INFORMATION AND CONTROL 64, 100–118 (1985)

Algorithms for Approximate String Matching*

ESKO UKKONEN

Department of Computer Science, University of Helsinki, Tukholmankatu 2, SF-00250 Helsinki, Finland

The edit distance between strings $a_1 \cdots a_m$ and $b_1 \cdots b_n$ is the minimum cost s of a sequence of editing steps (insertions, deletions, changes) that convert one string into the other. A well-known tabulating method computes s as well as the corresponding editing sequence in time and in space $O(mn)$ (in space $O(\min(m, n))$ if the editing sequence is not required). Starting from this method, we develop an improved algorithm that works in time and in space $O(s \cdot \min(m, n))$. Another improvement with time $O(s \cdot \min(m, n))$ and space $O(s \cdot \min(s, m, n))$ is given for the special case where all editing steps have the same cost independently of the characters involved. If the editing sequence that gives cost s is not required, our algorithms can be implemented in space $O(\min(s, m, n))$. Since $s = O(\max(m, n))$, the new methods are always asymptotically as good as the original tabulating method. As a by-product, algorithms are obtained that, given a threshold value t, test in time $O(t \cdot \min(m, n))$ and in space $O(\min(t, m, n))$ whether $s \leqslant t$. Finally, different generalized edit distances are analyzed and conditions are given under which our algorithms can be used in conjunction with extended edit operation sets, including, for example, transposition of adjacent characters.

1. INTRODUCTION

To define the edit distance between two strings, let $A = a_1 \cdots a_m$ be any string over an alphabet Σ and let the possible *editing operations* on A be:

(i) *deleting* a symbol from any position, say i, to give $a_1 \cdots a_{i-1} a_{i+1} \cdots a_m$;

(ii) *inserting* a symbol $b \in \Sigma$ at position i to give $a_1 \cdots a_i b a_{i+1} \cdots a_m$;

(iii) *changing* a symbol at position i to a new symbol $b \in \Sigma$ to give $a_1 \cdots a_{i-1} b a_{i+1} \cdots a_m$.

Each editing step can be understood as an application of a rewriting rule $a \to b$ where a and b, $a \neq b$, are in Σ or at most one of a and b is the empty string e. Rules with $b = e$ define deletions, rules with $a = e$ define insertions and rules with nonempty a and b define changes. Clearly, with these editing operations it is possible to convert, step-by-step, any string A into another string B.

* This paper is a revised and expanded version of a paper presented at the International Conference on "Foundations of Computation Theory" held in Borgholm, Sweden, August 21–27, 1983.

0019-9958/85 $3.00

Each editing operation $a \to b$ has a non-negative cost $\delta(a \to b)$. Given strings $A = a_1 \cdots a_m$ and $B = b_1 \cdots b_n$, we want to determine a sequence of editing operations which convert A into B so that the sum of individual costs of editing operations in the sequence is minimized. The minimum cost is denoted by $D(A, B)$ and called, as by Wagner and Fischer (1974), the *edit distance* from A to B; Sellers (1974) uses the term *evolutionary distance* while the idea was formulated already by Levenshtein (1966). The problem of computing $D(A, B)$ is also known as the *string-to-string correction problem*. Being able to compute the edit distance as well as the corresponding sequence of editing steps has applications in various string matching problems arising in areas such as information retrieval, pattern recognition, error correction, and molecular genetics.

Computing $D(A, B)$ becomes considerably simpler as soon as we may assume that there is always an editing sequence with cost $D(A, B)$ converting A into B such that if an element is deleted, inserted or changed, it is not modified again. This means that all editing operations could be applied on A in one parallel step yielding B; cf. the "traces" of Wagner and Fischer (1974).

As noted by Wagner and Fischer, this requirement is easily satisfied: It suffices that the cost function δ fulfills the triangle inequality, i.e.,

$$\delta(a \to c) \leqslant \delta(a \to b) + \delta(b \to c) \tag{1}$$

for all a, b, c such that $a \to c$, $a \to b$, and $b \to c$ are editing operations. We also assume that

$$\delta(a \to b) > 0 \tag{2}$$

for all operations $a \to b$. This is a natural requirement (since $a \neq b$) which is essential for our results. When (1) is true, distance $D(A, B)$ can be determined with a well-known tabulation method as follows: For all $0 \leqslant i \leqslant m$ and $0 \leqslant j \leqslant n$, denote by d_{ij} the edit distance $D(a_1 \cdots a_i, b_1 \cdots b_j)$ from string $a_1 \cdots a_i$ to string $b_1 \cdots b_j$. Then the $(m+1) \times (n+1)$ matrix (d_{ij}) can be computed from the recurrence

$$\begin{aligned} &d_{00} = 0 \\ &d_{ij} = \min(d_{i-1,j-1} + \text{IF } a_i = b_j \text{ THEN 0 ELSE } \delta(a_i \to b_j), \\ &\qquad\qquad d_{i-1,j} + \delta(a_i \to e), \\ &\qquad\qquad d_{i,j-1} + \delta(e \to b_j)), \qquad i > 0 \text{ or } j > 0. \end{aligned} \tag{3}$$

Clearly, matrix (d_{ij}) can be evaluated starting from d_{00} and proceeding row-by-row or column-by-column (and assuming that all undefined values d_{ij} referred to in the minimization step have default value ∞). This takes

time and space $O(mn)$. Finally, d_{mn} equals $D(A, B)$. Moreover, the sequence of editing steps that give $D(A, B)$ can be recovered from the matrix (d_{ij}) using the standard technique applied in dynamic programming in which one follows some "minimizing path" backwards from d_{mn} to d_{00} and records at each stage, which of the alternatives gives the minimum. So, if we have found that d_{ij} is on a minimizing path and, for example, $d_{ij} = d_{i-1,j} + \delta(a_i \to e)$ then $d_{i-1,j}$ is the next entry on the path and "delete a_i" is the editing operation.

This method (hereafter called the *basic algorithm*) with different variations has been invented and analyzed several times in various contexts, see, e.g., Lowrance and Wagner (1975), Needleman and Wunsch (1970), Sankoff (1972), Sellers (1974, 1980), Vintsyuk (1968), Wagner and Fisher (1974). Note that for computing d_{mn} without the editing sequence it suffices in the basic algorithm to save only one row or column of (d_{ij}) from which the next row or column can be generated. Hence only $O(\min(m, n))$ space is needed.

It turns out that the basic algorithm often evaluates unnecessary values d_{ij} and stores them inefficiently. These observations are presented in more detail in Section 2 where we also give the resulting improved algorithm for computing $D(A, B)$. Compared to the $O(mn)$ algorithm, the new method has the interesting feature that its efficiency does not depend only on m and n but also on the value of edit distance $D(A, B)$ to be computed. The smaller is $D(A, B)$, the faster is the algorithm. In Section 3 we modify the basic algorithm for the important special case where the cost function δ is constant. An application to the problem of computing the longest common subsequence is also considered. Section 4 presents some generalizations where we allow additional editing operations such as transpositions.

2. Improved Algorithm

Let us assume henceforth that the cost function δ satisfies (1) and (2) which means that recurrence (3) correctly defines matrix (d_{ij}). We now examine the relation between different entries d_{ij} more carefully.

Graphically, the dependencies between entries d_{ij} can be illustrated by drawing a directed arc from $d_{i'j'}$ to d_{ij} if and only if the minimization step in (3) gives d_{ij} from $d_{i'j'}$. The resulting graph is called the *dependency graph.* An example matrix (d_{ij}) for strings $A = yxxzy$ and $B = xyxzyz$ is shown in Fig. 1. The arcs of the dependency graph on paths from d_{00} to d_{56} are also represented.

Cost function δ used in the example is given by $\delta(a \to b) = 2$ whenever $a = e$ or $b = e$, and $\delta(a \to b) = 3$ in the remaining cases where $a \neq b$. From (3) it follows that vertical arcs correspond to deletions, horizontal arcs

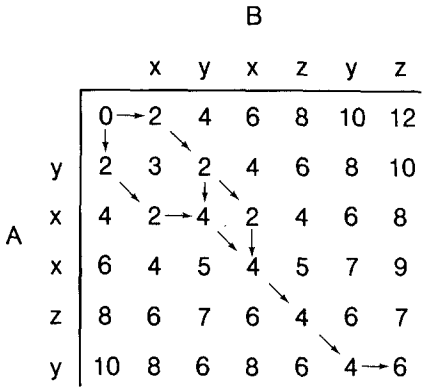

FIG. 1. Matrix (d_{ij}) with the dependency graph.

correspond to insertions, and diagonal arcs correspond to changes or matches. Moreover, if we label each arc with the cost $\delta(a \to b)$ of the associated editing operation (we let $\delta(a \to b) = 0$ if $a = b$), the value d_{ij} is the sum of labels on any path from d_{00} to d_{ij}. Hence we have

LEMMA 1. *If the dependency graph contains a directed path from d_{ij} to $d_{i'j'}$ then $d_{i'j'} = d_{ij} + d$, where d denotes the sum of labels on the path.*

The dependency graph can be understood as a subgraph of a larger graph of the form shown in Fig. 2. The graph has nodes (d_{ij}) and directed arcs such that an arc comes to d_{ij} from $d_{i-1,j}$, from $d_{i-1,j-1}$ and from $d_{i,j-1}$, and the costs associated with the arcs are $\delta(a_i \to e)$, $\delta(a_i \to b_j)$, and $\delta(e \to b_j)$, respectively. It is not difficult to see that the value of d_{mn} is the minimum total cost on the paths leading from d_{00} to d_{mn}. So the problem of computing the edit distance could be solved, say, with Dijkstra's algorithm for the single source shortest path problem which in this special case can be made to run in time $O(mn \log(mn))$. However, the regular

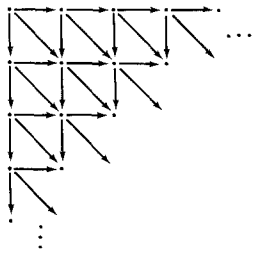

FIG. 2. Potential dependencies.

topology of the above graph allows simpler and more efficient solutions such as the basic $O(mn)$ method.

Returning to the dependency graph, it should be clear that only those entries d_{ij} that are on some path from d_{00} to d_{mn} are relevant for the value of d_{mn}. In fact, were some such path known a priori, we could compute d_{mn} by evaluating the entries on the path starting from d_{00} and assuming that all the entries not on the path have default value ∞. Also note that $d_{mn} = O(\max(m, n))$ since any path from d_{00} to d_{mn} contains at most $m+n$ arcs.

Consider now the problem of testing whether or not $D(A, B)$ is at most t where $t \geqslant 0$ is a given threshold value. This can be solved, of course, by evaluating (d_{ij}) with the basic algorithm and then testing whether $d_{mn} \leqslant t$. On the other hand, from Lemma 1 we know that the values d_{ij} are monotonically increasing along any path in the dependency graph. Therefore, if d_{mn} actually is $\leqslant t$ and if some d_{ij} gets a value larger than t, then d_{ij} cannot belong to any path leading to d_{mn}. Moreover, all entries that will not get a value $> t$, must be in a diagonal band of (d_{ij}) which is the narrower the smaller is t.

To make this precise, denote by Δ the minimum cost of all deletions and insertions, that is,

$$\Delta = \min(\delta(a \to b) \mid a \neq b \qquad \text{and} \qquad (a = e \text{ or } b = e)$$

and, by (2), $\Delta > 0$. To refer to the diagonals of (d_{ij}) we number them with integers $-m, -m+1, \ldots, 0, 1, \ldots, n$ such that the diagonal denoted by k consists of those d_{ij} for which $j - i = k$.

LEMMA 2. *If the dependency graph contains a directed path from d_{ij} to $d_{i'j'}$ then $d_{i'j'} \geqslant d_{ij} + |j' - i' - (j - i)| \cdot \Delta$.*

Proof. Since d_{ij} is on diagonal $k = j - i$ and $d_{i'j'}$ on diagonal $k' = j' - i'$, any path from d_{ij} to $d_{i'j'}$ contains at least $|k' - k|$ deletions (i.e., vertical arcs) if $k' - k \leqslant 0$ and at least $|k' - k|$ insertions (i.e., horizontal arcs) if $k' - k \geqslant 0$. Lemma 2 now follows from Lemma 1. ∎

Lemma 2 implies $d_{ij} \geqslant |j - i| \cdot \Delta$ for every d_{ij} on a path from d_{00} to d_{mn}, and so, by Lemma 1, $|j - i| \leqslant d_{ij}/\Delta \leqslant d_{mn}/\Delta$. Hence to compute d_{mn} it suffices to consider elements d_{ij} in the diagonal band given by $-d_{mn}/\Delta \leqslant j - i \leqslant d_{mn}/\Delta$. However, an even smaller diagonal band can be taken:

COROLLARY 1. *If d_{ij} is on some path leading from d_{00} to d_{mn} in the dependency graph then $-p \leqslant j - i \leqslant n - m + p$ if $m \leqslant n$, and $n - m - p \leqslant j - i \leqslant p$ if $m > n$, where $p = \lfloor \frac{1}{2}(d_{mn}/\Delta - |n - m|) \rfloor$.*

Proof. A path from d_{00} to d_{mn} consists of subpaths leading from d_{00} to d_{ij} and from d_{ij} to d_{mn}. Hence, from Lemma 2,

$$\begin{aligned} d_{mn} &\geqslant d_{00} + |j-i| \cdot \Delta + |n-m-(j-i)| \cdot \Delta \\ &= (|j-i| + |n-m-(j-i)|) \cdot \Delta. \end{aligned} \tag{4}$$

The rest of the proof is by a straightforward case analysis. For example, suppose that $n \geqslant m$ and $j \leqslant i$. Then from (4) we get $d_{mn} \geqslant [-(j-i)+n-m-(j-i)] \cdot \Delta$, that is, $d_{mn}/\Delta - (n-m) \geqslant -2(j-i)$. This means, because $j-i \leqslant 0$ is an integer and $n \geqslant m$, that $-\lfloor \frac{1}{2}(d_{mn}/\Delta - |n-m|) \rfloor \leqslant j-i \leqslant 0$, which is as required. The remaining cases are left to the reader. ∎

From Corollary 1 it follows that in testing whether $D(A, B) \leqslant t$, the evaluation of (d_{ij}) can be limited to the diagonal band which is, if p denotes $\lfloor \frac{1}{2}(t/\Delta - |n-m|) \rfloor$, between diagonals $-p$ and $n-m+p$ when $m \leqslant n$, and between diagonals $n-m-p$ and p when $m > n$. Figure 3 clarifies the numbering of the diagonals as well as shows the diagonal band for $m \leqslant n$.

So we obtain the following algorithm which assumes that all entries d_{ij} initially have value ∞:

```
PROCEDURE test_1(t):
  IF t/Δ < |n − m| THEN reject
  ELSE
    p := ⌊½((t/Δ) − |n − m|)⌋;
    FOR i := 0 UPTO m DO
      IF n ⩾ m THEN
        FOR j := max(0, i − p) UPTO min(n, i + (n − m) + p) DO
          evaluate d_ij from (3) ENDFOR
      ELSE
        FOR j := max(0, i + (n − m) − p) UPTO min(n, i + p) DO
          evaluate d_ij from (3) ENDFOR
      ENDIF;
    ENDFOR;
  ENDIF;
  IF d_mn ⩽ t THEN accept ELSE reject ENDIF.
```

Algorithm test_1 evaluates (in the nontrivial case $t/\Delta \geqslant |n-m|$) a band of (d_{ij}) that consists of $1+|n-m|+2p$ diagonals. Since each diagonal contains at most $\min(m, n)$ entries and since $1+|n-m|+2p \leqslant 1+t/\Delta = O(t)$, procedure test_1 evaluates $O(t \cdot \min(m, n))$ entries. Its time requirement is therefore $O(t \cdot \min(m, n))$. Also the space requirement can be made to $O(t \cdot \min(m, n))$ by storing only the entries in the band.

One immediately realizes, however, that to compute the next row of the

band, only the previous row is needed. Each row contains $1+|n-m|+2p=O(t)$ elements, hence the space complexity reduces in this way to $O(t)$. We get the following algorithm test_2 where array elements $r_0, r_1, \ldots, r_{|n-m|+2p}$ are used to successively store the rows of the diagonal band and r_{-1} and $r_{|n-m|+2p+1}$ are sentinels. Initially, $r_i=\infty$ for all i. Also assume $\delta(X\to Y)=\infty$ whenever $Y=b_h$ where $h<0$ or $h>n$:

```
PROCEDURE test2(t):
  IF t/Δ < |n-m| THEN reject
  ELSE
    p := ⌊½(t/Δ) - |n-m|)⌋;
    k' := k := IF n ≥ m THEN -p ELSE -p+(n-m);
    FOR i := 0 UPTO m DO
      FOR j := 0 UPTO |n-m|+2p DO
        r_j := IF i = j+k = 0 THEN 0
          ELSE min(r_j + IF a_i = b_{j+k} THEN 0 ELSE δ(a_i → b_{j+k}),
                   r_{j+1} + δ(a_i → e),
                   r_{j-1} + δ(e → b_{j+k}))
      ENDFOR;
      k := k+1;
    ENFOR;
  ENDIF;
  IF r_{|n-m|+2p+k'} ≤ t THEN accept ELSE reject ENDIF.
```

Instead of proceeding row-by-row in procedure test_2, an analogous columnwise evaluation of the diagonal band should be used when the columns are shorter than the rows, that is, when $m<n$. This makes the space requirement to $O(\min(t, m, n))$.

Procedure test_2 can further be improved by adding two pointers, p_1 and p_2, that point to the first and to the last value r_i which is $\leqslant t$; initially $p_1=1$ and $p_2=|n-m|+2p+1$. Then it suffices that j gets values starting from $\max(0, p_1-1)$, and when the interval represented by p_1 and p_2 vanishes, the algorithm can be terminated with rejection of t. This

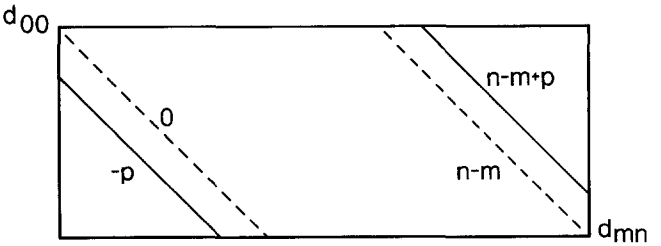

FIG. 3. Diagonals $-p$, 0, $n-m$, and $n-m+p$ when $m\leqslant n$.

modification does not improve the worst case complexity but is useful in practice because the diagonal band reserved for the test can be too broad.

Summarizing, we have proved

THEOREM 1. *There is an algorithm which, given strings* $a_1 \cdots a_m$ *and* $b_1 \cdots b_n$ *and a number* t, *tests in time* $O(t \cdot \min(m, n))$ *and in space* $O(\min(t, m, n))$ *whether or not* $D(a_1 \cdots a_m, b_1 \cdots b_n) \leqslant t$.[1]

It is also possible to determine the value of $D(a_1 \cdots a_m, b_1 \cdots b_n)$ with algorithm test_1 or with algorithm test_2: When $\text{test}_1(t)$ or $\text{test}_2(t)$ accepts t, we know that the value $d_{mn} = D(a_1 \cdots a_m, b_1 \cdots b_n)$ has been correctly computed. Hence test_1 or test_2 must be called successively with increasing values t, until t is accepted. Then d_{mn} gives the edit distance $D(a_1 \cdots a_m, b_1 \cdots b_n)$. For example, the following simple algorithm computes $D(a_1 \cdots a_m, b_1 \cdots b_n)$ in this way to s:

1. $t := (|n-m|+1) \cdot \Delta$;
2. WHILE $\text{test}_2(t)$ rejects DO $t := 2t$ ENDWHILE; (5)
3. $s := r_{|n-m|+2p+k'}$ where $r_{|n-m|+2p+k'}$ is as in test_2.

To analyze the time complexity of (5), let $t_0 = (|n-m|+1) \cdot \Delta$, $t_1 = 2t_0$, $t_2 = 2^2 t_0, \ldots, t_r = 2^r t_0$ be the values of t used as the parameters of test_2 on line 2. Noting our analysis of test_2, algorithm (5) needs time $O((\sum_{i=0}^{r} t_i) \cdot \min(m, n))$, that is, time $O(t_r \cdot \min(m, n))$. Since $s > t_r/2$, we get that the time complexity is, in fact, $O(s \cdot \min(m, n))$. The space requirement is dominated by the space for test_2, hence it is $O(\min(t_r, m, n))$, that is, $O(\min(s, m, n))$.

If the editing operation sequence that gives $D(a_1 \cdots a_m, b_1 \cdots b_n)$ is needed, algorithm (5) must be modified such that test_1 is used instead of test_2. The time requirement remains $O(s \cdot \min(m, n))$, but the space requirement increases to $O(s \cdot \min(m, n))$ since test_1 stores all entries in the diagonal band of (d_{ij}). From the stored values the editing steps can be recovered as explained in Section 1. So we have obtained

THEOREM 2. *The edit distance* $s = D(a_1 \cdots a_m, b_1 \cdots b_n)$ *as well as the corresponding sequence of editing steps can be computed in time and space* $O(s \cdot \min(m, n))$. *If the editing sequence is not needed, the space requirement can be reduced to* $O(\min(s, m, n))$.[1]

3. SPECIAL CASES

In this section we assume that each editing operation has the same cost, independently of the symbols involved. Without loss of generality, the con-

[1] To get correct upper bounds also when $t = 0$ or $s = 0$, one would prefer writing $t+1$ and $s+1$ instead of t and s in all O-expressions of this paper.

stant cost $\delta(a \to b)$ can be scaled to be $=1$. Then δ satisfies condition (1) and (2), and therefore the edit distance can again be computed from (3) which gets the form

$$\begin{aligned} &d_{00} = 0 \\ &d_{ij} = \min(d_{i-1,j-1} + \text{IF } a_i = b_j \text{ THEN 0 ELSE 1}, \\ &\qquad\qquad d_{i-1,j} + 1, \\ &\qquad\qquad d_{i,j-1} + 1), \qquad i > 0 \text{ or } j > 0. \end{aligned} \tag{6}$$

Now edit distance $D(a_1 \cdots a_m, b_1 \cdots b_n)$ simply means the minimum number of editing steps that transform $a_1 \cdots a_m$ into $b_1 \cdots b_n$.

It turns out that matrix (d_{ij}) can in this case be stored in a smaller space than for a general δ. This is because the values d_{ij} on the same diagonal form a non-decreasing sequence which increases in unit steps:

LEMMA 3. *Let the cost of every editing operation be equal to* 1. *Then for every* d_{ij}, $d_{ij} = d_{i-1,j-1}$ *or* $d_{ij} = d_{i-1,j-1} + 1$.

Proof. Since d_{ij} is always an integer, it suffices to show that $d_{ij} - 1 \leqslant d_{i-1,j-1} \leqslant d_{ij}$.

The minimization step in (6) directly implies that d_{ij} cannot be larger than $d_{i-1,j-1} + 1$, i.e., $d_{ij} - 1 \leqslant d_{i-1,j-1}$.

As regards the second inequality, it is trivially true for d_{00}. We proceed by induction on $i + j$. Assume first that the minimizing path to d_{ij} comes from $d_{i-1,j-1}$. Then (6) implies that $d_{ij} = d_{i-1,j-1}$ or $d_{ij} = d_{i-1,j-1} + 1$. Hence $d_{ij} \geqslant d_{i-1,j-1}$, as required. Assume then that the minimizing path to d_{ij} comes from $d_{i-1,j}$; the symmetric case where the path comes from $d_{i,j-1}$ is similar. Then again by (6), $d_{ij} = d_{i-1,j} + 1$. By induction hypothesis $d_{i-1,j} \geqslant d_{i-2,j-1}$. Hence $d_{ij} \geqslant d_{i-2,j-1} + 1$. Since $d_{i-1,j-1} \leqslant d_{i-2,j-1} + 1$ by (6), this implies that $d_{ij} \geqslant d_{i-1,j-1}$, as required. ∎

Lemma 3 suggests an alternative way of storing matrix (d_{ij}): For each diagonal of (d_{ij}), it suffices to store information which tells the points on the diagonal where the value increases. Formally, denote

$$f_{kp} = \text{the largest index } i$$

such that $d_{ij} = p$ and d_{ij} is on diagonal k. Since all values on diagonal k are $\geqslant |k|$, values f_{kp} are defined for $p = |k|, |k|+1, \ldots, p_{\max}$, where $p_{\max}$ is the largest value on diagonal k. In addition, it is convenient to define

$$\begin{aligned} f_{k,|k|-1} &= |k| - 1, \qquad \text{if } k < 0; \\ &= -1, \qquad \text{otherwise,} \end{aligned}$$

		x	y	x	z	y
	0	1	2	3	4	5
y	1	1	1	2	3	4
x	2	1	2	1	2	3
x	3	2	2	2	2	3
z	4	3	3	3	2	3

FIG. 4. Matrix (d_{ij}) for strings *yxxz* and *xyxzy*.

and $f_{kp} = -\infty$ for the remaining f_{kp} possibly referred to in algorithms to be presented.

An example matrix (d_{ij}) for strings *yxxz* and *xyxzy* is shown in Fig. 4. Since diagonal 1 of this matrix has values 1, 1, 1, 2, 3, we have that $f_{1,-1} = -\infty, f_{10} = -1, f_{11} = 2, f_{12} = 3, f_{13} = 4$.

Recovering each value d_{ij} from (f_{kp}) is simple: Find p such that $f_{k,p-1} < i \leqslant f_{kp}$, where $k = j - i$. Then $d_{ij} = p$. In particular, $d_{mn} = D(a_1 \cdots a_m, b_1 \cdots b_n)$ equals the unique p so that

$$f_{n-m,p} = m. \tag{7}$$

Clearly, storing matrix (d_{ij}) as (f_{kp}) does not increase space requirement. Rather, considerable saving is sometimes possible. For example, if the diagonal band evaluated by algorithm test_1 of Section 2 is represented with f_{kp}'s, the storage needed reduces to $O(t \cdot s)$, where $s = D(a_1 \cdots a_m, b_1 \cdots b_n)$. This further implies that the version of algorithm (5) which computes also the editing sequence can be made to work in space $O(s^2)$. More important is, however, that by adopting representation (f_{kp}) the edit distance can be computed in time $O(s \cdot \min(m, n))$ with a *direct* algorithm which avoids the reduction in (5) to tests $D(a_1 \cdots a_m, b_1 \cdots b_n) \leqslant t$. This decreases the constant factor in the time bound.

To develop this algorithm we need first an algorithm for computing (f_{kp}) directly without using d_{ij}'s as intermediate results. Assume that $p \geqslant |k|$ and that for all k', $f_{k',p-1}$ has been correctly computed. Then the following algorithm (8) computes f_{kp}:

1. $t := \max(f_{k,p-1} + 1, f_{k-1,p-1}, f_{k+1,p-1} + 1)$;
2. WHILE $a_{t+1} = b_{t+1+k}$ DO $t := t + 1$ ENDWHILE; (8)
3. $f_{kp} :=$ IF $t > m$ or $t + k > n$ THEN undefined ELSE t.

To prove (8) correct, note that by the induction hypothesis, the block of

entries with value $p-1$ reaches in matrix (d_{ij}) row $f_{k,p-1}$ on diagonal k, row $f_{k-1,p-1}$ on diagonal $k-1$, and row $f_{k+1,p-1}$ on diagonal $k+1$. Denote by t' the value of t after step 1, that is, $t'=\max(f_{k,p-1}+1, f_{k-1,p-1}, f_{k+1,p-1}+1)$, and by t'' the correct value of f_{kp}. We show that $t' \leqslant t''$ and that

$$a_{t'+1}=b_{t'+1+k},\ldots, a_{t''}=b_{t''+k},\ a_{t''+1}\neq b_{t''+k+1}. \tag{9}$$

This will show (8) correct since these conditions imply that the value of t equals t'' after step 2.

Assume, for example, that $t'=f_{k+1,p-1}+1$. Hence $d_{t'-1,t'-1+(k+1)}=p-1$. Since $t' \geqslant f_{k,p-1}+1$ and $t' \geqslant f_{k-1,p-1}$, we have $d_{t'-1,t'-1+k} \geqslant p-1$ and $d_{t',t'+(k-1)} \geqslant p-1$, and also $d_{t',t'+k} \geqslant p$. Then by (6), $d_{t',t'+k}=p$, and by Lemma 3, $t' \leqslant t''$, as required. To prove (9), notice first that from the definition of t'' it follows $d_{r,r+k}=p$ for $r=t',\ldots, t''$, and $d_{t''+1,t''+1+k}=p+1$ (or $t''+1>m$ or $t''+1+k>n$). Since $t'=\max(f_{k,p-1}+1,\ f_{k-1,p-1},\ f_{k+1,p-1}+1)$, the elements above and to the left of $d_{r,r+k}$ must be $\geqslant p$ for $r=t'+1,\ldots, t''+1$, that is, $d_{r-1,r+k} \geqslant p$ and $d_{r,r-1+k} \geqslant p$. Then, by (6), $d_{r,r+k}$ can be equal to p only if $a_r=b_{r+k}$, for $r=t'+1,\ldots, t''$, and $d_{t''+1,t''+1+k}$ can be equal to $p+1$ only if $a_{t''+1}\neq b_{t''+1+k}$. So (9) is true and the proof is complete.

To compute edit distance $s=D(a_1\cdots a_m, b_1\cdots b_n)$, we must find p such that (7) is true. This can be done with the next algorithm which calls algorithm (8) to compute (f_{kp}), column by column:

```
1. p := -1;
2. WHILE f_{n-m,p} ≠ m DO
3.     p := p + 1;
4.     FOR k := -p, -p+1,..., p DO                    (10)
           evaluate f_{kp} with algorithm (8) ENDFOR;
   ENDWHILE;
5. s := p.
```

Although (10) correctly computes s, it asks algorithm (8) to evaluate many entries f_{kp} whose value is actually undefined, because the range of values assigned for k in step 4 is too large. There are diagonals $-m, -m+1,\ldots, n$ in matrix (d_{ij}). Hence k can be restricted to $-m \leqslant k \leqslant n$. Moreover, each diagonal contains at most $\min(m, n)+1$ entries. This means, noting how f_{kp} is defined, that for a fixed k, f_{kp} may have a nontrivial value only for $p=|k|,\ |k|+1,\ldots, |k|+\min(m, n)$. Therefore it suffices that for a fixed p, k gets values such that $|k|=p-\min(m, n),\ldots, p$. Hence k must in step 4 satisfy the conditions

$$-m \leqslant k \leqslant n$$

and

$$(-p \leqslant k \leqslant -p + \min(m, n) \text{ or } p - \min(m, n) \leqslant k \leqslant p).$$

This can further be simplified such that we arrive at algorithm (11):

```
1. p := -1;
2. r := p - min(m, n);
3. WHILE f_{n-m,p} ≠ m DO
4.   p := p + 1;
5.   r := r + 1;
6.   IF r ⩽ 0 THEN FOR k := -p, -p + 1,..., p DO                (11)
       evaluate f_{kp} with algorithm (8) ENDFOR
     ELSE FOR k := max(-m, -p),..., -r, r,..., min(n, p) DO
       evaluate f_{kp} with algorithm (8) ENDFOR
     ENDIF;
   ENDWHILE;
7. s := p.
```

To analyze the time requirement of algorithm (11), let again $s = D(a_1 \cdots a_m, b_1 \cdots b_n)$. The values assigned for k in step 6 are in the range $-s,..., s$ and each value can occur for at most $\min(m, n) + 1$ different values p, or actually, for at most $\min(s, m, n) + 1$ different values p since $0 \leqslant p \leqslant s$ in step 6. Hence (11) evaluates $O(s \cdot \min(s, m, n))$ entries f_{kp} and therefore runs in time $O(s \cdot \min(s, m, n))$ without counting the time needed by the calls of algorithm (8) in step 6. The running time of (8) is dominated by the time of the while-loop in step 2. Obviously, test $a_{t+1} = b_{t+1+k}$ is performed for a fixed k at most once for each t. Therefore, for a fixed k again, the total time for step 2 during different calls of (8) is $O(\min(m, n))$. There are $O(s)$ different values k, hence the total time for the calls of algorithm (8) in algorithm (11) is $O(s \cdot \min(m, n))$. So (11) runs in total time $O(s \cdot \min(m, n))$.

As regards space, the above analysis shows that algorithm (11) evaluates $O(s \cdot \min(s, m, n))$ different entries f_{kp}. Hence $O(s \cdot \min(s, m, n))$ space suffices.

The editing operation sequence giving the edit distance s can be found from the stored values f_{kp} using a procedure that is analogous to the method used with the basic algorithm. In light of algorithm (8), one must now find a maximizing path leading to $f_{n-m,s}$. For example, the following procedure computes the editing operation sequence in time $O(s)$.

```
1.  p := s;
2.  k := n − m;
3.  WHILE p > 0 DO
4.    t := max(f_{k,p−1} + 1, f_{k−1,p−1}, f_{k+1,p−1} + 1);
5.    Let i, 1 ⩽ i ⩽ 3, be such that the ith of expressions
      f_{k,p−1} + 1, f_{k−1,p−1}, f_{k+1,p−1} + 1 has the largest value;
6.    IF i = 1 THEN
        announce edit operation "change a_t to b_{t+k}"
7.    ELSIF i = 2 THEN                                          (12)
        announce edit operation "insert b_{t+k} between a_t and a_{t+1}";
        k := k − 1
8.    ELSE
        announce edit operation "delete a_t";
        k := k + 1
      ENDIF;
9.    p := p − 1;
    ENDWHILE.
```

If the editing sequence is not needed, step 1 of algorithm (8) reveals that only values $f_{k,p-1}$ for all k are needed to evaluate values f_{kp}. Since $|k| \leqslant s$, the space requirement of (11) reduces to $O(s)$. This further reduces to $O(\min(s, m, n))$ since—as already noted—the same value $p-1$ can appear on at most $2 \cdot \min(m, n) + 1$ different diagonals of (d_{ij}), which means that values $f_{k,p-1}$ are nontrivial and need to be stored for at most $2 \cdot \min(m, n) + 1$ different k. So we have completed a proof of the next theorem.

THEOREM 3. *Let the cost of each editing operation be equal to* 1. *Then the edit distance* $s = D(a_1 \cdots a_m, b_1 \cdots b_n)$ *as well as the corresponding sequence of editing steps can be computed in time* $O(s \cdot \min(m, n))$ *and in space* $O(s \cdot \min(s, m, n))$. *If the editing sequence is not needed, the space requirement can be reduced to* $O(\min(s, m, n))$.

The only explicit difference between Theorem 3 and Theorem 2 is the smaller space bound $O(s \cdot \min(s, m, n))$ of Theorem 3. It should be emphasized, however, that algorithm (11) is simpler than algorithm (5). Hence the constant factors in Theorem 3 are smaller than in Theorem 2.

Also worth noting is that in the best case the running time of (11) can be significantly smaller than $O(s \cdot \min(m, n))$. This is in contrast with the basic algorithm of Section 1 which always needs time $\Theta(mn)$, and with algorithm (5) which always needs time $\Theta(s \cdot \min(m, n))$. At its best, algorithm (11) needs time $O(s^2 + \min(m, n))$. For example, the time requirement is of this

form for strings $(x^r y)^s$ and $(x^r z)^s$ whose edit distance is s. Algorithm (11) computes s in time $O(s^2 + sr)$.

Consider then the problem of testing for a given threshold value t, whether the edit distance of two strings is at most t. Clearly, this can be accomplished with a slightly modified algorithm (11): If p grows larger than t, announce that the edit distance is larger than t. Otherwise it is at most t. The method needs time $O(t \cdot \min(m, n))$ and space $O(\min(t, m, n))$.

Observe that Lemma 3 and hence Theorem 3 (and also Theorem 2) remain true if we reduce the editing operation set. For example, if insertion and deletion are the only operations, the correspondingly modified algorithm is as (11) but in the maximization step of algorithm (8) it suffices now to take the maximum of the second expression and the third expression.

When the cost of each individual editing step equals 1, computing the edit distance has an important application to finding the longest common subsequence (LCS) of two strings, as noted by Wagner and Fischer (1974). In fact, let s' be the edit distance of $a_1 \cdots a_m$ and $b_1 \cdots b_n$ when the allowed editing operations include only deletion and insertion. Then the length of the LCS for these strings is $r = (m + n - s')/2$. As already explained, a modified algorithm (11) computes s' from which we get r. The actual LCS can be found by performing on $a_1 \cdots a_m$ all deletions in the editing sequence that gives s'. For strings of approximately equal lengths, this method of computing the LCS seems as efficient as the recent method by Nakatsu, Kambayashi, and Yajima (1982). For example, in the case $m = n$ their algorithm takes time $O(m \cdot (m - r))$. This equals the time bound $O(s' \cdot m)$ of the modified algorithm (11), since $s' = 2m - 2r$.

Finally we consider possible generalizations of Lemma 3. One might suspect that Lemma 3 could be generalized to say that for all cost functions δ satisfying (1) and (2), the value of d_{ij} monotonically increases on every diagonal of (d_{ij}). That this is not the case, can be seen by the following example. Let the costs for the editing operations be

$$\delta(x, y) = \delta(y, x) = 1$$

$$\delta(0, x) = \delta(x, 0) = 3$$

$$\delta(0, y) = \delta(y, 0) = 2.$$

Then for strings $A = B = xy$, we obtain the matrix in Fig. 5, where the values on diagonals -1 and 1 are not monotonically increasing.

Assume, however, that the cost function δ satisfies (1) and (2), and that all deletions have the same cost and all insertions have the same cost. Thus for some constants c_1 and c_2 and for all $a \neq e$, $\delta(a \to e) = c_1$ and $\delta(e \to a) = c_2$. Then it is easy to modify the proof of Lemma 3 to show that

		x	y
	0	3	5
x	3	0	2
y	5	2	0

FIG. 5. Matrix (d_{ij}) for strings $A = B = xy$.

$d_{i-1,j-1} \leqslant d_{ij}$ for all i, j. Hence the value of d_{ij} increases along the diagonals of (d_{ij}), but the increments are not necessarily equal to 1. Representation (f_{kp}) cannot be used directly for (d_{ij}). However, denote by $v_k(r)$ the rth different value (in increasing order) occurring on diagonal k and by $f_k(r)$ the largest row index i such that $d_{i,i+k} = v_k(r)$. With these structures one can encode (d_{ij}) in algorithm (5). This complicates the procedure but improves the space efficiency in some cases.

4. EXTENSIONS

The problem of computing the edit distance can be extended in several directions. For example, editing operation sets that are larger than the set considered so far may be relevant in some applications. To generally analyze such extensions, we say that an *editing operation set* is any finite set $E \subset \Sigma^* \times \Sigma^*$ of ordered pairs (x, y), usually written as $x \to y$, over alphabet Σ such that $x \neq y$. Element $x \to y$ in E represents an editing operation that replaces an occurrence of x in a string in Σ^* by y. The editing operation set of Section 1 can be represented as

$$E_0 = \{x \to y \mid x, y \in \Sigma \cup \{e\}, x \neq y\}.$$

A cost function δ gives the cost $\delta(x \to y) > 0$ for each editing operation. We want again to determine a sequence of editing operations in E that convert a string $A = a_1 \cdots a_m$ into a string $B = b_1 \cdots b_n$ so that the sum of individual costs of editing operations is minimized. The minimum cost is denoted by $D_{E,\delta}(A, B)$. Hence our previous notation $D(A, B)$ is an abbreviation for $D_{E_0,\delta}(A, B)$. If it is not possible to transform A into B with the operations in E, we set $D_{E,\delta}(A, B) = \infty$.

For an arbitrary E, recurrence (3) defining matrix (d_{ij}) gets the form

$$\begin{aligned} &d_{00} = 0 \\ &d_{ij} = \min(\text{IF } a_i = b_j \text{ THEN } d_{i-1,j-1} \text{ ELSE } \infty, \\ &\qquad d_{i-k,j-r} + \delta(a_{i-k+1} \cdots a_i \to b_{j-r+1} \cdots b_j \mid \\ &\qquad\qquad a_{i-k+1} \cdots a_i \to b_{j-r+1} \text{ is in } E). \end{aligned} \tag{13}$$

However, d_{mn} computed from (13) is not equal to $D_{E,\delta}(A, B)$ for all E and δ. As mentioned in Section 1, a sufficient condition for equality is that no two steps are chained together in some sequence of editing steps giving $D_{E,\delta}(A, B)$.

To make this precise we define restricted editing sequences from A to B by specifying the *active part* for each intermediate string derived. At the beginning the whole A is active. Suppose then that we have arrived at an intermediate string uv with active part v. Let $x \to y$ in E be an editing operation such that x occurs in the active part, that is, v can be written as $v_1 x v_2$ for some (possibly empty) strings v_1 and v_2. Then an editing step that replaces x by y is an allowed operation, and v_2 is the active part of the new intermediate string $uv_1 yv_2$ obtained. If A produces B in this way, there is a *restricted editing sequence* from A to B. The minimum total cost of such sequences is denoted by $D'_{E,\delta}(A, B)$; if there are no restricted sequences from A to B, we set $D'_{E,\delta}(A, B) = \infty$. In some applications in error correction and in information retrieval, the restricted edit distance is a natural measure of similarity between different strings.

Obviously, $D'_{E,\delta}(A, B)$ is always $\geqslant D_{E,\delta}(A, B)$. Moreover, both distances coincide for editing operation set E_0 when the cost function satisfies (1). While $D_{E,\delta}$ is not effectively computable, $D'_{E,\delta}$ can be evaluated from (13), as can be easily shown by induction:

THEOREM 4. *Let matrix* (d_{ij}) *be defined by* (13). *Then* $d_{ij} = D'_{E,\delta}(a_1 \cdots a_i, b_1 \cdots b_j)$. *In particular,* $d_{mn} = D'_{E,\delta}(a_1 \cdots a_m, b_1 \cdots b_n)$.

As in Section 2, recurrence (13) defines a dependency graph over (d_{ij}). Lemma 1 is true also for this graph. To generalize Lemma 2, let

$$\Delta_{E,\delta} = \min(\delta(x \to y)/|p| \mid x \to y \text{ is in } E,\ p = |x| - |y| \neq 0).$$

(Here $|x|$, $|y|$ denote the length of strings x, y.) Then, as in the proof of Lemma 2, one sees that if there is a directed path from d_{ij} to $d_{i'j'}$ in the dependency graph defined by (13), then $d_{i'j'} \geqslant d_{ij} + |j' - i' - (j - i)| \cdot \Delta_{E,\delta}$. This is because a path from d_{ij} to $d_{i'j'}$ must cross at least $|j' - i' - (j - i)|$ diagonals, and the cost of crossing a diagonal is at least $\Delta_{E,\delta}$. Hence Lemma 2 as well as Corollary 1 are true with Δ replaced by $\Delta_{E,\delta}$. Furthermore, procedures test_1 and test_2 of Section 2, with every occurrence of Δ replaced by $\Delta_{E,\delta}$ and every occurrence of (3) replaced by (13), correctly decide whether $D'_{E,\delta}(A, B) \leqslant t$, and algorithm (5), after the same modification, correctly computes the restricted edit distance $D'_{E,\delta}(A, B)$. The modification does not change the complexity analysis of the algorithms. Hence the next generalization of Theorems 1 and 2 is true.

THEOREM 5. *There is an algorithm which, given strings* $a_1 \cdots a_m$ *and*

$b_1 \cdots b_n$ and a number t, tests in time $O(t \cdot \min(m, n))$ and in space $O(\min(t, m, n))$ whether or not restricted edit distance $s' = D'_{E,\delta}(a_1 \cdots a_m, b_1 \cdots b_n)$ is at most t. The value s' as well as the corresponding sequence of restricted editing steps can be computed in time and space $O(s' \cdot \min(m, n))$. If the editing sequence is not needed, the space requirement can be reduced to $O(\min(s', m, n))$.

So $D'_{E,\delta}$ can be evaluated efficiently while we do not consider algorithms for evaluating $D_{E,\delta}$. Note that the upper bound on $D_{E,\delta}$ given by $D'_{E,\delta}$ may be useful in some applications. An interesting related question is to characterize those E and δ for which $D_{E,\delta} = D'_{E,\delta}$.

Next we analyze a particular extension of E_0. Let $E_1 = E_0 \cup \{(ab, ba) \mid a, b \in \Sigma,\ a \neq b\}$, that is, E_1 is the set of deletion, insertion, and change operations extended with operations that transpose two adjacent symbols. Transposition is useful in correcting, e.g., typing errors. A related larger operation set was analyzed by Lowrance and Wagner (1975). We give a quite natural condition on δ which implies that $D_{E_1,\delta} = D'_{E_1,\delta}$.

THEOREM 6. *Let the cost function δ satisfy* (1) *and* (2) *and moreover, let $\delta(x \to y) \geq \delta(x' \to y')$ for every editing operation $x \to y$, $x' \to y'$ in E_1 such that $|xy| > |x'y'|$. Then $D_{E_1,\delta} = D'_{E_1,\delta}$.*

Proof. We show that for any editing sequence with operations in E_1 there exists an equivalent but restricted sequence of at most the same cost. A simple case analysis shows how to eliminate the first and hence all editing steps that do not operate on the active part. We give here only one example.

Let the first step $x \to y$ to be eliminated be a transposition. Hence $x = ab$ and $y = ba$ for some characters a, b. In addition, suppose that character a in x has been produced by an earlier transposition $ac \to ca$. So the total effect is to convert acb to cba. Now replace $ac \to ca$ and $ab \to ba$ with restricted steps $a \to e$ and $e \to a$ which have the same conversion effect but, by the assumptions of Theorem 6, at most the same cost. ▮

Assume finally, as in Section 3, that the cost function is constant. So $\delta(x \to y) = 1$ for all $x \to y$ in E_1. Then the conditions of Theorem 6 are satisfied and we could evaluate $D_{E_1,\delta}$ with a modified algorithm (5). But also Lemma 3 is immediately seen true for E_1 with constant cost function. Hence a more efficient solution is possible using the algorithms of Section 3. We briefly sketch the modifications necessary.

An expression that corresponds to transposition must be added to the list of expressions in the maximization step of algorithm (8). Therefore the following two steps replace step 1 of (8):

1a. $t := f_{k,p-1} + 1$;

1b. $t := \max(t, f_{k-1,p-1}, f_{k+1,p-1} + 1,$ IF $a_t a_{t+1} = b_{k+t+1} b_{k+t}$ THEN $t+1$ ELSE $-\infty)$.

Of course, algorithm (11) must now use (8), as just modified. In algorithm (12), step 4 as well as the rest of the algorithm must be expanded to cope with steps 1a and 1b.

5. Conclusion

We developed two versions of an algorithm that in time and in space $O(s \cdot \min(m, n))$ computes the edit distance s of two strings of length m and n. Both algorithms are easy to implement with small constant factors in the complexity bounds. The first algorithm works for arbitrary positive costs of individual editing steps. The second algorithm assumes that all steps have the constant cost equal to 1. Since $s = O(\max(m, n))$, the algorithms are asymptotically at least as efficient as the well known $O(mn)$ method, while for small s they are significantly faster. As a by-product, we derived algorithms to test in time $O(t \cdot \min(m, n))$ and in space $O(\min(t, m, n))$ for a given threshold value t, whether $s \leqslant t$. This kind of a test with a relatively small t is needed in applications where one wants to select from a larger set of strings all strings whose distance from a given string is at most t. In fact, the main stimulus to develops the methods of this paper came from certain applications in molecular genetics, where the $O(mn)$ algorithm is unnecessarily inefficient since m and n are large and t is small, cf., Peltola *et al.* (1983).

The derivation of the algorithm was based on a careful analysis of the $O(mn)$ method. Similar ideas can possibly be used in improving some other dynamic programming or tabulating algorithms.

Acknowledgments

The author is indebted to Hannu Peltola and Jorma Tarhio for useful remarks concerning this reasearch. The referees made several comments helpful in sharpening the formulation of the results.

References

1. Levenshtein, V. I. (1966), Binary codes capable of correcting deletions, insertions, and reversals, *Soviet Phys. Dokl.* **10**, 707–710.
2. Lowrance, R., and Wagner, R. A. (1975), An extension of the string-to-string correction problem, *J. Assoc. Comput. Mach.* **22**, 177–183.

3. NAKATSU, N., KAMBAYASHI, Y., AND YAJIMA, S. (1982), A longest common subsequence algorithm suitable for similar text strings, *Acta Inform.* **18**, 171–179.
4. NEEDLEMAN, S. B., AND WUNSCH, C. D. (1970), A general method applicable to the search for similarities in the amino acid sequence of two proteins, *J. Molecular Biol.* **48**, 443–453.
5. PELTOLA, H., SÖDERLUND, H., TARHIO, J., AND UKKONEN, E. (1983), Algorithms for some string matching problems arising in molecular genertics, *in* "Information Processing 83" (R. E. A. Mason, Ed.), pp. 59–64, North-Holland, Amsterdam.
6. SANKOFF, D. (1972), Matching sequences under deletion/insertion constraints, *Proc. Nat. Acad. Sci. U. S. A.* **69**, 4–6.
7. SELLERS, P. H. (1974), On the theory and computation of evolutionary distances, *SIAM J. Appl. Math.* **26**, 787–793.
8. SELLERS, P. H. (1980), The theory and computation of evolutionary distances: Pattern recognition, *J. Algorithms* **1**, 359–373.
9. VINTSYUK, T. K. (1968), Speech discrimination by dynamic programming, *Kibernetika (Kiev)* **4**, 81–88 [Russian]; *Cybernetics* **4**, 52–58.
10. WAGNER, R., AND FISCHER, M. (1974), The string-to-string correction problem, *J. Assoc. Comput. Mach.* **21**, 168–178.