APPROXIMATE BOYER-MOORE STRING MATCHING

JORMA TARHIO AND ESKO UKKONEN

University of Helsinki, Department of Computer Science
Teollisuuskatu 23, SF-00510 Helsinki, Finland

Draft

Abstract. The Boyer-Moore idea applied in exact string matching is generalized to approximate string matching. Two versions of the problem are considered. The k mismatches problem is to find all approximate occurrences of a pattern string (length m) in a text string (length n) with at most k mismatches. Our generalized Boyer-Moore algorithm is shown (under a mild independence assumption) to solve the problem in expected time $O(kn(\frac{1}{m-k} + \frac{k}{c}))$ where c is the size of the alphabet. A related algorithm is developed for the k differences problem where the task is to find all approximate occurrences of a pattern in a text with $\leq k$ differences (insertions, deletions, changes). Experimental evaluation of the algorithms is reported showing that the new algorithms are often significantly faster than the old ones. Both algorithms are functionally equivalent with the Horspool version of the Boyer-Moore algorithm when $k = 0$.

Key words: String matching, edit distance, Boyer-Moore algorithm, k mismatches problem, k differences problem

AMS (MOS) subject classifications: 68C05, 68C25, 68H05

Abbreviated title: Approximate Boyer-Moore Matching

1. Introduction

The fastest known exact string matching algorithms are based on the Boyer-Moore idea [BoM77, KMP77]. Such algorithms are “sublinear” on the average in the sense that it is not necessary to check every symbol in the text. The larger is the alphabet and the longer is the pattern, the faster the algorithm works. In this paper we generalize this idea to approximate string matching. Again the approach leads to algorithms that are significantly faster than the previous solutions of the problem.

We consider two important versions of the approximate string matching problem. In both, we are given two strings, the *text* $T = t_1 t_2 \ldots t_n$ and the *pattern* $P = p_1 p_2 \ldots p_m$ in some alphabet Σ, and an integer k. In the first variant, called the *k mismatches problem*, the task is to find all occurrences of P in T with at most k mismatches, that is, all j such that $p_i = t_{j-m+i}$ for $i = 1, \ldots, m$ except for at most k indexes i.

In the second variant, called the *k differences problem*, the task is to find (the end points of) all substrings P' of T with the edit distance at most k from P. The edit distance means the minimum number of editing operations (the differences) needed to convert P' to P. An editing operation is either an insertion, a deletion or a change of a character. The k mismatches problem is a special case with the change as the only editing operation.

There are several algorithms proposed for these two problems, see e.g. the survey [GaG88]. Both can be solved in time $O(mn)$ by dynamic programming [Sel80, Ukk85b]. A very simple improvement giving $O(kn)$ expected time solution for random strings is described in [Ukk85b]. Later, Landau and Vishkin [LaV88, LaV89], Galil and Park [GaP89], Ukkonen and Wood [UkW90] have given different algorithms that consist of preprocessing the pattern in time $O(m^2)$ (or $O(m)$) and scanning the text in worst-case time $O(kn)$. For the k differences problem, $O(kn)$ is the best bound currently known if the preprocessing is allowed to be at most $O(m^2)$. For the k mismatches problem Kosaraju [Kos88] gives an $O(n\sqrt{m}\, polylog(m))$ algorithm. Also see [GaG86, GrL89].

We develop a new approximate string matching algorithm of Boyer-Moore type for the k mismatches problem and show, under a mild independence assumption, that it processes a random text in expected time $O(kn(\frac{1}{m-k} + \frac{k}{c}))$ where c denotes the size of the alphabet. A related but different method is (independently) developed and analyzed in [Bae89a]. We also give an algorithm for the k differences problem and show in a special case that its expected processing time for a random text is $O(\frac{c}{c-2k}\, kn\, (\frac{k}{c+2k^2} + \frac{1}{m}))$. The preprocessing of the pattern needs time $O(m + kc)$ and $O((k + c)m)$, respectively. We have also performed extensive experimental comparison of the new methods with the old ones showing that Boyer-Moore algorithms are significantly faster, for large m and c in particular.

Our algorithms can be considered as generalizations of the Boyer-Moore algorithm for exact string matching, because they are functionally identical with the Horspool version [Hor80] of the Boyer-Moore algorithm when $k = 0$. The algorithm of [Bae89a] generalizes the original Boyer-Moore idea for the k mismatches problem.

All these algorithms are “sublinear” in the sense that it is not necessary to examine every text symbol. Another approximate string matching method of this type (based on totally different ideas) has recently been given in [ChL90].

The paper is organized as follows. We first consider the k mismatches problem for which we give and analyze the Boyer-Moore solution in Section 2. Section 3 develops an extension to the k differences problem and outlines an analysis. Section 4 reports our experiments.

2. The *k* mismatches problem

2.1. Boyer-Moore-Horspool algorithm

The characteristic feature of the Boyer-Moore algorithm [BoM77] 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 in the text currently below it. Between alignments, the pattern is shifted from left to right along the text.

In the original algorithm the shift is computed using two heuristics: the match heuristic and the occurrence heuristic. The *match* heuristic implements the requirement that after a shift, the pattern has to match all the text characters that were found to match at the previous alignment. The *occurrence* heuristic implements the requirement that we must align the rightmost character in the text that caused the mismatch with the rightmost character of the pattern that matches it. After each mismatch, the algorithm chooses the larger shift given by the two heuristics.

As the patterns are not periodic on the average, the match heuristic is not very useful. A simplified version of the method can be obtained by using the occurrence heuristic only. Then we may observe that it is not necessary to base the shift on the text symbol that caused the mismatch. Any other text character below the current pattern position will do as well. Then the natural choice is the text character corresponding to the rightmost character of the pattern as it potentially leads to the longest shifts. This simplification was noted by Horspool [Hor80]. We call this method the Boyer-Moore-Horspool or the BMH algorithm.

The BMH algorithm has a simple code and is in practice better than the original Boyer-Moore algorithm. In the preprocessing phase the algorithm computes from the pattern $P = p_1p_2...p_m$ the shift table d, defined for each symbol a in alphabet Σ as

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

For a text symbol a below p_m, the table d shifts the pattern right until the rightmost a in $p_1...p_{m-1}$ becomes above the a in the text. Table d can be computed in time $O(m + c)$ where $c = |\Sigma|$, by the following algorithm:

Algorithm 1. BMH-preprocessing.
for a in Σ **do** $d[a] := m$;
for $i := 1, ..., m-1$ **do** $d[p_i] := m - i$

The total BMH method [Hor80] including the scanning of the text $T = t_1t_2...t_n$ is given below:

Algorithm 2. The BMH method for exact string matching.

```
call Algorithm 1;
j := m;                                   {pattern ends at text position j}
while j ≤ n do begin
    h := j; i := m;                       {h scans the text, i the pattern}
    while i > 0 and t_h = p_i do begin
        i := i – 1; h := h – 1 end;       {proceed to the left}
    if i = 0 then report match at position j;
    j := j + d[t_j] end                   {shift to the right}
```

2.2. Generalized BMH algorithm

The generalization of the BMH algorithm for the k mismatches problem will be very natural: for $k = 0$ the generalized algorithm is exactly as Algorithm 2. Recall that the k mismatches problem asks for finding all occurrences of P in T such that in at most k positions of P, T and P have different characters.

We have to generalize both the right-to-left scanning of the pattern and the computation of the shift. The former is very simple; we just scan the pattern to the left until we have found $k + 1$ mismatches (unsuccessful search) or the pattern ends (successful search).

To understand the generalized shift it may be helpful to look at the k mismatches problem in a tabular form. Let M be a $m \times n$ table such that for $1 \le i \le m$, $1 \le j \le n$,

$$M[i, j] = \begin{cases} 0, & \text{if } p_i = t_j \\ 1, & \text{if } p_i \ne t_j \end{cases}$$

There is an exact match ending at position r of T if $M[i, r - \mathrm{m} + i] = 0$ for $i = 1, \ldots, m$, that is there is a whole diagonal of 0's in M ending at $M[m, r]$. Similarly, there is an approximate match with $\le k$ mismatches if the diagonal contains at most k 1's. This implies that any successive $k + 1$ entries of such a diagonal have to contain at least one 0.

Assume then that the pattern is ending at text position j and we have to compute the next shift. We consider the last $k + 1$ text characters below the pattern, the characters $t_{j-k}, t_{j-k+1}, \ldots, t_j$. Then, suggested by the above observation, we glide the pattern to the right until there is at least one match in $t_{j-k}, t_{j-k+1}, \ldots, t_j$. The maximum shift is $m - k$. Clearly this is a correct heuristic: A smaller shift would give an unsuccessful alignment because there are at least $k + 1$ mismatches, and a shift larger than $m - k$ would skip over a potential match.

Let $d(t_{j-k}, t_{j-k+1}, \ldots, t_j)$ denote the length of the shift. The values of $d(t_{j-k}, \ldots, t_j)$ could be precomputed and tabulated. This would lead to quite heavy preprocessing of at least time $\Theta(c^k)$. Instead, we apply a simpler preprocessing that makes it possible to compute the shift on-the-fly with small overhead while scanning.

In terms of M the shifting means finding the first diagonal above the current diagonal such that the new diagonal has at least one 0 for t_{j-k}, $t_{j-k+1}, \ldots, t_j$.

$$
\begin{array}{ccc}
 & & T \\
 & & \begin{array}{cccccccccccc} a & b & a & a & c & b & b & a & b & b & b & a \end{array} \\
P & \begin{array}{c} a \\ b \\ b \\ b \end{array} & \boxed{\begin{array}{cccccccccccc}
0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 \\
1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\
1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\
1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1
\end{array}} \\
 & & \longrightarrow
\end{array}
$$

Figure 1. Determining of shift ($k = 1$).

For example, consider table M in Fig. 1, where we assume that $k = 1$. We may shift from the diagonal of $M[1, 1]$ directly to the diagonal of $M[1, 3]$, as this diagonal contains the first 0 for characters $t_3 = a$, $t_4 = a$. Hence $d(a, a) = 2$

for the pattern *abbb*. Also note that t_4 alone would give a shift of 3 and t_3 a shift of 2, and $d(t_3, t_4)$ is the minimum over these component shifts.

In general, we compute $d(t_{j-k}, ..., t_j)$ as the minimum of the component shifts for each $t_{j-k}, ..., t_j$. The component shift for t_h depends both on the character t_h itself and on its position below the pattern. Possible positions are $m - k, m - k + 1, ..., m$. Hence we need a $(k + 1) \times c$ table d_k defined for each $i = m - k, ..., m$, and for each a in Σ, as

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

Here the values greater than $m - k$ are not actually relevant. Table d_k is presented in this form, because the same table is used in the algorithm solving the k differences problem.

Table d_k can be computed in time $O((m + c)k)$ by a straightforward generalization of the BMH-preprocessing which scans $k + 1$ times over P and each scanning creates a new row of d_k.

A more efficient method needs only one scan, from right to left, over P. For each symbol p_i encountered, the corresponding updates are made to d_k. To keep track of the updates already made, we use a table $ready[a]$, a in Σ, such that $ready[a] = j$ if $d_k[i, a]$ already has its final value for $i = m, m - 1, ..., j$. Initially, $ready[a] = m + 1$ for all a, and $d_k[i, a] = m$ for all i, a. The algorithm is as follows:

Algorithm 3. Computation of table d_k.

for a in Σ **do** $ready[a] := m + 1$;
for a in Σ **do**
    for $i := m$ **downto** $m - k$ **do**
        $d_k[i, a] := m$;
for $i := m - 1$ **downto** 1 **do begin**
    for $j := ready[p_i] - 1$ **downto** $max(i, m - k)$ **do**
        $d_k[j, p_i] := j - i$;
    $ready[p_i] := max(i, m - k)$ **end**

The initializations in steps 1–4 take time $O(kc)$. Steps 5–8 scan over P in time $O(m)$ plus the time of the updates of d_k in step 7. This takes time $O(kc)$ as each $d_k[j, p_i]$ is updated at most once. Hence Algorithm 3 runs in time $O(m + kc)$.

We have now the following total method for the k mismatches problem:

Algorithm 4. Approximate string matching with k mismatches.

1. compute table d_k from P with Algorithm 3;
2. $j := m$; {pattern ends at text position j}
3. **while** $j \leq n + k$ **do begin**
4. $h := j$; $i := m$; $neq := 0$; {h scans the text, i the pattern}
5. $d := m - k$; {initial value of the shift}
6.   **while** $i > 0$ **and** $neq \leq k$ **do begin**
7.   **if** $i \geq m - k$ **then** $d := min(d, d_k[i, t_h])$; {minimize over the component shifts}
8.   **if** $t_h \neq p_i$ **then** $neq := neq + 1$;
9. $i := i - 1$; $h := h - 1$ **end**; {proceed to the left}
10.   **if** $neq \leq k$ **then** report match at position j;
11. $j := j + d$ **end** {shift to the right}

2.3. Analysis

First recall that the preprocessing of P by Algorithm 3 takes time $O(m + kc)$ and space $O(kc)$. The scanning of T by Algorithm 4 obviously needs $O(mn)$ time in the worst case. The bound is strict for example for $T = a^n$, $P = a^m$.

Next we analyze the scanning time in the average case. The analysis will be done under the random string assumption which says that individual characters in P and T are chosen independently and uniformly from Σ. The time requirement is proportional to the number of the text-pattern comparisons in step 8 of Algorithm 4. Let $C_{loc}(P)$ be a random variable denoting, for some fixed c and k, the number of such comparisons for some alignment of pattern P between two successive shifts, and let $\overline{C}_{loc}(P)$ be its expected value.

Lemma 1. $\overline{C}_{loc}(P) < \left(\frac{c}{c-1} + 1\right)(k+1)$.

Proof. The distribution of $C_{loc}(P) - (k+1)$ converges to the negative binomial distribution (the Pascal distribution) with parameters $(k+1, 1-\frac{1}{c})$ when $m \rightarrow \infty$, because $C_{loc}(P) - (k+1)$ is the number of matches until we find the $k+1$st mismatch; the probability of the mismatch is $1-\frac{1}{c}$. As the expected value of $C_{loc}(P)$ increases with m, the expected value $\frac{k+1}{c-1}$ of this negative binomial distribution (see e.g. [Fel65]) would be an upper bound (and the limit as $m \rightarrow \infty$) of $\overline{C}_{loc}(P) - (k+1)$. This, however, ignores the effect of the fact that after a shift of length $d < m-k$ we know that at least one and at most $k+1$ of characters $p_{m-d-k}, \ldots, p_{m-d}$ will match. Hence to bound $\overline{C}_{loc}(P) - (k+1)$ properly, it surely suffices to add $k+1$ to the above bound which gives

$$\overline{C}_{loc}(P) - (k+1) < \frac{k+1}{c-1} + k + 1$$

and the lemma follows. □

Let $S(P)$ be a random variable denoting the length of the shift in Algorithm 4 for pattern P and for some fixed k and c when scanning a random T. Moreover, let P_0 be a pattern that repeatedly contains all characters in Σ in some fixed order until the length of P_0 equals m. Then it is not difficult to see that P_0 gives on the average the minimal shift, that is, the expected values satisfy $\overline{S}(P_0) \leq \overline{S}(P)$ for all P of length m. Hence a lower bound for $\overline{S}(P_0)$ gives a lower bound for the expected shift over all patterns of length m (c.f. [Bae89b]).

Lemma 2. $\overline{S}(P_0) \geq \frac{1}{2} min(\frac{c}{k+1}, m-k)$. Moreover, $\overline{S}(P_0) \geq 1$.

Proof. Let $t = min(c - 1, m - k - 1)$. Then the possible lengths of a shift are $1, 2, ..., t + 1$. Therefore

$$\bar{S}(P_0) = \sum_{i=0}^{t} Pr(S(P_0) > i)$$

where $Pr(A)$ denotes the probability of event A. Then

$$Pr(S(P_0) > i\ ) = \left(\frac{c - i}{c}\right)^{k+1}$$

because for each of the $k + 1$ text symbols that are compared with the pattern to determine the shift (step 8 of Algorithm 4), there are i characters not allowed to occur as the text symbols. Otherwise the shift would not be $> i$. Hence

$$\bar{S}(P_0) = \sum_{i=0}^{t} \left(1 - \frac{i}{c}\right)^{k+1}$$

which clearly is ≥ 1, because $t \geq 0$ as we may assume that $c \geq 2$ and that $k < m$.

We divide the rest of the proof into two cases.

Case 1: $m - k < \frac{c}{k+1}$. Then $t = m - k - 1$, and we have

$$\begin{aligned}
\bar{S}(P_0) &\geq \sum_{i=0}^{m-k-1} \left(1 - \frac{k+1}{c} \cdot i\right) \\
&= m - k - \frac{k+1}{c} \cdot \frac{(m-k-1)(m-k)}{2} \\
&\geq (m-k)\left(1 - \frac{k+1}{c} \cdot \frac{m-k}{2}\right) \geq \frac{1}{2}(m-k).
\end{aligned}$$

Case 2: $m - k \geq \frac{c}{k+1}$. Then $t \geq \left\lceil \frac{c}{k+1} \right\rceil - 1$, and we have

$$\bar{S}(P_0) \geq \sum_{i=0}^{\left\lceil \frac{c}{k+1} \right\rceil - 1} \left(1 - \frac{i}{c}\right)^{k+1} \geq \sum_{i=0}^{\left\lceil \frac{c}{k+1} \right\rceil - 1} \left(1 - \frac{k+1}{c} \cdot i\right)$$

$$= \left\lceil \frac{c}{k+1} \right\rceil - \frac{k+1}{c} \cdot \frac{1}{2} \cdot \left\lceil \frac{c}{k+1} \right\rceil \left(\left\lceil \frac{c}{k+1} \right\rceil - 1 \right)$$

$$\geq \left\lceil \frac{c}{k+1} \right\rceil \left(1 - \frac{1}{2} \cdot \frac{k+1}{c} \cdot \frac{c}{k+1} \right) = \frac{1}{2} \left\lceil \frac{c}{k+1} \right\rceil . \square$$

Consider finally the total expected number $\overline{C}(P)$ of character comparisons when Algorithm 4 scans a random T with pattern P. Let $f(P)$ be the random variable denoting the number of shifts taken during the execution, and let $\overline{f}(P)$ be its expected value. Then we have

$$\overline{C}(P) = \overline{f}(P) \cdot \overline{C}_{loc}(P).$$

To estimate $\overline{f}(P)$, we let S_i be a random variable denoting the length of i^{th} shift. At the start of Algorithm 4, P is aligned with T such that its first symbol corresponds to the text position 1, and at the end P is aligned such that its first symbol corresponds to some text position $\leq n - m + k + 1$ but the next shift would lead to a position $> n - m + k + 1$. Hence new shifts are taken until the total length of the shifts exceeds $n - m + k$. This implies that $f(P)$ equals the largest index ϕ such that

$$\sum_{i=1}^{\phi} S_i \leq n - m + k.$$

Assume now that the different variables S_i are independent, that is, the shift lengths are independent; note that this simplification is not true for two successive shifts such that the first one is shorter than $k + 1$. Then all variables S_i have a common distribution with expected value $\overline{S}(P) \geq \overline{S}(P_0)$. Under this assumption

$$\{ \sum_{i=1}^{\phi} S_i \}$$

is, in fact, a *pure renewal process* within interval $[0, n - m + k]$ in the terminology of [Fel66, Chapter XI]. Then the expected value of ϕ is $(n - m + k) / \overline{S}(P)$ for large $n - m + k$ (see [Fel66, p. 359]) Hence

$$\overline{f}(P) = O\left(\frac{n - m + k}{\overline{S}(P_0)} \right)$$

and by Lemma 2,

$$\overline{f} = O\left(max\left(\frac{k+1}{c}, \frac{1}{m-k}\right) \cdot (n - m + k)\right).$$

Recalling finally that $\overline{C}(P) = \overline{f}(P) \cdot \overline{C}_{loc}(P)$ and applying Lemma 1, we obtain that

$$\overline{C}(P) \leq O\left(max\left(\frac{k+1}{c}, \frac{1}{m-k}\right)(n - m + k)\left(\frac{c}{c-1} + 1\right)(k+1)\right)$$

which is $O(\frac{nk^2}{c} + \frac{nk}{m-k})$ as $n >> m$. Hence we have:

Theorem 1. The expected running time of Algorithm 4 is $O(nk(\frac{k}{c} + \frac{1}{m-k}))$, if the lengths of different shifts are mutually independent. The preprocessing time is $O(m + kc)$, and the working space is $O(kc)$.

Removing the independence assumption from Theorem 1 remains open.

3. The *k* differences problem

3.1. Basic solution by dynamic programming

The *edit distance* [WaF75, Ukk85a] between two strings, A and B, 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 $a \rightarrow \varepsilon$ (a deletion), $\varepsilon \rightarrow b$ (an insertion), or $a \rightarrow b$ (a change) where a, b are in Σ and ε is the empty string.

The *k differences problem* is, given pattern $P = p_1p_2...p_m$ and text $T = t_1t_2...t_n$ and an 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 by the following dynamic programming method [Sel80, Ukk85b]: Let D be a $m + 1$ by $n + 1$ table such that $D(i, j)$ is the minimum edit distance between $p_1p_2...p_i$ and any substring of T ending at t_j. Then

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

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

Table D can be evaluated column-by-column in time $O(mn)$. Whenever $D(m, j)$ is found to be $\le k$ 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.

3.2. Boyer-Moore approach

Our algorithm contains two main phases: the scanning and the checking. The scanning phase scans over the text and marks the parts that contain all the approximate occurrences of P. This is done by marking some entries $D(0, j)$ on the first row of D. The checking phase then evaluates all diagonals of D whose first entries are marked. This is done by the basic dynamic programming restricted to the marked diagonals. Whenever the dynamic programming refers to an entry outside the diagonals, the entry can be taken to be ∞. Because this is quite straightforward we do not describe it in detail. Rather, we concentrate on the scanning part.

The scanning phase repeatedly applies two operations: mark and shift. The shift operation is based on a Boyer-Moore idea. The mark operation decides whether or not the current alignment of the pattern with the text needs accurate checking by dynamic programming and in the positive case marks certain diagonals. To understand the operations we need the concept of a minimizing path in table D.

For every $D(i, j)$, there is a *minimizing arc* from $D(i-1, j)$ to $D(i, j)$ if $D(i, j) = D(i-1, j) + 1$, from $D(i, j-1)$ to $D(i, j)$ if $D(i, j) = D(i, j-1) + 1$, and from $D(i-1, j-1)$ to $D(i, j)$ if $D(i, j) = D(i-1, j-1)$ when $p_i = t_j$ or if $D(i, j) = D(i-1, j-1) + 1$ when $p_i \ne t_j$. The costs of the arcs are 1, 1, 0 and 1, respectively. The minimizing arcs show the actual dependencies between the values in table D. A *minimizing path* is any path that consists of minimizing arcs and leads from an entry $D(0, j)$ on the first row of D to an entry $D(m, h)$ on the last row of D. Note that $D(m, h)$ equals the sum of the costs of the arcs on the path. A minimizing path is *successful* if it leads to an entry $D(m, h) \le k$.

A *diagonal* h of D for $h = -m, ..., n$, consists of all $D(i, j)$ such that $j - i = h$. As any vertical or horizontal minimizing arc adds 1 to the value of the entry, the next lemma easily follows:

Lemma 3. The entries on a successful minimizing path are contained in $\le k + 1$ successive diagonals of D.

Our marking method is based on the following lemma. For each $i = 1, ..., m$, let the k *environment* of the pattern symbol p_i be the string $C_i = p_{i-k} \dots p_{i+k}$, where $p_j = \varepsilon$ for $j < 1$ and $j > m$.

Lemma 4. Let a successful minimizing path go through some entry on a diagonal h of D. Then for at most k indexes i, $1 \le i \le m$, character t_{h+i} does not occur in k environment C_i.

Proof. Column j, $h + 1 \le j \le h + m$, of D is called *bad* if t_j does not appear in C_{j-h}. The lemma claims that the number of the bad columns is $\le k$. Let M be the path in the lemma. Let R be the set of indexes j, $h + 1 \le j \le h + m$, such that path M contains at least one entry $D(i, j)$ on column j of D. If M starts or ends outside diagonal h, then the size of R can be $< m$. Then, however, M must have at least one vertical arc for each index j missing in R because M crosses diagonal h. Therefore $vert(M) \ge m - size(R)$ where $vert(M)$ is the number of vertical arcs of M.

By Lemma 3, M must be contained in diagonals $h - k, h - k + 1, ..., h + k$ of D. Hence for each j in R, path M must enter some entry on column j restricted to diagonals $h - k, ..., h + k$, that is, some entry $D(i - k, j), ..., D(i + k, j)$. Then if j is bad, the first arc in M that enters column j must add 1 to the total cost of M. Because such an arc enters a new column, it must be either a diagonal or a horizontal arc; note that with no restriction on generality we may assume that the very first arc of M is not a vertical one. Hence the number of bad columns in R is $\leq cost(M) - vert(M)$ where $cost(M)$ is the value of the final entry of M.

Moreover, there can be $m - size(R)$ additional bad columns as every column outside R can be bad. The total number of the bad columns is therefore at most $m - size(R) + cost(M) - vert(M) \leq cost(M) \leq k$. □

Lemma 4 suggests the following marking method. For diagonal h, check for $i = m, m - 1, ..., k + 1$ if t_{h+i} is in C_i until $k + 1$ bad columns are found. Note that to get minimum shift $k + 1$ (see below) we stop already at $i = k + 1$ instead of at $i = 1$. If the number of bad columns is $\leq k$, then mark diagonals $h - k, ..., h + k$, that is, mark entries $D(0, h - k), ..., D(0, h + k)$.

For finding the bad columns fast we need a 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 k environment C_i.

Clearly, the table can be computed by a simple scanning of P in time $O((c + k)m)$.

After marking we have to determine the length of shift, that is, what is the next diagonal after h around which the marking should eventually be done. The marking heuristics ensures that all successful minimizing paths that are properly before diagonal $h + k + 1$ are already marked. Hence we can safely make at least a shift of $k + 1$ to diagonal $h + k + 1$.

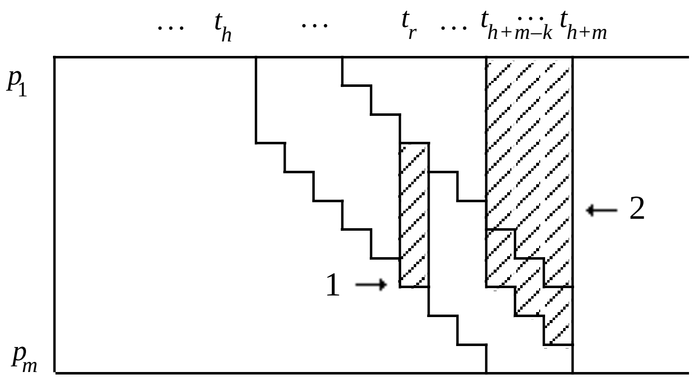

Figure 2. Mark and shift (k = 2).

This can be combined with the shift heuristics of Algorithm 4 of Section 2 based on table d_k. So we determine the first diagonal after h, say $h + d$, where at least one of the characters t_{h+m}, t_{h+m-1}, ..., t_{h+m-k} matches with the corresponding character of P. This is correct, because then there can be a successful minimizing path that goes through diagonal $h + d$. The value of d is evaluated as in Algorithm 4, using exactly the same precomputed table d_k. Note that unlike in the case of Algorithm 4, the maximum allowed value of d is now m, not $m - k$, as the marking starts from diagonal $h - k$, not from h. Finally, the maximum of $k + 1$ and d is the length of the shift.

In practice, the marking and the computation of the shift can be merged if we start the searching for the bad columns from the end of the pattern.

Fig. 2 illustrates marking and shifting. For $r = h + m$, $h + m - 1$, ..., $h + k + 1$ we check whether or not t_r appears among the pattern symbols corresponding to the shaded block 1 (the k environment). If $k + 1$ symbols t_r that do not appear are found, entries $D(0, h - k)$, ..., $D(0, h + k)$ are marked. Simultaneously we check what is the next diagonal after h containing a match between P and t_{h+m-k}, ..., t_{h+m} (shaded block 2). The next shift is to this diagonal but at least to diagonal $h + k + 1$.

We get the following algorithm for the scanning phase:

Algorithm 5. The scanning phase for the k differences problem.

```
1.   compute table Bad and, by Algorithm 3, table d_k from P;
2.   j := m;
3.   while j ≤ n + k do begin
4.       r := j; i := m;
5.       bad := 0;                          {bad counts the bad indexes}
6.       d := m;                            {initial value of shift}
7.       while i > k and bad ≤ k do begin
8.           if i ≥ m – k then d := min(d, d_k[i, t_r]);
9.           if Bad(i, t_r) then bad := bad + 1;
10.          i := i – 1; r := r – 1 end;
11.      if bad ≤ k then
12.          mark entries D(0, j – m – k), ..., D(0, j – m + k);
13.      j := j + max(k + 1, d) end
```

The loop in steps 7–9 can be slightly optimized by splitting it into two parts such that the first one handles $k + 1$ text characters and computes the length of shift, and the latter goes on counting bad indexes (a similar optimization also applies to Algorithm 4).

3.3. Analysis

The preprocessing of P requires $O((k + c)m)$ for computing table Bad and $O(m + kc)$ for computing table d_k. As $k < m$, the total time is $O((k + c)m)$. The working space is $O(cm)$.

The marking and shifting by Algorithm 5 takes time $O(mn/k)$ in the worst case. The analysis of the average case is similar to the analysis of Algorithm 4 in Section 2. Let $B_{loc}(P)$ be a random variable denoting, for some fixed c and k, the number of the columns examined (step 9 of Algorithm 5) until $k + 1$ bad columns are found and the next shift will be taken. Obviously, $B_{loc}(P)$

corresponds to $C_{loc}(P)$ of Lemma 1. For the expected value $\overline{B}_{loc}(P)$ we show the following rough bound:

Lemma 5. Let $2k + 1 < c$. Then $\overline{B}_{loc}(P) \leq \left(\frac{c}{c - 2k - 1} + 1\right)(k + 1)$.

Proof. The expected value of $B_{loc}(P) - (k + 1)$ can be bounded from above by the expected value of the negative binomial distribution with parameters ($k + 1$, q) where q is a lower bound for the probability that a column is bad. Recall that column j is called bad if text symbol t_j does not occur in the corresponding k environment. As the k environment is a substring of P of length at most $2k + 1$, it can have at most $2k + 1$ different symbols. Therefore the probability that a random t_j does not belong to the symbols of a k environment is at least $\frac{c - (2k + 1)}{c}$. Hence we can choose $q = \frac{c - (2k + 1)}{c}$.

The negative binomial distribution would then give for $\overline{B}_{loc}(P) - (k + 1)$ an upper bound $\frac{(2k + 1)(k + 1)}{c - (2k + 1)}$. However, the shift heuristic implies that after a shift of length $< m$ we know that at least one and at most $k + 1$ columns will not be bad. Hence to bound $\overline{B}_{loc}(P) - (k + 1)$ properly, we have to add $k + 1$ to the above bound which gives

$$\overline{B}_{loc}(P) - (k + 1) \leq \frac{(2k + 1)}{c - (2k + 1)}(k + 1) + k + 1$$

and the lemma follows. □

Let $S'(P)$ be a random variable denoting the length of the shift in Algorithm 5 for pattern P and for some fixed k and c. When scanning a random T, the special pattern P_0 again gives the shortest expected shift, that is, $\overline{S}'(P_0) \leq \overline{S}'(P)$ for all P of length m. Lemma 6 gives a bound for $\overline{S}'(P_0)$.

Lemma 6. $\overline{S}'(P_0) \geq \frac{1}{2} min(\frac{c}{k + 1}, m)$.

Proof. Let $t = min(c - 1, m - 1)$. Then the possible lengths of a shift are 1, 2, ..., $t + 1$; note that a shift actually is always $\geq k + 1$ according to our heuristic, but the heuristic can be ignored here as our goal is to prove a lower bound. Therefore

$$\overline{S}'(P_0) = \sum_{i=0}^{t} Pr(S'(P_0) > i).$$

If $0 \le i \le m - k - 1$, then

$$Pr(S'(P_0) > i) = \left(\frac{c - i}{c}\right)^{k+1}$$

because for each of the $k + 1$ text symbols that are compared with the pattern to determine the shift (step 8 of Algorithm 5), there are i characters not allowed to occur as the text symbols. This is exactly as in the proof of Lemma 2. A slight difference arises when $m - k \le i \le m - 1$. Then

$$Pr(S'(P_0) > i) =$$

$$\left(\frac{c - i}{c}\right)^{m-i} \cdot \frac{c - i + 1}{c} \cdot \frac{c - i + 2}{c} \cdot \ldots \cdot \frac{c - m + k + 1}{c}$$

because now the number of forbidden characters is i for the $m - i$ last text symbols and $i - 1, i - 2, \ldots, i - (m - k - 1)$ for the remaining $k + 1 - (m - i)$ text symbols, listed from right to left. But also in this case

$$Pr(S'(P_0) > i) \ge \left(\frac{c - i}{c}\right)^{k+1}.$$

Hence

$$\overline{S}'(P_0) \ge \sum_{i=0}^{t} \left(1 - \frac{i}{c}\right)^{k+1}.$$

The rest of the proof is divided into two cases which are so similar to the cases in the proof of Lemma 2 that we do not repeat the details. If $m < \frac{c}{k+1}$, then $\overline{S}'(P_0) \ge \frac{1}{2} m$. If $m \ge \frac{c}{k+1}$, then $\overline{S}'(P_0) \ge \frac{1}{2}\left\lceil \frac{c}{k+1} \right\rceil$. □

As the length of a shift is always $\ge k + 1$, we get from Lemma 6

$$\overline{S}'(P) \ge \overline{S}'(P_0)$$

$$\geq max\left(k + 1,\ min\left(\frac{c}{2(k + 1)}, \frac{m}{2}\right)\right)$$

$$= min\left(max\left(k + 1, \frac{c}{2(k + 1)}\right), max\left(k + 1, \frac{m}{2}\right)\right)$$

$$\geq \frac{1}{2} min\left(k + 1 + \frac{c}{2(k + 1)}, \frac{m}{2}\right).$$

The number of text positions at which a right-to-left scanning of P is performed between two shifts is again

$$O\left(\frac{n - m}{\overline{S}'(P)}\right) = O\left(\frac{n - m}{\overline{S}'(P_0)}\right).$$

This can be shown as in the analysis of Algorithm 4. Note that for Algorithm 5 we need not assume explicitly that the lengths of different shifts are independent. They are independent as the length of the minimum shift is k + 1.

Hence the expected scanning time of Algorithm 5 for pattern P is

$$O\left(\overline{B}_{loc}(P) \cdot \frac{n - m}{\overline{S}'(P)}\right).$$

When we apply here the upper bound for $\overline{B}_{loc}(P)$ from Lemma 5 and the above lower bound for $\overline{S}'(P)$, and simplify, we obtain our final result.

Theorem 2. Let $2k + 1 < c$. Then the expected scanning time of Algorithm 5 is $O(\frac{c}{c - 2k}) \cdot kn \cdot (\frac{k}{c + 2k^2} + \frac{1}{m}))$. The preprocessing time is $O((k + c)m)$ and the working space $O(cm)$.

The checking of the marked diagonals can be done after Algorithm 5 or in cascade with it in which case a buffer of length $2m$ is enough for saving the relevant part of text T. The latter approach is presented in Algorithm 6, which contains a modification of Algorithm 5 as its subroutine, function *NPO*.

Algorithm 6. The total algorithm for the *k* differences problem.

```
function NPO; begin              {the next possible occurrence}
while j ≤ n + k do begin
        r := j; i := m; bad := 0; d := m;
        while i > k and bad ≤ k do begin
                if i ≥ m – k then d := min(d, d_k[i, t_r]);
                if Bad(i, t_r) then bad := bad + 1;
                i := i – 1; r := r – 1 end;
        if bad ≤ k then goto out;
        j := j + max(k + 1, d) end
out: if j ≤ n + k then begin
        NPO := j – m – k;
        j := j + max(k + 1, d) end
else NPO := n + 1 end;

compute tables Bad and d_k;
j := m;
for i := 0 to m do H_0[i] := i;
H := H_0;
top := min(k + 1, m);            {top – 1 is the last row with the value ≤ k}
col := NPO;
lastcol := col + m + 2k – 1;
while col ≤ n do
        for r := col to lastcol do begin
                c := 0;
                for i := 1 to top do begin
                        if p_i = t_r then d := c;
                        else d := min((H[i – 1], H[i], c)) + 1;
                        c := H[i]; H[i] := d end;
                while H(top) > k do top := top – 1;
                if top = m then report match at j;
                else top := top + 1 end;
        next := NPO;
        if next > lastcol + 1 then begin
                H := H_0;
                top := min(k + 1, m);
                col := next end
        else col := lastcol + 1;
        lastcol := next + m + 2k – 1 end
```

The checking phase of Algorithm 6 evaluates a part of D by dynamic programming (see Section 3.1). Because entries on every diagonal are monotonically increasing [Ukk85a], the computation along a marked 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. Algorithm 6 implements this idea in a slightly streamlined way. Instead of restricting the evaluation of D exactly on the marked diagonals (which could be done, of course, but leads to more complicated code), we evaluate each *column* of D that intersects some marked diagonal. Each such column is evaluated from its first entry to the last one that could be $\leq k$. This can be easily decided using the diagonalwise monotonicity of D [Ukk85b]. The evaluation of each separate block of columns can start from a column identical to the first column of D (H_0 in Algorithm 6; H stores the previous as well as the current column under evaluation). For random strings, this method spends expected time of $O(k)$ on each column (this conjecture of [Ukk85b] has recently been proved by W. Chang). Hence the total expected time of the checking phase remains $O(kn)$.

Asymptotically, steps 22–37 of Algorithm 6 are executed very seldom. Hence except for small patterns, small alphabets and large k's, the expected time for the checking phase tends to be small in which case the time bound of Theorem 2 is valid for our entire algorithm.

3.4. Variations

Each marking operation before the next shift takes time $O(m)$ in the worst case. At the cost of decreased accuracy of marking we can reduce this by limiting the number of the columns whose badness is examined. The time reduces to $O(k)$ when we examine only at most ak columns for some constant $a > 1$. If there are not more than k bad columns among them, then the diagonals are marked. This variation appealingly has the feature that the total time of marking and shifting reduces to $O(n)$ in the worst case. Of course, the gain may be lost in the checking phase, as more diagonals will be marked.

On the other hand, the accuracy of the marking heuristic, which quite often conservatively marks too many diagonals in its present form, can be improved by a more careful analysis of whether or not a column is bad. Such an analysis can be based, at the cost of longer preprocessing, on the observation that two matches on successive columns of D can occur in the same minimizing path only if they are on the same diagonal.

In Algorithm 6, the width of the band of columns inspected is $m + 2k$. The algorithm works better for small alphabets and short patterns, if a wider width is used, because that will reduce reinspection of text positions during the scanning phase. If the width is at least $2m + k$, then we can in the case of a potential match make a shift of $m + 1$, which guarantees that no text position is reinspected in that situation.

4. Experiments and conclusions

We have tested extensively our algorithms and compared them with other methods. We will present results of a comparison with the $O(kn)$ expected time dynamic programming method [Ukk85b] which we have found to be the best in practice among the old algorithms we have tested [JTU90].

Table 1 shows total execution times of Algorithms 4 and 6 and the corresponding dynamic programming algorithms *DP1* (the k mismatches problem) and *DP2* (the k differences problem). Preprocessing, scanning and checking times are specified for Algorithm 6, as well as preprocessing times for Algorithm 4. In our tests, we used random patterns of varying lengths and random texts of length 100,000 characters over alphabets of different sizes. The tests were run on a VAX 8800 under VMS. In order to decrease random variation, the figures of Table 1 are averages of ten runs. Still more repetitions should be necessary to eliminate variation as can seen in the duplicate entries of Table 1 corresponding to different test series with the same parameters.

Figures 3–6 have been drawn from the data of Table 1. Figures 3 and 4 show the total execution times when k = 4 and m varies for alphabet sizes c = 2 and 90. Figures 5 and 6 show the corresponding times when m = 8 and k varies for alphabet sizes c = 4 and 30.

Our algorithms, as all algorithms of Boyer-Moore type, work very well for large alphabets, and the execution time decreases when the length of the pattern grows. An increment of the error limit k slows down our algorithms more than the dynamic programming algorithms. Observe also that the Boyer-Moore approach is relatively better in solving the k differences problem than in solving the k mismatches problem.

Our methods turned out to be faster than the previous methods, when the pattern is long enough (m > 5), the error limit k is relatively small and the alphabet is not very small (c > 5). Results of the practical experiments are consistent with our theoretical analysis. To devise a more accurate and complete theoretical analysis of the algorithms is left as a subject for further study.

Table 1. Execution times (in units of 10 milliseconds) of the algorithms (n = 100,000). Prepr., Scan and Check denote the preprocessing, scanning and checking times, respectively.

c	m	k	ALG. 4		DP1	ALG. 6				DP2
			Prepr.	Total		Prepr.	Scan	Check	Total	
2	8	4	0	574	227	0	129	406	535	403
2	16	4	0	681	403	0	240	705	945	700
2	32	4	0	681	371	0	451	759	1210	756
2	64	4	0	679	385	0	881	813	1694	817
2	128	4	0	688	349	0	1762	792	2554	786
2	256	4	0	691	361	0	3172	827	3999	824
4	8	4	0	451	213	0	129	469	598	465
4	16	4	0	453	224	0	235	557	792	553
4	32	4	0	447	222	0	427	731	1158	550
4	64	4	0	464	227	0	700	538	1238	563
4	128	4	0	459	226	0	849	216	1065	556
4	256	4	0	436	226	0	724	2	726	553
30	8	4	0	151	174	0	84	84	168	406
30	16	4	0	88	170	0	75	0	75	410
30	32	4	0	78	167	0	72	0	72	406
30	64	4	0	75	167	0	70	0	70	403
30	128	4	0	79	167	1	73	0	74	404
30	256	4	0	79	167	1	73	0	74	403
90	8	4	0	126	166	0	63	2	65	389
90	16	4	0	50	164	0	40	0	40	389
90	32	4	0	33	166	0	30	0	30	390
90	64	4	0	27	165	1	25	0	26	389
90	128	4	0	27	164	2	26	0	28	388

90	256	4	1	27	164	4	27	0	31	387
2	8	0	0	89	102	0	106	9	115	164
2	8	1	0	234	155	0	260	246	506	278
2	8	2	0	371	193	0	208	361	569	353
2	8	3	0	488	220	0	158	405	563	399
2	8	4	0	570	223	0	127	405	533	404
2	8	5	0	628	223	0	109	407	516	407
2	8	6	0	677	221	0	93	405	498	401
4	8	0	0	56	78	0	63	0	63	129
4	8	1	0	95	113	0	112	43	155	229
4	8	2	0	211	153	0	199	358	557	353
4	8	3	0	344	175	0	158	415	573	408
4	8	4	0	480	211	0	128	447	575	445
4	8	5	0	575	225	0	108	481	589	477
4	8	6	0	582	232	0	98	505	603	503
30	8	0	0	16	68	0	18	0	18	115
30	8	1	0	36	93	0	32	0	32	187
30	8	2	0	63	120	0	54	0	54	263
30	8	3	0	102	144	0	68	5	73	336
30	8	4	0	157	169	0	79	44	123	412
30	8	5	0	222	194	0	84	170	254	484
30	8	6	0	364	219	0	90	519	609	548
90	8	0	0	15	67	0	16	0	16	114
90	8	1	0	32	93	0	29	0	29	189
90	8	2	0	55	119	0	40	0	40	258
90	8	3	0	87	144	0	53	0	53	332
90	8	4	0	132	170	0	63	1	64	408
90	8	5	0	208	198	0	78	37	115	484
90	8	6	0	344	221	0	84	207	291	554

Figures 3–6 have been drawn from the data of Table 1. Figures 3 and 4 show the total execution times when k = 4 and m varies for alphabet sizes c = 2 and 90. Figures 5 and 6 show the corresponding times when m = 8 and k varies for alphabet sizes c = 4 and 30.

Our algorithms, as all algorithms of Boyer-Moore type, work very well for large alphabets, and the execution time decreases when the length of the pattern grows. An increment of the error limit k slows down our algorithms more than the dynamic programming algorithms. Observe also that the Boyer-Moore approach is relatively better in solving the k differences problem than in solving the k mismatches problem.

Our methods turned out to be faster than the previous methods, when the pattern is long enough (m > 5), the error limit k is relatively small and the alphabet is not very small (c > 5). Results of the practical experiments are consistent with our theoretical analysis. To devise a more accurate and complete theoretical analysis of the algorithms is left as a subject for further study.

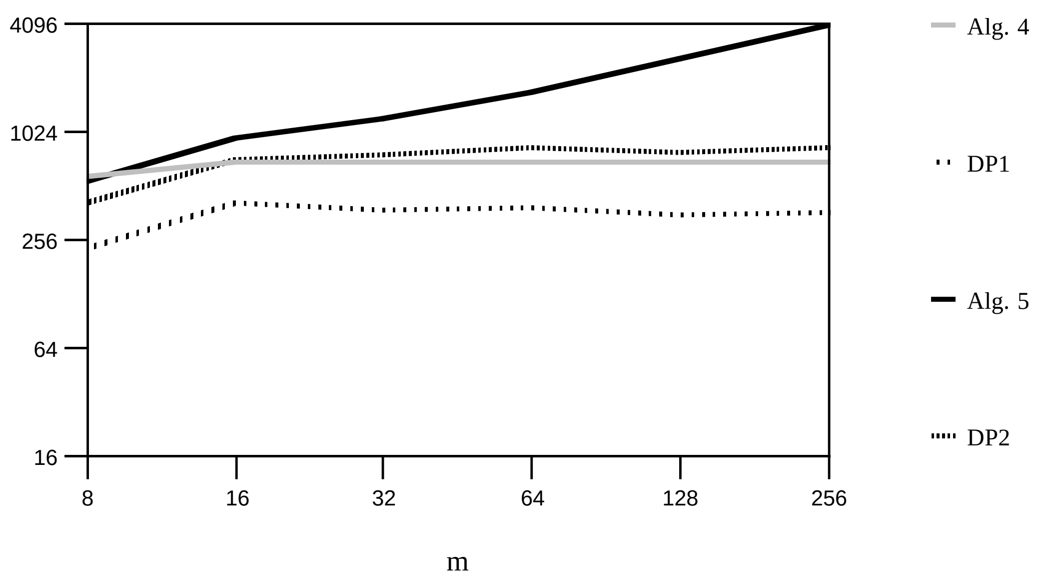

Figure 3. Total times for k = 4 and c = 2.

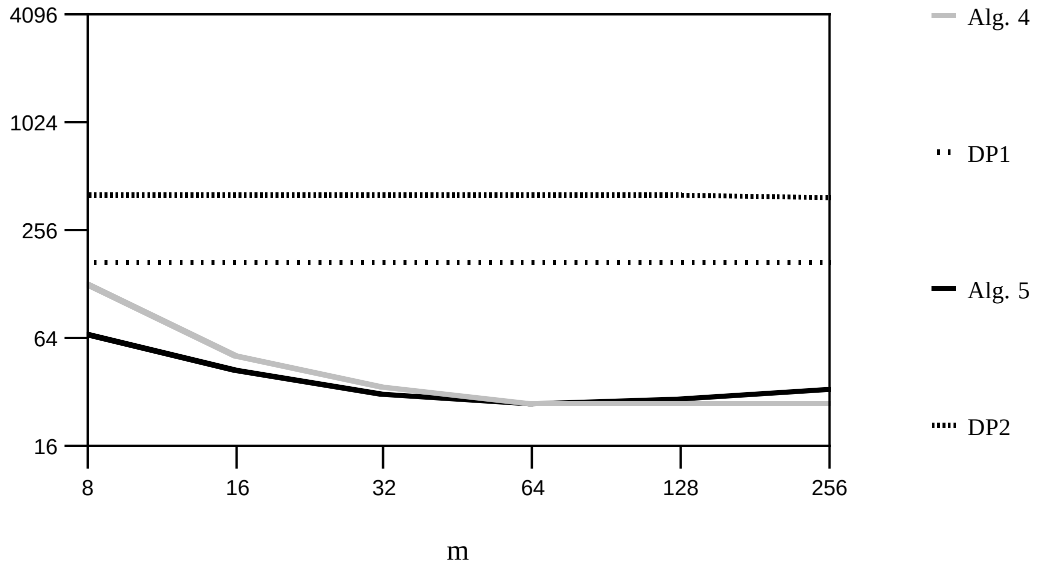

Figure 4. Total times for k = 4 and c = 90.

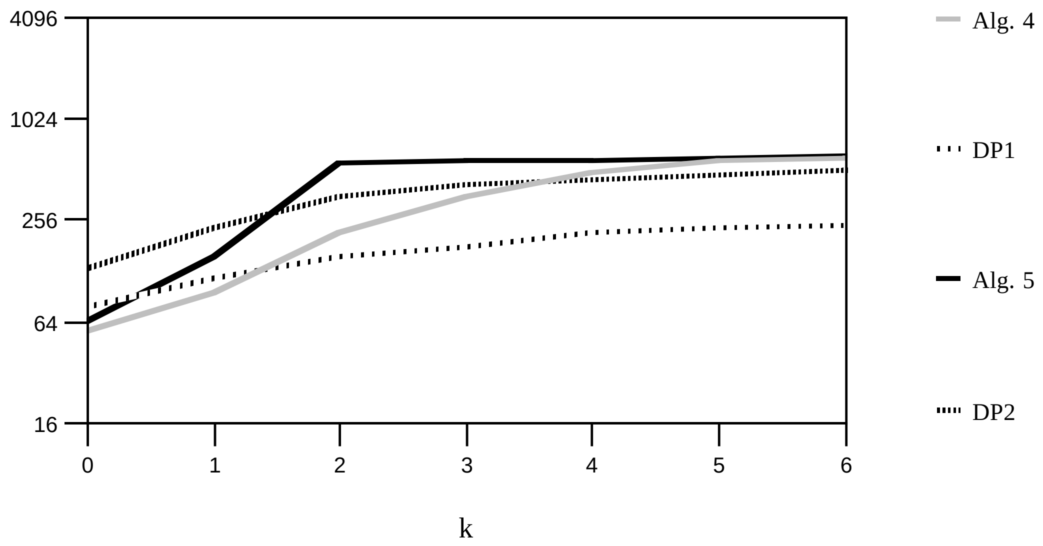

Figure 5. Total times for m = 8 and c = 4.

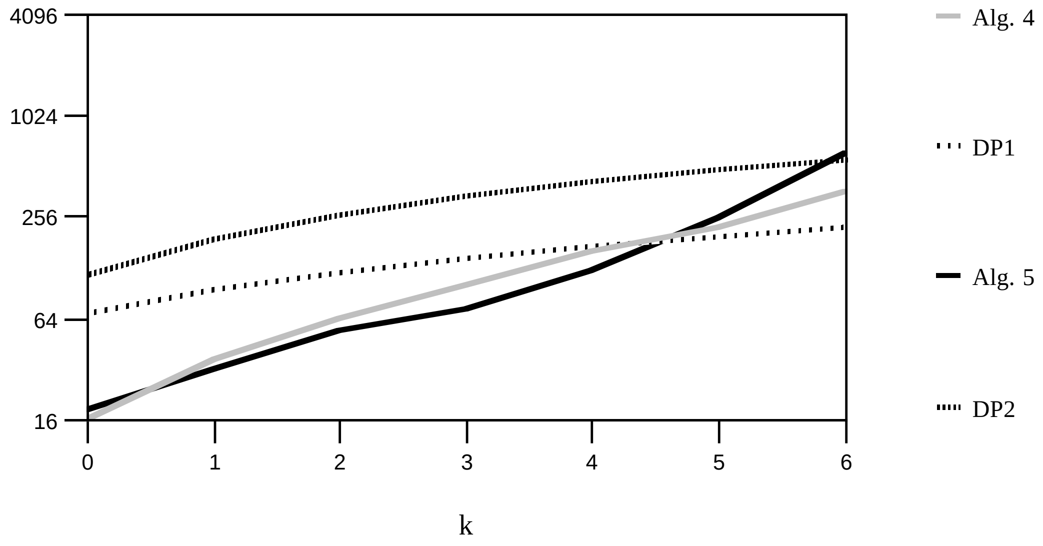

Figure 6. Total times for m = 8 and c = 30.

Acknowledgement

Petteri Jokinen performed the experiments which is gratefully acknowledged.

References

[Bae89a] R. Baeza-Yates: Efficient Text Searching. Ph.D. Thesis, Report CS-89-17, University of Waterloo, Computer Science Department, 1989.

[Bae89b] R. Baeza-Yates: String searching algorithms revisited. In: *Proceedings of the Workshop on Algorithms and Data Structures* (ed. F. Dehne et al.), Lecture Notes in Computer Science 382, Springer-Verlag, Berlin, 1989, 75–96.

[BoM77] R. Boyer and S. Moore: A fast string searching algorithm. *Communcations of the ACM* **20** (1977), 762–772.

[ChL90] W. Chang and E. Lawler: Approximate string matching in sublinear expected time. In: *Proceedings of the 31st IEEE Annual Symposium on Foundations of Computer Science*, IEEE, 1990, 116–124.

[Fel65] W. Feller: *An Introduction to Probability Theory and Its Applications.* Vol. I. John Wiley & Sons, 1965.

[Fel66] W. Feller: *An Introduction to Probability Theory and Its Applications.* Vol. II. John Wiley & Sons, 1966.

[GaG86] Z. Galil and R. Giancarlo: Improved string matching with k mismatches. *SIGACT News* **17** (1986), 52–54.

[GaG88] Z. Galil and R. Giancarlo: Data structures and algorithms for approximate string matching. *Journal of Complexity* **4** (1988), 33–72.

[GaP89] Z. Galil and K. Park: An improved algorithm for approximate string matching. *Proceedings of the 16th International Colloquium on Automata, Languages and Programming*, Lecture Notes in Computer Science 372, Springer-Verlag, Berlin, 1989, 394–404.

[GrL89] R. Grossi and F. Luccio: Simple and efficient string matching with k mismatches. *Information Processing Letters* **33** (1989), 113–120.

[Hor80] N. Horspool: Practical fast searching in strings. *Software Practice & Experience* **10** (1980), 501–506.

[JTU90] P. Jokinen, J. Tarhio and E. Ukkonen: A comparison of approximate string matching algorithms. In preparation.

[Kos88] S. R. Kosaraju: Efficient string matching. Extended abstract. Johns Hopkins University, 1988.

[KMP77] D. Knuth, J. Morris and V. Pratt: Fast pattern matching in strings. *SIAM Journal on Computing* **6** (1977), 323–350.

[LaV88] G. Landau and U. Vishkin: Fast string matching witk k differences. *Journal of Computer and System Sciences* **37** (1988), 63–78.

[LaV89] G. Landau and U. Vishkin: Fast parallel and serial approximate string matching. *Journal of Algorithms* **10** (1989), 157–169.

[Sel80] P. Sellers: The theory and computation of evolutionary distances: Pattern recognition. *Journal of Algorithms* **1** (1980), 359–372.

[Ukk85a] E. Ukkonen: Algorithms for approximate string matching. *Information Control* **64** (1985), 100–118.

[Ukk85b] E. Ukkonen: Finding approximate patterns in strings. *Journal of Algorithms* **6** (1985), 132–137.

[UkW90] E. Ukkonen and D. Wood: Fast approximate string matching with suffix automata. Report A-1990-4, Department of Computer Science, University of Helsinki, 1990.

[WaF75] R. Wagner and M. Fischer: The string-to-string correction problem. *Journal of the ACM* **21** (1975), 168–173.