EFFICIENT STRING MATCHING WITH k MISMATCHES

Gad M. LANDAU

Department of Computer Science, School of Mathematical Sciences, Tel Aviv University, Tel Aviv, Israel

Uzi VISHKIN *

Department of Computer Science, Courant Institute of Mathematical Sciences, New York University, New York, NY 10012, U.S.A.

Communicated by M. Nivat
Received March 1985
Revised October 1985

Abstract. Given a text of length n, a pattern of length m, and an integer k, we present an algorithm for finding all occurrences of the pattern in the text, each with at most k mismatches. The algorithm runs in $O(k(m \log m+n))$ time.

1. Introduction

The problem of *string matching with k mismatches* is defined as follows. Suppose we are given a text of length n, a pattern of length m and an integer k. Find all occurrences of the pattern in the text with at most k locations in which the text and the pattern have different symbols.

Example 1.1. Let the text be *bbababacaacbb*, the pattern *aaaaabaaab*, and $k=4$. Let us see whether there is an occurrence with $\leqslant k$ mismatches that starts at the third location of the text.

```
    a a a a a b a a a b
b b a b a b a c a a c b b
      |   |   |     |
```

In only four locations the text and the pattern have different symbols, implying that an occurrence of the pattern starts at the third location of the text.

Note that the case $k=0$ is the extensively studied string matching problem. Let us mention a few notable algorithms for the string matching problem: linear time

* Present affiliation: Department of Computer Science, School of Mathematical Sciences, Tel Aviv University, Tel Aviv, Israel. The research of this author was supported by NSF Grants NSF-DCR-8318874 and NSF-DCR-8413359, ONR Grant N00014-85-K-0046 and by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U.S. Department of Energy under Contract Number DE-AC02-76ER03077.

serial algorithms [2, 4, 6, 9], [7] (a randomized algorithm), and parallel algorithms [3, 9]. The problem has a strong pragmatical flavor. In practice, we often need to analyze situations where the data are not completely reliable. Specifically, consider a situation where the strings which are the input for our problem contain errors and we still need to find all possible occurrences of the pattern in the text as in reality. Assuming some bound on the number of errors would clearly imply our problem. We present an algorithm for string matching with k mismatches which runs in time $O(k(m \log m+n))$ on a random-access-machine (RAM) [1].

Postscript

After all the results in the present paper have been achieved, we want to make the following remarks.

(1) A. Slisenko has brought to our attention the paper [5] in which another algorithm for the same problem has been given. Ivanov claims that his algorithm runs in time $O(f(k)(n+m))$, where $f(k)$ is a function of k. $f(k)$ is described by a combination of two intricate recursive inequalities. No additional hints regarding the behavior of $f(k)$ were found in his paper. We were unable to solve these inequalities. However, we managed to show that $f(k)$ is bounded from below by 2^k for every positive integer k. It might be that $f(k)$ grows even substantially faster than 2^k. His algorithm runs faster than ours only when k is very small and m and n are almost of the same order of magnitude. In all other cases, our algorithm is faster. An even more important advantage of our algorithm is that it is simple and intuitive, while Ivanov's algorithm is very complicated (its description needed over forty journal pages).

(2) Recently, we found, in [8], an efficient algorithm for a more general problem than the one considered here. The definition of the new problem allows three kinds of differences between the text and the pattern: (a) a symbol of the text corresponds to *no* symbol of the pattern; (b) a symbol of the pattern corresponds to *no* symbol of the text; (c) a symbol of the text corresponds to a non-identical symbol of the pattern (as in the present paper). Allowing at most k such differences, the new algorithm runs in $O(m^2+k^2n)$ time and $O(m^2)$ space.

2. Analysis of the text

Our algorithm has two parts. In the first part the pattern is analyzed. The outcome of this analysis is used in the second part for analyzing the text. The next section describes the first part. The present section is devoted to the second part. We show how to use the results of the pattern analysis in order to find all occurrences of the pattern in the text with at most k mismatches.

The input to the text analysis consists of the following:

(a) The pattern: an array $A=a_1, \ldots, a_m$;

(b) The text: an array $T=t_1, \ldots, t_n$;

(c) The output of the pattern analysis: a two-dimensional array $\textit{Pat-Mismatch}[1, \ldots, m-1; 1, \ldots, 2k+1]$, where, row i of the array $(\textit{Pat-Mismatch}(i, 1), \ldots, \textit{Pat-Mismatch}(i, 2k+1))$ contains the $2k+1$ first locations in which $a_{i+1}, \ldots, a_m$ has different symbols than $a_1, \ldots, a_{m-i}$. ($\textit{Pat-Mismatch}(i, \nu) = f$ means that $a_{i+f} \neq a_f$ and f is the mismatch number ν from left to right.) If there are only $c < 2k+1$ mismatches between $a_{i+1}, \ldots, a_m$ and $a_1, \ldots, a_{m-i}$ we enter the default value $m+1$ from location $c+1$ on. That is,

$$\textit{Pat-Mismatch}(i, c+1) = \cdots = \textit{Pat-Mismatch}(i, 2k+1) = m+1.$$

The text is analyzed into the array $\textit{Text-Mismatch}[0, \ldots, n-m; 1, \ldots, k+1]$. Following the text analysis, row i of the array $(\textit{Text-Mismatch}(i, 1), \ldots, \textit{Text-Mismatch}(i, k+1))$ contains the $k+1$ first mismatches between the strings $t_{i+1}, \ldots, t_{i+m}$ and $a_1, \ldots, a_m$. ($\textit{Text-Mismatch}(i, \nu) = f$ means that $t_{i+f} \neq a_f$ and this is mismatch number ν from left to right.) If there are only $c < k+1$ mismatches between $t_{i+1}, \ldots, t_{i+m}$ and $a_1, \ldots, a_m$, then we enter the default value $m+1$ from location $c+1$ on. That is, $\textit{Text-Mismatch}(i, c+1) = \cdots = \textit{Text-Mismatch}(i, k+1) = m+1$.

Remark. This solves our problem since $\textit{Text-Mismatch}(i, k+1) = m+1$ means that there is an occurrence of the pattern which starts at t_{i+1} with at most k mismatches.

We start with a very high-level specification of the algorithm. It is explained by the verbal and illustrative descriptions that follow.

Algorithm Text-Analysis

```
Initialize: $\textit{Text-Mismatch}[0, \ldots, n-m; 1, \ldots, k+1] := m+1$;
            $r := 0; j := 0$;
        for $i := 0$ to $n-m$ do
          begin
            $b := 0$;
            if $i < j$
            then Merge($i, r, j, b$);
            if $b < k+1$
            then $r := i$; Extend($i, j, b$)
          end.
```

The **for**-loop is responsible for 'sliding' the pattern to the right one place at a time. At iteration i, we check if an occurrence of the pattern starts at t_{i+1}. Suppose that r is an iteration prior to i $(0 \leq r < i)$ that maximizes $j = r + \textit{Text-Mismatch}(r, k+1)$. Namely, j is the rightmost index of the text to which we arrived at previous iterations of the loop. Each iteration consists of calling procedure Merge (if $i < j$) and possibly procedure Extend (note that, at the beginning, $i = 0$, $j = 0$ and therefore, Merge is not invoked at the first iteration). Merge finds mismatches between $t_{i+1}, \ldots, t_j$ and $a_1, \ldots, a_{j-i}$ and reports in b the number of mismatches found. If $b \geq k+1$ we proceed to the next iteration. Otherwise, EXTEND scans the text from t_{j+1} on, till it either finds $k+1$ mismatches or till it hits t_{i+m} and finds that there is an occurrence of the pattern which starts at t_{i+1} with at most k mismatches. The situation is illustrated in Fig. 1.

Example 2.1. The text is $t_1, \ldots, t_{13} = bbabababacaacbb$, the pattern $a_1, \ldots, a_{10} = aaaaabaaab$, and $k = 4$ (as in Example 1.1). At iteration 2 we check if an occurrence of the pattern starts at t_3. Apply iterations 0 and 1 to get *TEXT-MISMATCH*$(0, 5) = 10$ and *TEXT-MISMATCH*$(1, 5) = 7$. Therefore, $r = 0$ and $j = 10$. Apply MERGE to find three mismatches between $t_3, \ldots, t_{10}$ and $a_1, \ldots, a_8$: *TEXT-MISMATCH*$(2, 1) = 2$, *TEXT-MISMATCH*$(2, 2) = 4$ and *TEXT-MISMATCH*$(2, 3) = 6$. Since $3 \leq k$, we apply EXTEND to find a single mismatch between $t_{11}t_{12}$ and a_9a_{10}: *TEXT-MISMATCH*$(2, 4) = 9$. Since only four mismatches were found between $a_1, \ldots, a_{10}$ and $t_3, \ldots, t_{12}$, we conclude that an occurrence of the pattern starts at t_3.

Let us explain the role that procedure MERGE plays at iteration i of the TEXT-ANALYSIS. In the previous paragraph we stated that MERGE finds mismatches between $t_{i+1}, \ldots, t_j$ and $a_1, \ldots, a_{j-i}$ and reports in b the number of mismatches found. That is, MERGE computes *TEXT-MISMATCH*$[i; 1, \ldots, b]$ $(b \leq k+1)$. MERGE uses two kinds of data that were computed in iterations prior to i of TEXT-ANALYSIS.

(a) The mismatches with respect to (in short, w.r.t.) $r+1$ in the text. Obviously, such mismatches which occur in locations $< i+1$ in the text are irrelevant for checking whether there is an occurrence of the pattern that starts at t_{i+1}. Let q be the smallest integer satisfying *TEXT-MISMATCH*$[r, q]$ is greater than $i - r$. Thus, MERGE uses *TEXT-MISMATCH*$[r; q, \ldots, k+1]$ (Fig. 1(b)).

(b) *PAT-MISMATCH*$[i-r; 1, \ldots, s]$, where s is the rightmost mismatch in *PAT-MISMATCH*$[i-r; 1, \ldots, 2k+1]$ such that *PAT-MISMATCH*$(i-r, s)$ is less than $(j-i+1)$ (Fig. 1(c)).

We apply a case analysis in order to understand how to use these previously computed data. We need the following two conditions for the case analysis. Consider any location x of the text, $i+1 \leq x \leq j$. We define two conditions on x.

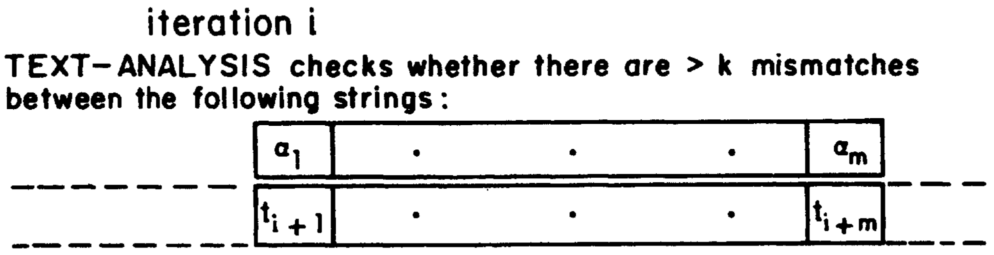
Condition 1. x falls under a mismatch w.r.t. r. That is, $t_x \neq a_{x-r}$ and for some d $(q \leq d \leq k+1)$ $x - r =$ *TEXT-MISMATCH*(r, d). (This corresponds to a mismatch between two locations, one from the bottom line and the other from the middle line in Fig. 1(d).)

Consider laying one copy of the pattern starting at t_{r+1} and another copy starting at t_{i+1} (the upper and middle lines in Fig. 1(d)).

Condition 2. x falls under a mismatch between these two copies of the pattern. That is $a_{x-r} \neq a_{x-i}$. Also, $x - i =$ *PAT-MISMATCH*$(i-r, f)$ for some f $(1 \leq f \leq s)$.

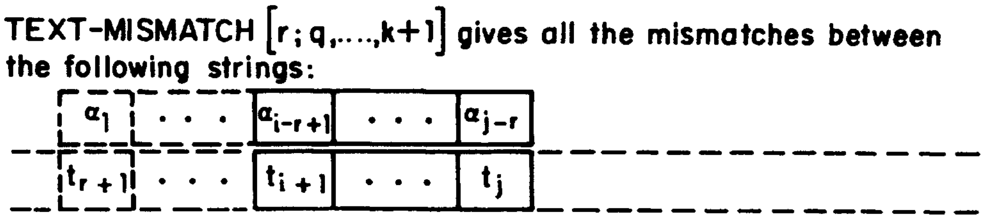
Location x may satisfy either both conditions or any one of them or none.

We are now ready to present the case analysis for any location of the text x, $i+1 \leq x \leq j$, and how it affects the question: $t_x = a_{x-i}$? (In words, does location x of the text match location $x - i$ of the pattern?)

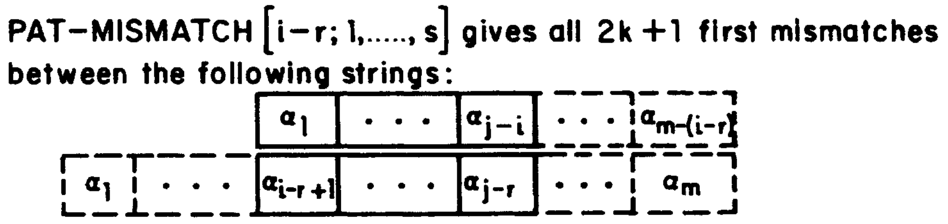

iteration i

TEXT-ANALYSIS checks whether there are > k mismatches between the following strings:

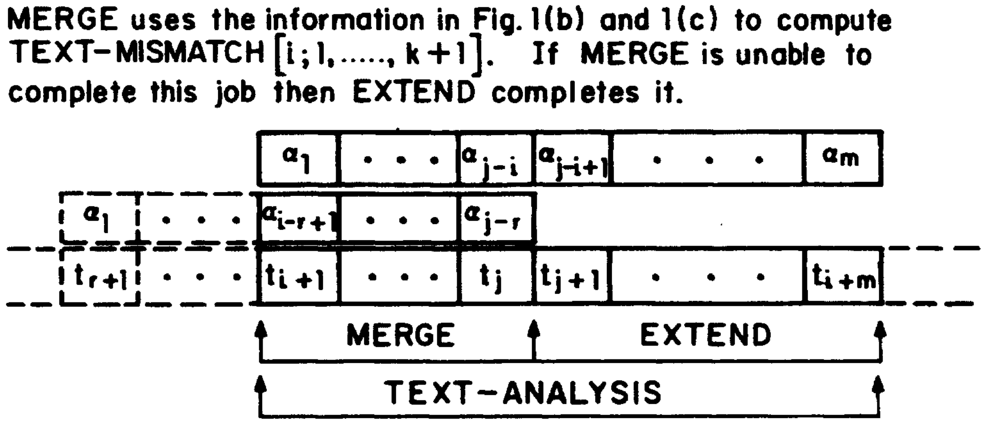
(a)


TEXT-MISMATCH $[r; q, \ldots, k+1]$ gives all the mismatches between the following strings:
α_1 . . . α_{i-r+1} . . . α_{j-r}
t_{r+1} . . . t_{i+1} . . . t_j

(b)


PAT-MISMATCH $[i-r; 1, \ldots, s]$ gives all 2k+1 first mismatches between the following strings:
α_1 . . . α_{j-i} . . . $\alpha_{m-(i-r)}$
α_1 . . . α_{i-r+1} . . . α_{j-r} . . . α_m

(c)


MERGE uses the information in Fig. 1(b) and 1(c) to compute TEXT-MISMATCH $[i; 1, \ldots, k+1]$. If MERGE is unable to complete this job then EXTEND completes it.
α_1 . . . α_{j-i} α_{j-i+1} . . . α_m
α_1 . . . α_{i-r+1} . . . α_{j-r}
t_{r+1} . . . t_{i+1} . . . t_j t_{j+1} . . . t_{i+m}
MERGE
EXTEND
TEXT-ANALYSIS

(d)

Fig. 1.

Case 0. x does not satisfy Condition 1 and x also does not satisfy Condition 2. Location x of the text must match location $x-i$ of the pattern ($t_x = a_{x-i}$) and we need not bother to compare t_x and a_{x-i} (a similar argument is used in the algorithm of [6]).

Case 1. x satisfies one of the two conditions and does not satisfy the other. Let us justify why $t_x \neq a_{x-i}$ in any of these two possibilities. If Condition 1 holds and Condition 2 does not hold, then $t_x \neq a_{x-r}$ and $a_{x-r} = a_{x-i}$. Therefore, $t_x \neq a_{x-i}$. If Condition 1 does not hold and Condition 2 holds, then $t_x = a_{x-r}$ and $a_{x-r} \neq a_{x-i}$. Therefore, $t_x \neq a_{x-i}$. So, we know that there must be a mismatch at location x and again we dispense with comparing t_x and a_{x-i}. However, we do need to increase the counter of mismatches b by one and update $\textsc{Text-Mismatch}(i, b)$.

Case 2. x satisfies both conditions. Here we are unable to reason whether $t_x = a_{x-i}$ or not. So, we compare these two symbols. If they are different, we update b and $\textsc{Text-Mismatch}(i, b)$ as in Case 1.

Example 2.2. The text, the pattern, and k are as in the previous examples. At the beginning of iteration 2, $r = 0$ and $j = 10$. Merge finds mismatches between $t_3, \ldots, t_{10}$ and $a_1, \ldots, a_8$. Since there are only three mismatches, Merge finds all of them.

pattern:			*a*	*a*	*a*	*a*	*a*	*b*	*a*	*a*
pattern:	*a*	*a*	*a*	*a*	*a*	*b*	*a*	*a*	*a*	*b*
text:	*b*	*b*	*a*	*b*	*a*	*b*	*a*	*c*	*a*	*a*
location:	1	2	3	4	5	6	7	8	9	10

For each location of the text, we specify the case it falls into:
- location 3: Case 0;
- location 4: Case 1 (Condition 1 is satisfied);
- location 5: Case 0;
- location 6: Case 1 (Condition 2 is satisfied);
- location 7: Case 0;
- location 8: Case 2 and we find that the symbols are different;
- location 9: Case 0;
- location 10: Case 2 and we find that the symbols are equal.

Specifically, procedure Merge operates as if it merges the increasing sequence of $\leq k+1$ locations

$$r + \textsc{Text-Mismatch}(r, q), \ldots, r + \textsc{Text-Mismatch}(r, k+1)$$

and the increasing sequence of $\leq 2k+1$ locations

$$i + \textsc{Pat-Mismatch}(i-r, 1), \ldots, i + \textsc{Pat-Mismatch}(i-r, s)$$

into one increasing sequence. However, instead of explicitly merging the two sequences, Merge checks whether each location satisfies Case 1 or Case 2 and treats the location according to the case analysis given above.

```
Procedure MERGE($i, r, j, b$)
  Input: (1) TEXT-MISMATCH[$r; q, \dots, k+1$]
         (2) PAT-MISMATCH[$i-r; 1, \dots, s$]
  Initialize: $d := q$; $f := 1$;
```

(* The variable d will be used in the form *TEXT-MISMATCH*(r, d). Initially it is q and then it is increased by one at a time. The variable f will be used in the form *PAT-MISMATCH*$(i-r, f)$. Initially it is 1 and then it is increased by one at a time.*)

```
        while not [Case (a) or Case (b) or Case (c)] do
```

(* We stop iterating the **while**-loop and return control to TEXT-ANALYSIS in any of the following cases:

Case (a) $b = k+1$. This means that we have already found $k+1$ mismatches with respect to i.

Case (b) $d = k+2$. When d was assigned with $k+1$, then in the middle line we were exactly over location j of the bottom line. A careful observation at the way in which d is updated in procedure MERGE reveals that the fact that d was increased to $k+2$ implies that in the middle line we must have also passed location j of the bottom line and therefore, it is time to return control to TEXT-ANALYSIS and continue the search for mismatches by procedure EXTEND.

Case (c) $i +$ *PAT-MISMATCH*$(i-r, f) > j$ and *TEXT-MISMATCH*$(r, d) = m+1$. The first conjunct means that in the upper line of Fig. 1(d) we have already passed location j of the bottom line. The second conjunct means that there was an occurrence of the pattern at t_{r+1} with $d-1$ mismatches and in the middle line of Fig. 1(d) we have also already passed the location j of the bottom line.*)

```
  begin
    if $i +$ PAT-MISMATCH$(i-r, f) > r +$ TEXT-MISMATCH$(r, d)$
    (* Case 1: Condition 1 is satisfied *)
    then
      $b := b+1$;
      TEXT-MISMATCH$(i, b) :=$ TEXT-MISMATCH$(r, d) - (i-r)$;
      $d := d+1$;
    else
      if $i +$ PAT-MISMATCH$(i-r, f) < r +$ TEXT-MISMATCH$(r, d)$
      (* Case 1: Condition 2 is satisfied *)
      then
        $b := b+1$;
        TEXT-MISMATCH$(i, b) :=$ PAT-MISMATCH$(i-r, f)$;
        $f := f+1$;
      else
        (*$i +$ PAT-MISMATCH$(i-r, f) = r +$ TEXT-MISMATCH$(r, d)$*)
        (* Case 2 *)
        if $a_{\text{PAT-MISMATCH}(i-r,f)} \neq t_{i+\text{PAT-MISMATCH}(i-r,f)}$
        then
```

```
            $b := b+1$;
            TEXT-MISMATCH$(i, b) :=$ PAT-MISMATCH$(i-r, f)$;
        $f := f+1$; $d := d+1$
    end.
```

2.1. *Correctness of Procedure* MERGE

Consider iteration i.

Claim. *If there are $\geqslant k+1$ mismatches in locations $\leqslant j$, then* MERGE *finds the first $k+1$ of them. If there are $<k+1$ mismatches in locations $\leqslant j$, then* MERGE *finds all of them.*

Proof of Claim. Condition 1 holds for $\leqslant k+1$ locations, which are $>i$ and $\leqslant j$. Let y be the number of locations in this range for which Condition 2 holds. We do not know anything about y. Suppose *PAT-MISMATCH*$(i-r, 1), \ldots,$ *PAT-MISMATCH*$(i-r, y)$ had had included all mismatches between two copies of the pattern which are $i-r$ apart. Then, by our case analysis, MERGE could have found all mismatches in the range between $i+1$ and j. But *PAT-MISMATCH*$[i-r; 1, \ldots, 2k+1]$ contains no more than $2k+1$ mismatches. We have to show that we never need more than this for the claim to hold. If *PAT-MISMATCH*$(i-r, 2k+1) \geqslant j-i$, then we have all mismatches between the two patterns for which Condition 2 holds for locations $\leqslant j$ in the text and the claim follows. The remaining case is when *PAT-MISMATCH*$(i-r, 2k+1) < j-i$. This gives $2k+1$ locations, which are $>i$ and $<j$, for which Condition 2 holds. Recall that Condition 1 holds for $\leqslant k$ locations in this range. Therefore, there are $\geqslant k+1$ locations, which are $>i$ and $<j$, for which Condition 2 holds and Condition 1 does not hold. All these locations satisfy Case 1. Therefore, they suffice to establish that there is no occurrence with $\leqslant k$ mismatches starting at t_{i+1} and the claim follows. □

Procedure EXTEND finds mismatches between $t_{j+1}, \ldots, t_{i+m}$ and $a_{j-i+1}, \ldots, a_m$, by comparing proper pairs of symbols from the pattern and the text in the naive way. EXTEND stops once it finds the $(k+1)$st mismatch. If there is an occurrence of the pattern with at most k mismatches which starts at t_{i+1}, then EXTEND stops at t_{i+m} after it finishes verifying this fact.

```
Procedure EXTEND$(i, j, b)$
            while $(b < k+1)$ and $(j-i < m)$ do
              begin
                $j := j+1$
                if $t_j \neq a_{j-i}$
                then $b := b+1$; TEXT-MISMATCH$(i, b) := j-i$;
              end.
```

2.2 Complexity

The running time of TEXT-ANALYSIS is $O(nk)$. For each iteration i ($0 \leq i \leq n-m$) the operations in TEXT-ANALYSIS excluding MERGE and EXTEND take $O(1)$ time. MERGE treats entries of the form *PAT-MISMATCH*$[j-r; 1, \ldots, 2k+1]$ (whose number is $2k+1$) and entries of the form *TEXT-MISMATCH*$[r; 1, \ldots, k+1]$ (whose number is $k+1$). Each of the operations of MERGE can be charged to one of these $3k+2$ entries in such a way that each entry is being charged by $O(1)$ operations. Therefore, MERGE requires $O(k)$ time. The total number of operations performed by EXTEND throughout all the iterations is $O(n)$ since it scans each symbol of the text at most once. So, we get in total $O(n(1+k+1)) = O(nk)$.

3. Analysis of the pattern

In this section, we describe the pattern analysis, in which *PAT-MISMATCH*$[1, \ldots, m-1; 1, \ldots, 2k+1]$ is computed.

Let $[1, \ldots, m-1]$ be the set of $m-1$ rows of *PAT-MISMATCH*. Assume, w.l.g., that m is some power of 2. The algorithm uses a partition of this set into $\log_2 m$ sets as follows

$$[1], [2, 3], [4, 5, 6, 7], [8, \ldots, 15], \ldots, [\tfrac{1}{2}m, \ldots, m-1].$$

The pattern analysis has $\log m$ stages:

Stage l, $1 \leq l \leq \log m$. Compute *PAT-MISMATCH* for the rows of set l (where set l, $1 \leq l \leq \log m$, is $[2^{l-1}, \ldots, 2^l - 1]$).

Essentially, in the computation of each stage we apply the text analysis algorithm of the previous section with one exception. For set l we find up to the minimum between $2^{\log m - l}2k+1$ (instead of $k+1$) and $m-2^l$ (implied by the length of the pattern) mismatches. We do not elaborate on stage $\log m$, where we actually find up to $2k+1$ mismatches. In order to keep this presentation short, we overview the similarities to the text analysis and elaborate only on the afore-mentioned exception.

The input to stage l ($1 \leq l \leq \log m$) of the pattern analysis consists of the following:

(a) the array $a_1, \ldots, a_{m-2^{l-1}}$, which plays the role of the pattern (in the text analysis);

(b) the array $a_{2^{l-1}+1}, \ldots, a_m$, which plays the role of the text;

(c) the two-dimensional array

$$\textit{PAT-MISMATCH}[1, \ldots, 2^{l-1}-1; 1, \ldots, \min(2^{\log m - l}4k+1, m-2^{l-1})],$$

which is the output of the previous $l-1$ stages of the pattern analysis.

The output of stage $\log l$ is

$$\textit{PAT-MISMATCH}[2^{l-1}, \ldots, 2^l - 1; 1, \ldots, \min(2^{\log m - l}2k+1, m-2^l)].$$

Below, we give a very high-level specification of stage l of the pattern analysis.

```
Initialize: PAT-MISMATCH[$2^{l-1}, \ldots, 2^l - 1; 1, \ldots, \min(2^{\log m - l}2k+1, m-2^l)$]
            $:= m+1$; $r := 2^{l-1}$; $j := 2^{l-1}$;
        for $i := 2^{l-1}$ to $2^l - 1$ do
          begin
            $b := 0$;
            if $i < j$
            then MERGE($i, r, j, b$);
            if $b < \min(2^{\log m - l}2k+1, m-2^l)$
            then $r := 1$; EXTEND($i, j, b$)
          end.
```

One important difference with respect to the text analysis needs to be emphasized: In TEXT-ANALYSIS we were after the $k+1$ first mismatches for each location of the text, while here, in stage l, we want to find the $\min(2^{\log m - l}2k+1, m-2^l)$ first mismatches. The correctness proof of iteration i of procedure MERGE, in the previous section, needed the first $2k+1$ locations for which Condition 2 holds in order to find the first $k+1$ mismatches. A careful look at the proof reveals that the number of locations for which Condition 2 holds must be at least two times more than the number of mismatches we look for. So, in stage $\log m$, we look for $2k+1$ mismatches, and therefore, we need $4k+1$ locations for which Condition 2 holds. In stage l we look for $\min(2^{\log m - l}2k+1, m-2^l)$ mismatches and we need $\min(2^{\log m - l}4k+1, m-2^{l-1})$ locations for which Condition 2 holds.

3.1. Complexity

We have $\log m$ stages. Let us focus on stage l $(1 \leq l \leq \log m)$. Within such stage l we further focus on iteration i, $2^{l-1} \leq i \leq 2^l - 1$. For each such iteration:

(1) the operations in the 'main program' excluding MERGE and EXTEND take O(1) time;

(2) as in the previous section MERGE takes O('number of mismatches we look for') time. That is, $O(\min(2^{\log m - l}2k+1, m-2^l)) \leq O(2k2^{\log m - l})$ time.

The total number of operations performed by EXTEND throughout all iterations of stage l is $O(m)$. Stage l has 2^{l-1} iterations, therefore it takes $O(m + 2^l(2k2^{\log m - l})) = O(km)$ time. Since we have $\log m$ stages, the running time of the pattern analysis is $O(\sum_{l=1}^{\log m} km) = O(km \log m)$.

Acknowledgment

The article of Ivanov [5] is written in Russian, a language which is unknown to us. Slisenko's kind help in identifying and deciphering the main points in this paper is gratefully acknowledged. Also, 'merci' to the editor and the unknown referee.

References

[1] A.V. Aho, J.E. Hopcroft and J.D. Ullman, *The Design and Analysis of Computer Algorithms* (Addison-Wesley, Reading, MA, 1974).

[2] R.S. Boyer and J.S. Moore, A fast string searching algorithm, *Comm. ACM* **20** (1977) 762–772.

[3] Z. Galil, Optimal parallel algorithms for string matching, *Proc. 16th ACM Symp. on Theory of Computing* (1984) 240–248.

[4] Z. Galil and J.I. Seiferas, Time-space-optimal string matching, *J. Comput. System Sci.* **26** (1983) 280–294.

[5] A.G. Ivanov, Distinguishing an approximative word's inclusion on Turing machine in real time, *Izv. Akad. Nauk SSSR Ser. Mat.* **48** (1984) 520–568 (in Russian).

[6] D.E. Knuth, J.H. Morris and V.R. Pratt, Fast pattern matching in strings, *SIAM J. Comput.* **6** (1977) 322–350.

[7] R.M. Karp and M.O. Rabin, Efficient randomized pattern-matching algorithms, Unpublished manuscript, 1980.

[8] G.M. Landau and U. Vishkin, Efficient string matching in the present of errors, *Proc. 26th IEEE Symp. on Foundations of Computer Science* (1985) 126–136.

[9] U. Vishkin, Optimal parallel pattern matching in strings, *Proc. 12th ICALP*, Lecture Notes in Computer Science **194** (Springer, Berlin, 1985) 497–508; also, *Inform and Control*, to appear.