithm in the previous section. It consists of $n - m + k + 1$ iterations, one for each C-diagonal, as algorithm MN2 does. Whereas algorithm MN2 relies only on direct comparisons of the text with the pattern, the new algorithm uses both direct comparisons and lookups of the *Prefix* table. If a substring of the text had matches with a substring of the pattern, the algorithm looks up the *Prefix* table for the substring of the text. Otherwise, it directly compares the text with the pattern. For the matched part of the text the algorithm compares two substrings of the pattern instead of comparing a substring of the pattern with a substring of the text. This technique which first appeared in the Knuth–Morris–Pratt algorithm [3] was also used in [4] and [7].

A *reference triple* (u, v, w) consists of a start position u, an end position v, and a D-diagonal w such that substring $x_u \ldots x_v$ of the text matches substring $y_{u-w} \ldots y_{v-w}$ of the pattern and $x_{v+1} \neq y_{v+1-w}$. Note that w is the D-diagonal where the match occurred. We call $y_{u-w} \ldots y_{v-w}$ the *reference* of $x_u \ldots x_v$. If $u > v$ in a triple (u, v, w), the triple is called *null*, and it indicates that $[u, v]$ is an empty interval and $x_{v+1} \neq y_{v+1-w}$. The idea of triples which are equivalent to reference triples appeared in [4].

At iteration c we compute C-diagonal c which is $C(e, c-e)$, $0 \leq e \leq k$. Let q be the text position such that x_{q+1} is the rightmost character of the text which was compared with the pattern before iteration c (i.e., x_{q+1} had a mismatch). Suppose that from previous iterations we have $k+1$ reference triples (u_0, v_0, w_0), (u_1, v_1, w_1), ... , (u_k, v_k, w_k) such that the set of intervals $[u_0, v_0]$, $[u_1, v_1]$, ... , $[u_k, v_k]$ is a partition of interval $[c, q]$ with a possible hole between v_e and u_{e+1} for $0 \leq e < k$ (i.e., either $u_{e+1} = v_e + 1$ or $u_{e+1} = v_e + 2$). Initially, $q = 0$ and all triples are $(0, 0, 0)$.

Let t be the current text position ($col + 1$ in Figures 5 and 6) in the computation of $C(e, c-e)$ for $0 \leq e \leq k$. To compute $C(e, c-e)$, we look for the first mismatch $x_j \neq y_{j-(c-e)}$ for $j \geq t$. Then $C(e, c-e)$ will be $j - 1$. For notational convenience, let $d = c - e$ hereafter. If $t > q$, we have no reference triples for x_t. So we compare the text with the pattern until there is a mismatch. While $t \leq q$, we compare the pattern with references unless t is the position of a hole, in which case x_t is directly compared with the pattern. If t is within the interval of a reference triple (u_r, v_r, w_r) for some $0 \leq r \leq k$, we look up the *Prefix* table. The current pattern position p is $t - d$, and the

```
Algorithm KN
initializations
for c ← 0 to n − m + k do
    r ← 0;
    for e ← 0 to k do
        d ← c − e;
        col ← max(C(e − 1, d − 1) + 1, C(e − 1, d) + 1, C(e − 1, d + 1));
        found ← false;
        while not found do
            if within(col + 1, k, r) then
                f ← v_r − col;
                g ← Prefix(col + 1 − d, col + 1 − w_r);
                if f = g then
                    col ← col + f;
                else
                    col ← col + min(f, g);
                    found ← true;
                end if
            else
                if col < n and col − d < m and y_{col+1−d} = x_{col+1} then
                    col ← col + 1;
                else
                    found ← true;
                end if
            end if
        end while
        C(e, d) ← min(col, m + d);
        update reference triple (u_e, v_e, w_e);
    end for
end for
```

Figure 6. The algorithm KN

reference position corresponding to t is $t - w_r$. We look at $Prefix(p, t - w_r)$. Let f be $v_r - t + 1$, the length of the reference from $t - w_r$ to $v_r - w_r$. Let g be $Prefix(p, t - w_r)$, the length of the longest common prefix of $y_p \ldots y_m$ and $y_{t-w_r} \ldots y_m$. There are three cases:

(i) $f < g$: text $x_t \ldots x_{t+f-1}$ matches pattern $y_p \ldots y_{p+f-1}$, but $x_{t+f} \neq y_{p+f}$ because $x_{t+f} \neq y_{t+f-w_r}$ by the definition of reference triples and $y_{t+f-w_r} = y_{p+f}$ since $f < g$.

(ii) $f = g$: text $x_t \ldots x_{t+f-1}$ matches pattern $y_p \ldots y_{p+f-1}$, and x_{t+f} may or may not match y_{p+f} because $x_{t+f} \neq y_{t+f-w_r}$ and $y_{t+f-w_r} \neq y_{p+f}$.

(iii) $f > g$: text $x_t \ldots x_{t+g-1}$ matches pattern $y_p \ldots y_{p+g-1}$, but $x_{t+g} \neq y_{p+g}$ because $x_{t+g} = y_{t+g-w_r}$ and $y_{t+g-w_r} \neq y_{p+g}$.

In cases (i) and (iii) we have found j which is $t + \min(f, g)$. In case (ii) we continue at position $t + f$.

After iteration c we update reference triples for the next iteration. Let s_e, $0 \leq e \leq k$, be the

first position of the text which was considered for $C(e,d)$. $C(e,d)$ itself is the last position where a series of matches (possibly null) ended. Namely, $x_{s_e},\ldots,x_{C(e,d)}$ had matches with the pattern if $s_e \le C(e,d)$. Therefore, triples $(s_e, C(e,d), d)$, $0 \le e \le k$, are reference triples which came from the computation of C-diagonal c. We combine the old reference triple (u_e, v_e, w_e) and the reference triple $(s_e, C(e,d), d)$ to obtain the new reference triple (u'_e, v'_e, w'_e) for each $0 \le e \le k$. Two triples (u_e, v_e, w_e) and $(s_e, C(e,d), d)$ compete for (u'_e, v'_e, w'_e), and the one with larger end position (i.e., v_e vs. $C(e,d)$) wins. New v'_e and w'_e are those of the winner. It follows by induction on the iteration number that after the update, v'_e is the maximum of $C(e, i-e)$, $0 \le i \le c$. New u'_e is $c+1$ if $e = 0$. If $e > 0$, there are two cases for u'_e. In each case we show that (u'_e, v'_e, w'_e) is a reference triple and u'_e is either $v'_{e-1} + 1$ or $v'_{e-1} + 2$.

(i) If (u_e, v_e, w_e) is the winner, u'_e is the maximum of u_e and $v'_{e-1} + 1$.
  (a) (u'_e, v'_e, w'_e) is a reference triple because (u_e, v_e, w_e) is a reference triple and $u'_e \ge u_e$.
  (b) u_e is either $v_{e-1} + 1$ or $v_{e-1} + 2$ from previous iterations, and $v_{e-1} \le v'_{e-1}$ by the previous competition. Thus, $u_e > v'_{e-1} + 1$ only when $u_e = v_{e-1} + 2$ and $v_{e-1} = v'_{e-1}$, in which case $u_e = v'_{e-1} + 2$.

(ii) If $(s_e, C(e,d), d)$ is the winner, u'_e is the maximum of s_e and $v'_{e-1} + 1$.
  (a) (u'_e, v'_e, w'_e) is a reference triple because $(s_e, C(e,d), d)$ is a reference triple and $u'_e \ge s_e$.
  (b) s_e is the maximum of $C(e-1, d+1)+1$, $C(e-1,d)+2$, and $C(e-1,d-1)+2$. v'_{e-1} is the maximum of $C(e-1,d+1)$, $C(e-1,d)$, $C(e-1,d-1)$, $\ldots$, $C(e-1,-(e-1))$. Thus, $s_e > v'_{e-1}+1$ when $s_e = C(e-1,d)+2$ and $v'_{e-1} = C(e-1,d)$ or when $s_e = C(e-1,d-1)+2$ and $v'_{e-1} = C(e-1,d-1)$. In both cases, $s_e = v'_{e-1} + 2$.

Therefore, (u'_0, v'_0, w'_0), (u'_1, v'_1, w'_1), $\ldots$, (u'_k, v'_k, w'_k) are reference triples, and the set of intervals $[u'_0, v'_0]$, $[u'_1, v'_1]$, $\ldots$, $[u'_k, v'_k]$ is a partition of interval $[c+1, q']$ with a possible hole between v'_e and u'_{e+1} for $0 \le e < k$, where q' is the largest end position of the triples (q' may not be v'_k if triple (u'_k, v'_k, w'_k) is null). Instead of updating all reference triples at the end of iteration c, we can update the reference triple (u_e, v_e, w_e) after the computation of $C(e,d)$. If the old reference triple (u_e, v_e, w_e) is the winner, triple $(s_e, C(e,d), d)$ is simply discarded. Otherwise, the text position to be considered next is greater than $C(e,d)$ by lemma 3.1, so we can update (u_e, v_e, w_e) securely.

Example 3. Consider x = abbdadcbc, y = adbbc, and $k = 2$. Figure 7 shows the reference triples at the beginning of iteration c, $1 \le c \le 6$. The triples marked with * are null triples.

	Reference triples		
c	0	1	2
1	$(1,1,0)$	$(2,3,-1)$	$(4,3,-2)^*$
2	$(2,1,0)^*$	$(2,3,-1)$	$(5,4,-1)^*$
3	$(3,2,2)^*$	$(3,3,-1)$	$(5,4,-1)^*$
4	$(4,3,3)^*$	$(4,4,2)$	$(5,4,-1)^*$
5	$(5,6,4)$	$(7,6,3)^*$	$(7,7,2)$
6	$(6,6,4)$	$(8,9,4)$	$(10,8,3)^*$

Figure 7. Reference triples

The algorithm KN in Figure 6 shows the text processing. Procedure $within(t,k,r)$ tests if text position t is within an interval of reference triples, in which case r is the index of the reference triple

```
procedure within(t, k, r)
    while r ≤ k and t > v_r do r ← r + 1;
    if r > k then return(false);
    else
        if t ≥ u_r then return(true);
        else return(false);
    end if
end
```

Figure 8. The procedure $within(t, k, r)$

within whose interval text position t is. At the beginning of iteration c, r is 0. By lemma 3.1, r never decreases during iteration c. If $t > v_i$ for all $0 \leq i \leq k$, obviously $t > q$; $within(t, k, r)$ returns **false**. If $t \leq q$, it increases r by 1 until $t \leq v_r$. If $u_r \leq t$, t is in interval $[u_r, v_r]$; it returns **true**. If $t < u_r$, t is the position of a hole; it returns **false**. Figure 8 shows the procedure $within(t, k, r)$. The text position q is implicitly maintained by the reference triples.

At iteration c the number of repetitions of the **while** loop in algorithm KN is the number of direct comparisons and lookups of the *Prefix* table. Direct comparisons are counted in two ways.

(i) If $t > q + 1$ (i.e., x_t is a new character), the comparison is charged to text position t.
(ii) If $t \leq q + 1$ (i.e., x_t was compared before iteration c), the comparison is charged to C-diagonal c.

When $t > q + 1$, there can be at most k repetitions of text position t during iteration c by lemma 3.2. At the next iteration the text position belongs to (ii). Thus, there are $O(kn)$ comparisons for the whole text processing by rule (i). In interval $[c, q + 1]$ there are at most k holes from the reference triples and another hole at $q + 1$. A direct comparison at a hole either increases e (when a mismatch occurs) or causes passing the hole (when a match occurs). Hence, $O(k)$ comparisons are charged to C-diagonal c by rule (ii). Table lookups are charged to C-diagonal c. A lookup of the table increases either e (when $f \neq g$) or r (when $f = g$); $O(k)$ lookups are charged to C-diagonal c. Since there are $n - m + k + 1$ C-diagonals, the total time of the text processing is $O(kn)$.

The text processing maintains table C and $k + 1$ reference triples. To find edit distances (i.e., string matching with k differences) we keep only two previous C-diagonals. Thus, the space required for the text processing is $O(k)$. If we want to find both edit distances and edit sequences, we need to keep k C-diagonals [8], which leads to $O(k^2)$ space.

4. The Preprocessing

In the preprocessing of pattern y we compute upper triangular table $Prefix(i, j)$, $1 \leq i < j \leq m$, where $Prefix(i, j)$ is the length of the longest common prefix of $y_i \ldots y_m$ and $y_j \ldots y_m$. The procedure in Figure 9 computes $Prefix(i, j)$ diagonal by diagonal. For each diagonal d it starts to compare y_1 with y_{1+d} and proceeds on the diagonal until there is a mismatch $y_c \neq y_{c+d}$. Then $Prefix(1, 1 + d) = c - 1$, $Prefix(2, 2 + d) = c - 2, \ldots,$ $Prefix(c, c + d) = 0$. It resumes the comparison with y_{c+1} and y_{c+1+d}, and repeats until it reaches the end of the pattern. If the procedure makes c comparisons in the inner **while** loop, it fills in c entries of the *Prefix* table. Since there are $m(m - 1)/2$ entries, the preprocessing takes time and space $O(m^2)$. An alternate computation of

```
for d ← 1 to m − 1 do
    i ← 0;
    while i + d < m do
        c ← 1;
        while i + c + d ≤ m and y_{i+c} = y_{i+c+d} do c ← c + 1;
        for j ← 1 to c do Prefix(i + j, i + j + d) ← c − j;
        i ← i + c;
    end while
end for
```

Figure 9. The preprocessing

the *Prefix* table which also takes time and space $O(m^2)$ appears in [4]. Taking into account both preprocessing and text processing, our algorithm takes time $O(kn + m^2)$ and space $O(m^2)$.

Using a suffix tree and the lowest common ancestor algorithm, the time bound of the preprocessing can be reduced to $O(m \log m)$ for general alphabets or to $O(m)$ for alphabets whose size is fixed, and the space bound is reduced to $O(m)$ [2, 4, 5]. Since, however, the constant hidden in the suffix tree and the lowest common ancestor algorithm is quite large, it is mostly of theoretical interest. In this case our algorithm takes time $O(kn + m \log \tilde{m})$ and space $O(m)$.

5. Conclusion

We have presented a new algorithm for the string matching with k differences which improves the known algorithms. It is interesting that the time and space bounds of the algorithm are the same as those of Galil and Giancarlo's algorithm [1] for the string matching with k mismatches. In addition to the bounds, they are similar in that both algorithms use the *Prefix* table and maintain $k + 1$ references (by a D-diagonal and mismatched text positions in [1], and by reference triples in ours). They are different in that [1] finds start positions of occurrences of the pattern in the text while our algorithm finds end positions of the occurrences, and [1] builds references from one D-diagonal while our algorithm builds them from at most $k + 1$ D-diagonals.

An additional difference was considered in [6] and [8].

(4) Two adjacent characters ab of the pattern correspond to the transposed characters ba of the text.

Transposition is the edit operation which corrects the difference of type (4). Our algorithm can be extended to include the difference of type (4) with some modifications. Since a mismatch $x_i \neq y_{i-d}$ may turn out to be a difference of type (4) $x_i x_{i+1} = y_{i+1-d} y_{i-d}$,

$$col \leftarrow \max(C(e-1, d-1) + 1, C(e-1, d) + 1, C(e-1, d+1));$$

should be replaced by

```
i ← C(e − 1, d) + 1;
if y_{i+1−d} y_{i−d} = x_i x_{i+1} then i ← i + 1;
col ← max(i, C(e − 1, d − 1) + 1, C(e − 1, d + 1));
```

Now the first position s_e of the text which is considered for $C(e, d)$ can be $v'_{e-1} + 3$ because of transposed characters. Thus there are at most two holes between the intervals of reference triples.

References

[1] Galil, Z., and Giancarlo, R. Improved string matching with k mismatches. *SIGACT News* **17** (1986), 52–54.

[2] Galil, Z., and Giancarlo, R. Data structures and algorithms for approximate string matching. *Journal of Complexity* **4** (1988), 33–72.

[3] Knuth, D. E., Morris, J. H., and Pratt, V. R. Fast pattern matching in strings. *SIAM J. Comput.* **6** (1977), 323–350.

[4] Landau, G. M., and Vishkin, U. Fast string matching with k differences. *J. Comput. System Sci.* **37** (1988), 63–78.

[5] Landau, G. M., and Vishkin, U. Fast parallel and serial approximate string matching. *Journal of Algorithms* **10** (1989).

[6] Lowrance, R., and Wagner, R. A. An extension of the string-to-string correction problem. *J. Assoc. Comput. Mach.* **22** (1975), 177–183.

[7] Main, M. G., and Lorentz, R. J. An $O(n \log n)$ algorithm for finding all repetitions in a string. *Journal of Algorithms* **5** (1984), 422–432.

[8] Ukkonen, E. Algorithms for approximate string matching. *Information and Control* **64** (1985), 100–118.

[9] Wagner, R. A., and Fischer, M. J. The string-to-string correction problem. *J. Assoc. Comput. Mach.* **21** (1974), 168–173.