EFFICIENT STRING MATCHING IN THE PRESENCE OF ERRORS

Gad M. Landau

Department of Computer Science
School of Mathematical Sciences
Tel Aviv University
Tel Aviv 69978, Israel

Uzi Vishkin

Department of Computer Science
Courant Institute of Mathematical Sciences
New York University
and
Department of Computer Science
School of Mathematical Sciences
Tel Aviv University
Tel Aviv 69978, Israel

ABSTRACT

Consider the string matching problem where differences between characters of the pattern and characters of the text are allowed. Each difference is due to either a mismatch between a character of the text and a character of the pattern or a superfluous character in the text or a superfluous character in the pattern. 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 differences. The algorithm runs in $O(m^2 + k^2n)$ time. Given the same input we also present an algorithm for finding all occurrences of the pattern in the text, each with at most k mismatches (superfluous characters in either the text or the pattern are not allowed). This algorithm runs in $O(k(m\log m + n))$ time.

I. INTRODUCTION

In the known problem of pattern matching in strings (e.g., as discussed in [KMP]) we are interested in finding all occurrences of the pattern (of length m) in the text (of length n). In the present paper we are interested in finding all such occurrences with at most k differences, where k is a non-negative integer.

Example. Let the text be *abcdefghi* , the pattern *bxdyegh* and $k=3$. Let us see whether there is an occurrence with $\leq k$ differences that starts at the second location of the text. For this we propose the following correspondence between *bcdefghi* and *bxdyegh*. 1. *b* (of the text) corresponds to *b* (of the pattern).2. *c* to *x*. 3. *d* to *d*. 4. Nothing to *y*. 5. *e* to *e*. 6. *f* to nothing.7. *g* to *g*. 8. *h* to *h*. The correspondence can be illustrated as

```
b x d y e   g h
b c d   e f g h i
```

In only three places the correspondence is between non-equal characters, implying that there is an occurrence of the pattern at the second location of the text with 3 differences as required.

We distinguish three types of differences. (a) A character of the pattern corresponds to a different character of the text. (Item 2 in the Example). In this case we say that there is a *mismatch* between the two characters. (b) A character of the pattern corresponds to "no character" in the text. (Item 4). (c) A character of the text corresponds to "no character" in the pattern. (Item 6).

We consider two problems. The above input applies to both of them.

The *string matching with k mismatches* problem. (In short, the *k mismatches* problem). Find all occurrences of the pattern in the text with at most k differences of type (a).

The main problem. The *string matching with k differences* problem. (In short, the *k differences* problem). Find all occurrences of the pattern in the text with at most k differences of type (a),(b) and (c).

This research was supported by NSF grant NSF-DCR-8318874 and NSF-DCR-8413359, ONR grant N0014-85-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.

Both problems have a strong pragmatical flavor. In practice, we often need to analyze situations where the data is 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. The errors may include a character being replaced by another character, a character being omitted or a superfluous character being inserted. Assuming some bound on the number of errors would clearly imply our problems. Applications of our solution for the k differences problem in Molecular Biology are discussed in [LVN]. [SK] give a comprehensive review of applications of the k differences problem.

For the k mismatches problem we give a new algorithm which runs in time $O(k(m\log m + n))$. For the k differences problem we give a new algorithm which runs in time $O(m^2 + k^2 n)$. Both algorithms are designed for a random-access-machine (RAM) [AHU].

Both our algorithms consist of a pattern analysis part to be followed by a text analysis. For the k mismatches problem we have actually achieved $O(km\log m)$ time for the pattern analysis and $O(kn)$ time for the text analysis.

For the k differences problem we have achieved $O(m^2)$ time for the pattern analysis and $O(k^2 n)$ time for the text analysis.

Therefore, whenever the time for the text analysis dominates the computation time, the running time of our algorithms are close to optimal. There are a few realistic possibilities where this happens:

1. The same pattern has to be matched with different texts or the pattern is known in advance and we have plenty of time to analyze it.
2. m is sufficiently small with respect to n ($m = o(n/\log n)$ in the k mismatches algorithm and $m = o(k\sqrt{n})$ in the k differences algorithm).

Perhaps surprisingly we were able to adopt the conservative and simple framework of [KMP] in both algorithms. That is, we first build a table based on analysis of the pattern. Then, we examine the text from left to right checking possible occurrences with respect to one starting location (in the text) at each iteration. Besides the tables built in the pattern analysis, the input to each iteration consists of the knowledge acquired in previous iterations. The rightmost location in the text to which we arrived in a previous iteration is of particular significance. Each iteration consists of manipulating this knowledge. And if necessary (till this rightmost location we have not sufficient evidence to exclude the possibility of occurrence), we proceed to investigate the text to the right of this rightmost location.

We use a further analogy to the [KMP] algorithm for the known string matching problem of [KMP] in order to explain our contribution in the algorithm for the k differences problem. Consider the following most trivial *strings equality* problem: Given two strings, find if they are identical. Observing this problem and its immediate solution was clearly a step (even if it was very minor) in devising string matching algorithms. Our presentation is strongly motivated by this analogy. It has two major steps.

In the first step we define an auxiliary problem which is analogous to the string equality problem when the k differences problem is considered (instead of the string matching problem). The solution of the auxiliary problem uses known techniques ([U]). Our contribution is providing the second major step. That is, we give an algorithm for the k differences problem using the solution for the first step. The auxiliary problem of the first major step is less obvious than the strings equality problem and provides an essential part of our algorithm. However, we feel that our contribution which is analogous to the whole algorithm of [KMP] overcomes a more involved and general problem than in [KMP].

For the original string matching problem and the k mismatches problem, the definition of the problems imply immediately algorithms which run in $O(mn)$ time. [S] gave a simple $O(nm)$ time algorithm for the k differences problem. In his survey on future directions for research in string matching, Z. Galil [G84b] discusses the k mismatches problem, which seems substantially easier than our main problem. He states that it is an open question whether the naive algorithm for the k mismatches problem which takes $O(mn)$ time can be improved. Note that even our algorithm for the main problem is much faster than $O(nm)$. We already noted that the case $k = 0$ (in both problems) is the extensively studied string matching problem. There are a few notable algorithms for the string matching problem: linear time serial algorithms - [BM], [GS], [KMP], [KR] (a randomized algorithm) and [V], parallel algorithms - [G84a] and [V]. Note that none of these algorithms is suitable to cope with our problems.

Postscript. After all the results with respect to the k mismatches problem have been achieved, A. Slisenko has brought to our attention the paper [I] in which another algorithm for the k mismatches 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 40 journal pages).

After all the results with respect to the k differences problem have been achieved we became aware of [U85]. It presents an interesting algorithm the k differences problem. The algorithm runs in time $O(m|\Sigma|G+n)$ and requires $O((|\Sigma|+m)G)$ space, where $|\Sigma|$ is the size of the alphabet and $G=min(3^m, 2^k|\Sigma|^k m^{k+1})$. Preprocessing of the pattern takes $O(m|\Sigma|G)$ time. Then analysis of the text takes $O(n)$ time which is pretty impressive. However, the author himself seems to be aware that the space and preprocessing time requirements make the algorithm impractical, in general. (For comparison, the space requirement of our algorithm for the k differences problem is $O(m^2)$).

Section 2 gives an exceedingly simple algorithm for string matching with one mismatch. Section 3 presents the algorithm for the k mismatches problem (this section is a preliminary version of [LV1]). In Section 4 the algorithm for the k differences problem is given (this section is a preliminary version of [LV2]).

II. STRING MATCHING WITH 1 MISMATCH

Below, we describe a simple idea for finding all occurrences of the pattern in the text with at most one mismatch. [ML] showed how to apply the [KMP] algorithm for the following slightly modified string matching problem. Find for each location of the text whether an occurrence of the pattern starts at it. For each location in which there is no occurrence of the pattern find the leftmost character in which there is a mismatch.

The algorithm has three steps.

1. Run this modified string matching algorithm on the input text and pattern.
2. Inverse the input text (if it was $t_1,\ldots,t_n$ let it be $t_n,t_{n-1},\ldots,t_1$) and the input pattern and run the same algorithm. Clearly, it will find for each location of the text, in which there is no occurrence of the pattern, the rightmost character in which there is a mismatch.
3. For all t_{i+1} such that the leftmost mismatched character for the pattern starting at t_{i+1} (found in step 1), falls at the same place as the rightmost mismatched character for the reversed pattern starting at t_{i+m} (found in step 2), conclude that there is an occurrence of the pattern starting at t_{i+1} with at most one mismatch.

Important remark. We leave it to the interested reader to find how to generalize this algorithm to for the string matching problem with one difference or even for a wider definition of this problem where one "successive chunk of differences" is allowed. We slightly elaborate on this in the final version of the paper.

III. STRING MATCHING WITH k MISMATCHES

The algorithm has two parts:

(a) The pattern analysis. We build a table which is based on analysis of the pattern.

(b) The text analysis. We show how to find efficiently all the occurrences of the pattern in the text with at most k mismatches, using the result of the pattern analysis.

We first describe the text analysis and afterwards the pattern analysis is described.

Analysis of the text.

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 $PAT-MISMATCH[1,\ldots,m-1;1,\ldots,2k+1]$.

Where, row i of the array ($PAT-MISMATCH(i,1)$,..., $PAT-MISMATCH(i,2k+1)$), contains the $2k+1$ first locations in which $a_{i+1},\ldots,a_m$ has different symbols than $a_1,\ldots,a_{m-i}$. ($PAT-MISMATCH(i,v)=f$ means that $a_{i+f}\neq a_f$ and f is the mismatch number v 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, $PAT-MISMATCH(i,c+1)=,\ldots,PAT-MISMATCH(i,2k+1)=m+1$.

The text is analyzed into the array $TEXT-MISMATCH[0,\ldots,n-m;1,\ldots,k+1]$. Following the text analysis, row i of the array ($TEXT-MISMATCH(i,1),\ldots,TEXT-MISMATCH(i,k+1)$), contains the $k+1$ first mismatches between the strings $t_{i+1},\ldots,t_{i+m}$ and $a_1,\ldots,a_m$. ($TEXT-MISMATCH(i,v) = f$ means that $t_{i+f} \neq a_f$ and this is mismatch number v 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, $TEXT-MISMATCH(i,c+1) = \ldots = TEXT-MISMATCH(i,k+1) = m+1$.

Remark. This solves our problem since $TEXT-MISMATCH(i,k+1) = m+1$ 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.

TEXT-ANALYSIS

Initialize: $r:=0; j:=0;$
$TEXT-MISMATCH[0,\ldots,n-m;1,\ldots,k+1] := m+1;$
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 $j = r + TEXT-MISMATCH(r,k+1)$. 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 Figure 1.

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 correspond 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 ready now 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?)

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 [KMP]).

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 $TEXT-MISMATCH(i,b)$.

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 *TEXT-MISMATCH(i,b)* as in Case 1.

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

$r + TEXT\text{-}MISMATCH(r,q), \ldots, r+TEXT\text{-}MISMATCH(r,k+1)$

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

$i + PAT\text{-}MISMATCH(i-r,1), \ldots, i+PAT\text{-}MISMATCH(i-r,s)$

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) $TEXT-MISMATCH[r;q,\ldots,k+1]$
2) $PAT-MISMATCH[i-r;1,\ldots,s]$

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 were 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_{PAT-MISMATCH(i-r,f)} ≠
            t_{i+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
```

Correctness of procedure MERGE. Consider iteration i.

Claim. If there are $\geq k+1$ mismatches in locations $\leq j$ then MERGE finds the first $k+1$ of them. If there are $<k+1$ mismatches in locations $\leq j$ then MERGE finds all of them.

Proof of claim. Condition 1 holds for $\leq k+1$ locations, which are $>i$ and $\leq j$. Let y be the number of locations in this range for which Condition 2 holds. We do not know anything about y. Suppose $PAT\text{-}MISMATCH(i-r,1), \ldots, PAT\text{-}MISMATCH(i-r,y)$ 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 $PAT-MISMATCH(i-r,2k+1) \geq j-i$ then we have all mismatches between the two patterns for which Condition 2 holds for locations $\leq j$ 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 hold for $\leq k$ locations in this range. Therefore, there are $\geq k+1$ 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 $\leq k$ 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 ≠ a_{j-i}
        then
            b:=b + 1 ;
            TEXT-MISMATCH[i,b]:= j-i ;
    end
```

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 $O(n(1+k+1)) = O(nk)$.

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: $[1],[2,3],[4,5,6,7],[8\ldots15],\ldots,[m/2,\ldots,m-1]$.

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 $[2^{l-1},\ldots,2^l-1]$.)

We describe in more detail the last stage (stage $\log m$) and discuss briefly later how to extend the same technique for the earlier stages. Essentially, we apply the text analysis algorithm of the previous section. In order to keep this presentation short, we overview the similarities to the text analysis and elaborate only on the differences.

The input to stage $\log m$ of the pattern analysis consists of the following:

a) The array $a_1, \ldots, a_{m/2}$, which plays the role of the pattern (in the text analysis).

b) The array $a_{m/2+1}, \ldots, a_m$, which plays the role of the text.

c) The two dimensional array $PAT-MISMATCH[1,\ldots,m/2-1;1,\ldots,4k+1]$, which is the output of the previous $\log m - 1$ stages of the pattern analysis.

The output of stage $\log m$ is $PAT-MISMATCH[m/2,\ldots,m-1;1,\ldots,2k+1]$.

Below, we give a very high-level specification of stage $\log m$ of the pattern analysis.

```
Initialize: r:=m/2 ; j:=m/2;
            PAT-MISMATCH[m/2,...,m-1;
            1,...,2k+1] := m+1 ;
for i:=m/2 to m-1  do
    begin
        b:=0;
        if i < j
        then MERGE(i,r,j,b);
        if b<2k + 1
        then  r := i ; 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 we want to find the $2k+1$ 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 these first $k+1$ mismatches. A careful check of the proof will show that the first $4k+1$ locations were Condition 2 does not hold would have sufficed for finding the first $k+1$ mismatches, as required here. This explains item c) in the input for

stage $\log m$.

Next, we describe briefly stage l, $(1 \le l < \log m)$, by emphasizing the differences with respect to stage $\log m$ which was described above.
The input to stage l 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
$PAT-MISMATCH[1,\ldots,2^{l-1}-1;1,\ldots,$
$\min(2^{\log m - l}4k+1, m-2^{l-1})]$, which is the output of the previous $l-1$ stages of the pattern analysis.

The output of iteration i at stage l $(2^{l-1} \le i \le 2^l - 1)$ is $PAT-MISMATCH[i;1,\ldots,$ $\min(2^{\log m - l}2k+1, m-i)]$
We note three differences in this stage, with respect to stage $\log m$:

a) At stage l the *for* loop is *for* $i:=2^{l-1}$ *to* 2^l-1. At each iteration i, we look for the mismatches between $a_{i+1}, \ldots, a_m$ and $a_1, \ldots, a_{m-i}$, $(2^{l-1} \le i \le 2^l-1)$.
b) Iteration i of stage l looks for $(\min(2k2^{\log m - l}+1, m-i))$ mismatches.
c) The output of stages *1,...,l-1* must give the first $(\min(4k2^{\log m - l}+1, m-2^{l-1}))$ mismatches.

Complexity. For each iteration i at stage l $(2^{l-1} \le i \le 2^l - 1)$, $(1 \le l \le \log m)$ the operations in the "main program" excluding MERGE and EXTEND take $O(1)$ time. As in the previous section MERGE requires O("number of mismatches we look for ") time. Here it means $O(2k2^{\log m - l})$ 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. We have $\log m$ stages. So, the running time of the pattern analysis is $O(\sum_{l=1}^{\log m} km) = O(km \log m)$.

IV. STRING MATCHING WITH k DIFFERENCES.

The algorithm has two parts:
(a) The pattern analysis. We build a table which is based on analysis of the pattern.
(b) The text analysis. We show how to find efficiently all the occurrences of the pattern in the text with at most k differences, using the result of the pattern analysis.

Analysis of the text.
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
$MAX-LENGTH[0,\ldots,m-1;0,\ldots,m-1]$.
$MAX-LENGTH(i,j)=f$ means that $a_{i+1}, \ldots, a_{i+f} = a_{j+1}, \ldots, a_{j+f}$, and $a_{i+f+1} \neq a_{j+f+1}$. In words, consider laying the suffix of the pattern starting at a_{i+1} over the suffix of the pattern starting at a_{j+1}. $MAX-LENGTH(i,j)$ is the longest match of prefixes between these two suffixes.

Output of the text analysis: All occurrences with $\le k$ differences of the pattern in the text.
The description of the text analysis uses a known solution to an auxiliary problem. The relation between the auxiliary problem and the text analysis is discussed briefly in the introduction. Since we wanted this presentation to be self contained we describe this solution.

The auxiliary problem:
Input. Two strings: $A = a_1, \ldots, a_m$ and $B = b_1, \ldots, b_{m+k}$. We want to find whether an occurrence of A with at most k differences starts at b_1. We first show an $O(m^2)$ time algorithm for the auxiliary problem. Later we show how to derive from it an $O(km)$ time algorithm.

$O(m^2)$ time algorithm for the auxiliary problem. We use a matrix $D_{[0,\ldots,m;0,\ldots,m+k]}$, where $D_{i,l}$ is the number of differences between $a_1, \ldots, a_i$ and $b_1, \ldots, b_l$.
It should be obvious that if $D_{m,l} \le k$, for at least one l, $m-k \le l \le m+k$, then the answer to the auxiliary problem is yes.

The following algorithm computes the matrix $D_{[0,\ldots,m;0,\ldots,m+k]}$
Initialization $D_{0,0} := 0$.
for all l, $1 \le l \le m+k$, $D_{0,l} := l$.
for all i, $1 \le i \le m$, $D_{i,0} := i$,.
for $i:=1$ *to* m *do*
  for $l:=1$ *to* $m+k$ *do*
    $D_{i,l} :=$ min $(D_{i-1,l}+1, D_{i,l-1}+1, D_{i-1,l-1}$ if $a_i = b_l$ or $D_{i-1,l-1}+1$ otherwise).
    ($D_{i,l}$ is the minimum of three numbers. These numbers are obtained from the predecessors of $D_{i,l}$ on its column, row and diagonal, respectively).

This algorithm clearly runs in $O(m^2)$ time.

$O(km)$ time algorithm for the auxiliary problem. (Due to [U]). *Diagonal* d of the matrix consists of all $D_{i,l}$'s such that $l-i = d$.
Lemma 1 [U]. For every i,l, $D_{i,l} - D_{i-1,l-1}$ is either zero or one.

Lemma 1 implies that we can store the information of the matrix in a more compact way.
For a number of differences e and a diagonal d, let $L_{d,e}$ denote the largest row i such that $D_{i,l} = e$ and $D_{i,l}$ is on diagonal d. Note, that this implies that there are e differences between $a_1, \ldots, a_{L_{d,e}}$ and $b_1, \ldots, b_{L_{d,e}+d}$, and $a_{L_{d,e}+1} \neq b_{L_{d,e}+d+1}$.

Corollary. For our auxiliary problem we need only values of $L_{d,e}$, where e and d satisfy $e \leq k$ and $|d| \leq e$.

Proof. $e \leq k$ is obvious. The initial values of the matrix and Lemma 1 imply that all the $D_{i,l}$ on a diagonal d are $\geq |d|$ and therefore given a number of differences e we need only values of $L_{d,e}$ where $|d| \leq e$.
The answer to the auxiliary problem is yes if one of the $L_{d,e}$, $(|d| \leq e \leq k)$, equals m.
Given d and e we compute $L_{d,e}$ using its definition. That is, $L_{d,e}$ is the largest row such that $D_{i,l} = e$, and $D_{i,l}$ is on the diagonal d. In the above $O(m^2)$ time algorithm the assignment of e into $D_{i,l}$ was done using one (or more) of the following data:
(a) $D_{i-1,l-1}$ (the predecessor of $D_{i,l}$ on the diagonal d) is $e-1$ and $a_i \neq b_l$.
(b) $D_{i,l-1}$ (the predecessor of $D_{i,l}$ on row i which is also on the diagonal "below" d) is $e-1$.
(c) $D_{i-1,l}$ (the predecessor of $D_{i,l}$ on column l which is also on the diagonal "above" d) is $e-1$.
(d) $D_{i-1,l-1}$ is also e and $a_i = b_l$.
This implies that we can start from $D_{i,l}$ and follow its predecessors on diagonal d by possibility (d) till the first time one (or more) of possibilities (a) (b) and (c) occur.
The $O(km)$ time algorithm for the auxiliary problem is given below. The reader is invited to convince oneself that the initialization step (Instruction 1) is done in a way which enables the computation of the $L_{d,e}$'s in the subsequent instructions. Instructions 2-6 "inverse" this description, in order to compute $L_{d,e}$'s $(|d| \leq e \leq k)$. $L_{d,e-1}$, $L_{d-1,e-1}$, and $L_{d+1,e-1}$ are used to initialize the variable *row* (Instruction 3), which is then increased by one at a time till it hits the correct value of $L_{d,e}$ (Instruction 4).

The $O(kn)$ time algorithm for the auxiliary problem

```
[1] Initialization
    for d:=-(k+1) to (k+1) do
        L_{d,|d|-2} := -∞;
        if d < 0
            then L_{d,|d|-1} := |d| - 1 ;
            else L_{d,|d|-1} := -1 ;
[2] for e:=0 to k do
        for d:=-e to e do
            [3] row := max[(L_{d,e-1}+1), (L_{d-1,e-1}),
                    (L_{d+1,e-1}+1)].
                [4] while a_{row+1} = b_{row+1+d} do
                        row:=row+1.
                [5] L_{d,e} := row.
                [6] if L_{d,e} = m
                        then print *YES* and stop.
```

Complexity. The algorithm computes $L_{d,e}$ for $2k+1$ diagonals. For each diagonal variable *row* can get at most m different values. Therefore, the computation takes $O(km)$ time.

Overview of the text analysis. Let us go back to the text analysis. The text analysis consists of $n-m+k$ iterations. At iteration i we check if an occurrence with $\leq k$ differences of the pattern starts at t_{i+1}. Let t_j be the rightmost symbol in the text that was reached at an iteration prior to i. Assume, w.l.g., that we reached t_j for the first time at iteration r, $0 \leq r < i$.

Example 1. Let $t_{17}, \ldots, t_{30}$ be *abaaacddacdcab*, and let $a_1, \ldots, a_{13}$ be *aaaaeddcdcbab* (here $r=16$, $j=30$). $k = 4$. The correspondence

a a a a e d d c d c b a b
a b a a a c d d a c d c a b

gives k differences. It can be easily checked that a correspondence with less differences is impossible.

The definition of the text analysis (given later) implies that there are at most $k+1$ differences between $t_{r+1},\ldots,t_j$ and some prefix of the pattern. Hence, there are also at most $k+1$ differences between some suffix of this prefix of the pattern and $t_{i+1},\ldots,t_j$. We call this suffix of prefix of the pattern the *subpattern*. In the example let i be 20 and the subpattern will be $a_4, \ldots, a_{13}$. This means that for some correspondence between symbols of $t_{i+1}, \ldots, t_j$ and the subpattern there are at least $j-i-k$ symbols of $t_{i+1}, \ldots, t_j$ that have a match in the subpattern. It is easy to see that all the symbols of the text that have successive matches in this correspondence form at most $k+1$ (successive) substrings in $t_{i+1},\ldots,t_j$. For each such substring we know its corresponding substring in the pattern. Suppose a substring of the text $t_{p+1}, \ldots, t_{p+f}$ matches a substring of the pattern $a_{c+1}, \ldots, a_{c+f}$ and $t_{p+f+1} \neq a_{c+f+1}$, we denote this by the *triple* (p,c,f). There are at most k symbols in $t_{i+1}, \ldots, t_j$ which do not have matching symbols in the subpattern. We denote each such symbol t_{h+1} by the triple $(h,0,0)$. We denote the sequence of this triples by $S_{i,j}$.

In example 1 $S_{20,30}$ is (20,3,1), (21,0,0), (22,5,2), (24,0,0), (25,7,3), (28,11,2).

Recall that at iteration i we want to find if $a_1, \ldots, a_m$ occurs at $t_{i+1}, \ldots$. This seems similar to the auxiliary problem. However, here we have more information: the sequence $S_{i,j}$ and *MAX−LENGTH*. Iteration i uses this information.

Iteration i.

Iteration i consists of the $O(km)$ time algorithm of the auxiliary problem with the following modification in Instruction 4 . Instruction 4 increases the variable *row* by one at a time. The sequence $S_{i,j}$ and *MAX−LENGTH* enables to increase *row* by much larger jumps as long as we do not require information about symbols of the text, which are beyond t_j. Once *row* takes us beyond t_j (i.e., $i + row + 1 + d > j$), $S_{i,j}$ and *MAX−LENGTH* do not help us any more and we apply (the old) Instruction 4 as in the computation of the auxiliary problem.

We finish this overview of iteration i by showing how to apply the sequence $S_{i,j}$ and *MAX−LENGTH* to obtain these jumps. The *while* loop of Instruction 4 looks for the longest match between prefixes of some suffix of the text $t_{i+row+d+1}, \ldots$, where $i+1 \le i+row+d \le j$ and some suffix of the pattern $a_{row+1}, \ldots$. We explain how to find the maximum w such that $a_{row+1}, \ldots, a_{row+w}$ equals $t_{i+row+d+1}, \ldots, t_{i+row+d+w}$. Suppose that according to $S_{i,j}$ the substring $t_{i+row+d+1}, \ldots, t_{i+row+d+f}$ matches $a_{c+1}, \ldots, a_{c+f}$ for some index c of the pattern and $c+f$ is the maximal index of the pattern for which this match holds. We can find those c and f using the fact that for each t_{h+1} $(i \le h < j)$ there exists a triple (p_1, c_1, f_1) in $S_{i,j}$ such that $p_1 \le h \le p_1 + f_1$. ((p_1, c_1, f_1) *covers* t_{h+1}). For the computation of w we need to break into the following cases:

Case (*a*). $f \ge 1$. *MAX−LENGTH*(c, row) gives the maximal number g such that $a_{c+1}, \ldots, a_{c+g}$ equals $a_{row+1}, \ldots, a_{row+g}$. Case (a) has two subcases.

Case (*a* 1). $f \ne g$. It is easy to see that here $t_{i+row+d+1}, \ldots, t_{i+row+d+min(f,g)} = a_{row+1}, \ldots, a_{row+min(f,g)}$ and $t_{i+row+d+min(f,g)+1} \ne a_{row+min(f,g)+1}$. Therefore, we assign $w := min(f, g)$.

Case (*a* 2). $f = g$. This implies $t_{i+row+d+1}, \ldots, t_{i+row+d+f} = a_{row+1}, \ldots, a_{row+f}$ but does not reveal whether $t_{i+row+d+f+1}$ equals $a_{row+f+1}$ or not. Therefore, we assign $row := row+f$ and apply again the present case analysis accumulating this "jump" over f symbols into w.

Case (*b*). $f = 0$. Case (b) has two subcases.

Case (*b* 1). $t_{i+row+d+1} \ne a_{row+1}$. Hence, we assign $w := 0$.

Case (*b* 2). $t_{i+row+d+1} = a_{row+1}$. Therefore, we assign $row := row+1$, and we apply again the present case analysis accumulating this propagation of 1 into w.

The text analysis algorithm

```
j:=0;
for i:=0 to n-m+k do
begin
    [1] Proper initialization (as in the O(km)
    algorithm for the auxiliary problem)
    [2] for e:=0 to k do
        for d:=-e to e do
            [3] row := max[(L_{d,e-1}+1),
                (L_{d-1,e-1}), (L_{d+1,e-1}+1)].
            [4.new] while i+row+d+1 ≤ j do
                [4.new.1] take from S_{i,j} the triple
                that "covers" t_{i+row+d+1}.
                Derive from this triple the
                indices c,f such that
                t_{i+row+d+1}, ..., t_{i+row+d+f}
                = a_{c+1}, ..., a_{c+f}
                (t_{i+row+d+f+1} ≠ a_{c+f+1})
                [4.new.2] if f ≥ 1
                then (* case a *)
                    [4.new.3] if f ≠
                        MAX−LENGTH(c,row)
                    then (* case a1 *)
                        row := row + min(f,
                        MAX−LENGTH(c,row))
                        go to 5
                    else (* case a2 *)
                        row:= row + f ;
                else (* case b *)
                    [4.new.4] if t_{i+row+d+1} ≠
                    a_{row+1}
                    then (* case b1 *)
                        go to 5
                    else (* case b2 *)
                        row:= row+1
            od
            [4.old] while a_{row+1}=t_{i+row+1+d} do
                    row:=row+1.
            [5] L_{d,e} := row.
            [6] if L_{d,e} = m
             then print *YES* and go to 7
        od
    [7] If new symbols of the text were reached
    (j was increased) then starting from the
    L_{d,k} which implies the new j
    (j=L_{d,k} + d + i) we reconstruct the new
    S_{i,j}.
end
```

Implementation remarks.

Instruction 4.new.1: When we compute $L_{0,0}$ we start searching for the indices (f, c) at the first triple of $S_{i,j}$. We know which triple was checked, when any $L_{d,e}$ gets its value. So, when computing a new $L_{d,e}$ we know what were the last triples we checked in the computation of

each one of $L_{d-1,e-1}, L_{d,e-1}, L_{d+1,e-1}$. At Instruction 3 *row* got its initial value from the maximum of $L_{d-1,e-1}, L_{d,e-1}+1, L_{d+1,e-1}+1$. The last triple that was checked in the computation of the one which gives this maximum is the first to be checked in the computation of $L_{d,e}$.

Instruction 7: At the end of each iteration i if at least one new symbol of the text was reached we have to create a new sequence of triples instead of $S_{r,j}$. If $t_{\bar{j}}$ is the rightmost symbol of the text which was reached in such an iteration then denote the new sequence $S_{i,\bar{j}}$. In order to compute $S_{i,\bar{j}}$ we hold an sequence (of triples) for each $L_{d,e}$ during its computation at iteration i. This sequence "realizes" $L_{d,e}$. That is, it gives a correspondence between $a_1, \ldots, a_{L_{d,e}}$ and $t_{i+1}, \ldots, t_{i+L_{d,e}+d}$, with exactly e differences.

At the beginning of each iteration i each $L_{d,e}$ has an empty such sequence. We use again the fact that initially (at Instruction 3) *row* is the maximum among $L_{d-1,e-1}$, $L_{d,e-1}+1$, and $L_{d+1,e-1}+1$ and finally (at Instruction 5) is $L_{d,e}$. Assume that we know the sequences of the predecessors of $L_{d,e}$ (Namely, the sequences of $L_{d-1,e-1}$, $L_{d,e-1}$ and $L_{d+1,e-1}$). We get the sequence of $L_{d,e}$ by adding triples to the end of the sequence of the predecessor which gives the maximum in initializing *row*. Let r_1 be the initial value of *row*. If r_1 got its value from $L_{d-1,e-1}$ (or $L_{d,e-1}$) then we add to its sequence the triple $(i+r_1+d-1, 0, 0)$. (Meaning that for t_{i+r_1+d}, there is no corresponding symbol in the pattern). Following Instruction 5, if $L_{d,e} > r_1$, we next add the triple $(i+r_1+d, r_1, L_{d,e}-r_1)$ to the sequence of $L_{d,e}$. This is done regardless of whether the source of r_1 was $L_{d-1,e-1}$ or $L_{d,e-1}$ or $L_{d+1,e-1}$. (This triple describes the match between substrings of the pattern and the text which was found during the computation of $L_{d,e}$ given $L_{d-1,e-1}$, $L_{d,e-1}$ and $L_{d+1,e-1}$).

At the end of iteration i we check which of the sequences of the $2k+1$ $L_{d,k}$ reached the rightmost symbol of the text. If the index of this symbol is greater than j ($L_{d,e}+d+i > j$), then we take its sequence to be the new $S_{i,j}$.

Complexity. The old Instruction 4 (where *row* is increased by one at a time without using $S_{i,j}$ and *MAX−LENGTH*) is employed each time we move to a new symbol of the text. We maintain $O(k)$ diagonals at any time during the text analysis and may need to compare the new symbol for each of them. Hence, the old Instruction 4 requires a total of $O(kn)$ time throughout the text analysis. In order to evaluate the number of steps which are required by the new Instruction 4 at iteration i, we use again the fact that $O(k)$ diagonals are computed. The sequence $S_{i,j}$ has at most $2k+1$ triples. We can charge each operation performed on any one of the diagonals to either a difference being discovered (there are $\leq k$ such differences), or to a triple of $S_{i,j}$ being examined (there are $\leq 2k+1$ triples). This amounts to $O(k)$ operation per diagonal at each iteration i. Therefore, the total running time of the text analysis is $O(k^2 n)$.

Analysis of the pattern

The pattern is an array $A = a_1, \ldots, a_m$. The output of the pattern analysis is the two dimensional array
$MAX-LENGTH[0,\ldots,m-1;0,\ldots,m-1]$ where
$MAX-LENGTH(i,j) = f$ means that
$a_{i+1}, \ldots, a_{i+f} = a_{j+1}, \ldots, a_{j+f}$ and
$a_{i+f+1} \neq a_{j+f+1}$.
The array *MAX−LENGTH* is symmetric. That is, $MAX-LENGTH(i,j) = MAX-LENGTH(j,i)$. We will apply a slight modification of the string matching algorithm of [KMP] due to [ML] which runs in $O(m+n)$ time. Given a pattern of length m and a text of length n this modification finds for each entry of text one of two things.

1. An occurrence of the pattern starts at this entry.
2. The first mismatch (from the left). That is, it finds the first character of the text which differs from a character of the pattern.

We compute each row l of *MAX−LENGTH* separately. Take $a_{l+1}, \ldots, a_m$ to be the pattern and $a_1, \ldots, a_m$ to be the text. We compute $MAX-LENGTH[l;1,\ldots,m-1]$ simply by applying this modification. The computation of each row takes $O(m)$ time. Since, there are m rows the total runing time of the pattern analysis is $O(m^2)$.

ACKNOWLEDGMENT. [I] 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.

REFERENCES

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

[BM] R.S. Boyer and J.S. Moore, "A fast string searching algorithm", Comm. ACM 20 (1977), 762-772.

[G84a] Z. Galil, "Optimal parallel algorithms for string matching", Proc. 16th ACM Symposium on Theory of Computing, 1984, 240-248.

[G84b] Z. Galil, "Open Problems in Stringology", preprint, 1984.

[GS] Z. Galil and J.I. Seiferas, "Time-space-optimal string matching", J. Computer and System Sciences 26 (1983), 280-294.

[I] A.G. Ivanov "Distinguishing an approximative word's inclusion on Turing machine in real time", (A Russian paper. The title is translated into English), Izvestia Academee Nauk USSR Seria Matematicheskiya 48 (1984), 520-568.

[KMP] D.E. Knuth, J.H. Morris and V.R. Pratt, "Fast pattern matching in strings", SIAM J. Comp. 6 (1977), 322-350.

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

[LV1] G.M. Landau and U. Vishkin "Efficient string matching with k mismatches", Theoret. Comput. Sci., to appear.

[LV2] G.M. Landau and U. Vishkin "Efficient string matching with k differences", TR-36/85, Department of Computer Science, Tel Aviv University, 1985.

[LVN] G.M. Landau, U. Vishkin and R. Nussinov "An efficient string matching algorithm with k differences for nucleotide and amino acid sequences ", TR-37/85, Department of Computer Science, Tel Aviv University, 1985.

[ML] M.G. Main and R.J. Lorentz, "An $O(n \log n)$ algorithm for finding all repetitions in a string", J. of Algorithms (1984), pp. 422-432.

[S] P. H. Sellers, The theory and computation of evolutionary distances: Pattern recognition. J. of Algorithms 1 (1980), 359-373.

[SK] D. Sankoff and J.B. Kruskal (editors), Time Warps, String Edits, and Macromolecules: the Theory and Practice of Sequence Comparison, Addison-Wesley, Reading, MA, 1983.

[U] E. Ukkonen, "On approximate string matching", Proc. Int. Conf. Found. Comp. Theor., Lecture Notes in Computer Science 158, Springer-Verlag, 1983, 487-495.

[U85] E. Ukkonen, "Finding approximate pattern in strings", J. of Algorithms 6 (1985), 132-137.

[V] U. Vishkin, "Optimal parallel pattern matching in strings", Proc. 12th ICALP, Lecture Notes in Computer Science 194, Springer-Verlag, 1985, 497-508. Also, Information and Control, to appear.

[WF] R. Wagner and M. Fisher "The string-to-string correction problem", J. ACM 21 (1974), 168-178.

iteration i

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

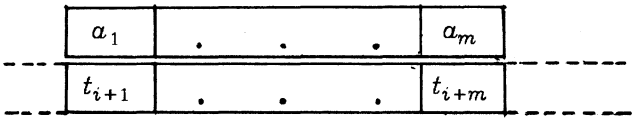

Figure 1(a)

$TEXT-MISMATCH[r;q,\ldots,k+1]$ gives all the mismatches between the following strings:

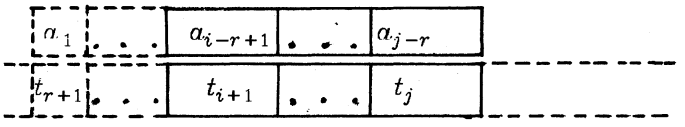

Figure 1(b)

$PAT-MISMATCH[i-r;1,\ldots,s]$ gives all, $(\leq 2k+1)$, the mismatches between the following strings:

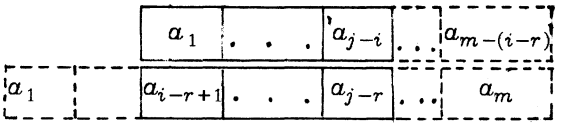

Figure 1(c)

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.

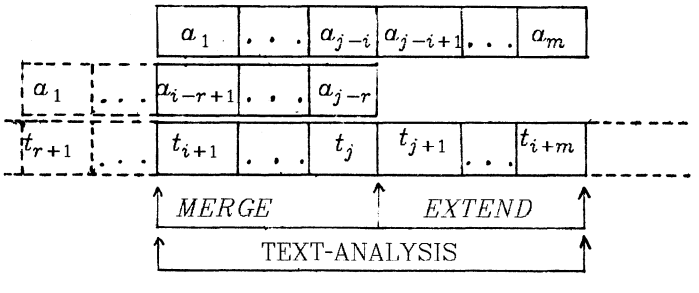

Figure 1(d)

Figure 1