A Faster Algorithm Computing String Edit Distances*

WILLIAM J. MASEK

MIT Laboratory for Computer Science, Cambridge, Massachusetts 02139

AND

MICHAEL S. PATERSON

School of Computer Science, University of Warwick, Coventry, Warwicks, United Kingdom

The edit distance between two character strings can be defined as the minimum cost of a sequence of editing operations which transforms one string into the other. The operations we admit are deleting, inserting and replacing one symbol at a time, with possibly different costs for each of these operations. The problem of finding the longest common subsequence of two strings is a special case of the problem of computing edit distances. We describe an algorithm for computing the edit distance between two strings of length n and m, $n \geqslant m$, which requires $O(n \cdot \max(1, m/\log n))$ steps whenever the costs of edit operations are integral multiples of a single positive real number and the alphabet for the strings is finite. These conditions are necessary for the algorithm to achieve the time bound.

1. INTRODUCTION

Wagner and Fischer [7] presented an algorithm for determining a sequence of edit transformations that changes one string into another. The execution time of their algorithm is proportional to the product of the lengths of the two input strings. The same three types of operations are used here, namely: (1) inserting a character into a string; (2) deleting a character from a string; and (3) replacing one character of a string with another. We present an algorithm with an asymptotically faster execution time, for example, $O(n^2/\log n)$ when both strings are of length n, providing that the alphabet for the strings is finite and all edit costs are integral multiples of some real number r.

This algorithm computes an optimal edit sequence for pairs of strings. As a special case it can compute the longest common subsequence of two strings.

For the infinite alphabet case Wong and Chandra [8] obtained upper and lower bounds proportional to n^2 using a slightly restricted model of computation. Aho *et al.* [1] obtained similar results for the longest common subsequence problem. Lowrance and Wagner [6]

18

extended the results of [7] to include the operation of interchanging adjacent characters. They developed an $O(n^2)$ algorithm solving their extended problem.

1.1. *Basic Definitions*

The following notation and conventions will be used.

A — A string of characters over some alphabet Σ.

$|A|$ — The length of string A.

A_n — The nth character of the string A ($|A_n| = 1$).

$A^{i,j}$ — The string $A_i \cdots A_j$ ($|A^{i,j}| = j - i + 1$).

A^n — An abbreviation for $A^{1,n}$.

λ — The null string also denoted A^0.

An *edit operation* is a pair $(a, b) \neq (\lambda, \lambda)$ of strings of length less than or equal to 1, also denoted as $a \to b$. String B results from string A by the edit operation $a \to b$, written "$A \to B$ via $a \to b$," if $A = \sigma a \tau$ and $B = \sigma b \tau$ for some strings σ and τ. We call $a \to b$ a *replacement* operation if $a \neq \lambda$ and $b \neq \lambda$, a *delete* operation if $b = \lambda$, and an *insert* operation if $a = \lambda$.

A sequence S of edit operations will be called an *edit sequence*. Let $S = s_1, s_2, ..., s_m$ be an edit sequence; an S *derivation from A to B* is a sequence of strings $C_0, C_1, ..., C_m$ such that $A = C_0$, $B = C_m$ and for all $1 \leqslant i \leqslant m$, $C_{i-1} \to C_i$ via s_i. (Note in this case the C_i's represent a sequence of complete strings, not individual characters.) If there is some S derivation of A to B, we say S *takes A to B*.

A *cost function γ* is a function assigning a nonnegative real number to each edit operation $a \to b$. We define $\gamma(S)$ for any edit sequence $S = s_1, ..., s_m$ to be $\gamma(S) = \sum_{1 \leqslant i \leqslant m} \gamma(s_i)$. The *edit distance $\delta(\gamma, A, B)$* from string A to string B using the cost function γ is defined by $\delta(\gamma, A, B) = \min\{\gamma(S) \mid S$ is an edit sequence taking A to $B\}$.

We may assume $\gamma(a \to b) = \delta(\gamma, a, b)$ for all edit operations $a \to b$. This leads to no loss of generality, since for any γ' we may define a new cost function γ by $\gamma(a \to b) = \delta(\gamma', a, b)$. Then γ satisfies the stated property and $\delta(\gamma', A, B) = \delta(\gamma, A, B)$ for all strings A and B.

Let δ denote the distance function between the strings A and B using the cost function γ; then we denote $\delta(\gamma, A^i, B^j)$ by $\delta_{i,j}$. We write the cost of replacing a with b as $R_{a,b}$, the cost of deleting a as D_a, and the cost of inserting a as I_a. We will assume $|A| \geqslant |B|$ throughout.

1.2. *Previous Results*

Wagner and Fischer's matrix-filling algorithm in [7] computes δ by constructing a $(|A| + 1) \times (|B| + 1)$ *edit matrix* whose i, jth entry is $\delta_{i,j}$ (Fig. 1a). They showed that each internal element of the matrix is determined by three adjacent matrix elements. The *initial vectors* of a matrix are its first row and column. The *final vectors* of a matrix are its last row and column. Theorem 1 describes how they computed the initial vectors

	B						
A	λ	b	a	b	a	a	a
λ	0	1	2	3	4	5	6
a	1	2	1	2	3	4	5
b	2	1	2	1	2	3	4
a	3	2	1	2	1	2	3
b	4	3	2	1	2	3	4
b	5	4	3	2	3	4	5
b	6	5	4	3	4	5	6

(a)

	B						
A	λ	b	a	b	a	a	a
λ	0	1	2	3	4	5	6
a	1			2			5
b	2			1			4
a	3	2	1	2	1	2	3
b	4			1			4
b	5			2			5
b	6	5	4	3	4	5	6

(b)

FIG. 1. (a, b) Computing distances with matrices. The alphabet is $\{a, b\}$. Assume $I = D = 1$, $R_{a,b} = R_{b,a} = 2$ and $R_{a,a} = R_{b,b} = 0$.

of the matrix, and Theorem 2 provides the rule for computing subsequent matrix elements.

THEOREM 1 [7]. $\delta_{0,0} = 0$, and for all i, j such that $1 \leqslant i \leqslant |A|$, $1 \leqslant j \leqslant |B|$,

$$\delta_{i,0} = \sum_{1 \leqslant r \leqslant i} D_{A_r}, \quad \text{and} \quad \delta_{0,j} = \sum_{1 \leqslant r \leqslant j} I_{B_r}.$$

THEOREM 2 [7]. For all i, j such that $1 \leqslant i \leqslant |A|$, $1 \leqslant j \leqslant |B|$:

$$\delta_{i,j} = \min(\delta_{i-1,j-1} + R_{A_i,B_j}, \delta_{i-1,j} + D_{A_i}, \delta_{i,j-1} + I_{B_j}).$$

Each of the $|A| \cdot |B|$ internal entries in the edit matrix for A and B can thus be computed in constant time, so the construction of the entire matrix can be performed with $O(|A| \cdot |B|)$ elements steps. Our algorithm reduces the time needed to $O(|A| \cdot |B|/\max(1, |B|/\log|A|))$ if the alphabet is finite and the edit costs are restricted.

2. A FASTER ALGORITHM

The transitive closure of a directed graph with n nodes can be easily computed with an $n \times n$ matrix using $O(n^2)$ row operations. Arlazarov, et al. [3] proved that if the matrix was split up into submatrices with a small number of rows, and all of the possible computations on submatrices were precomputed, the problem could be solved using $O(n^2/\log n)$ row operations. This algorithm is commonly referred to as "the four Russians' algorithm." Our algorithm applies similar techniques to Wagner and Fischer's edit matrices.

(Hopcroft, Paul, and Valiant [5] provided a generalized version of the four Russians' technique by showing that every computation performable in $O(n^2)$ steps on a multitape Turing machine can be performed in $O(n^2/\log n)$ steps on a unit-cost random-access

machine. Since the Fischer–Wagner algorithm can easily be implemented on a multitape Turing machine running in $O(n^2)$ steps, it might appear that our result follows as an immediate corollary. This, however, is not the case, since our method achieves time $O(n^2/\log n)$ on a random-access machine under the logarithmic cost criterion [2]. The operations executed on these machines have cost proportional to the length of their operands plus the length of the addresses of their operands. For example, storing the number n into a memory of size 2^m requires time m to address the memory register plus time $\log n$ to actually store the number.)

The four Russians' algorithm works faster by splitting the computation into many smaller computations. It computes all possible smaller computations, then puts them together (using some of the small computations many times) to get the larger computation. We follow a similar strategy. First, all possible $(m + 1) \times (m + 1)$ submatrices which can occur in the full matrix are computed for a suitably chosen parameter m; then these submatrices are combined to form the full matrix (like Fig. 1b), and the edit cost is computed.

2.1. Computing all the Submatrices

Define the (i, j, k) submatrix of the edit matrix δ to be the $k + 1 \times k + 1$ submatrix whose upper left corner entry is (i, j). (Fig. 1b shows the borders of the (i, j, k) submatrices $(0, 0, 3)$, $(0, 3, 3)$, $(3, 0, 3)$, and $(3, 3, 3)$.) It is obvious from Theorem 2 that the values in an (i, j, k) submatrix are determined solely by its initial vectors $\delta(i, j)$, $\delta(i, j + 1)$,..., $\delta(i, j + k)$, and $\delta(i, j)$, $\delta(i + 1, j)$,..., $\delta(i + k, j)$ along with its two strings $A^{i+1, i+k}$ and $B^{j+1, j+k}$. The first part of our algorithm computes the values for all (i, j, m) submatrices which can occur in any edit matrix using the same alphabet and cost function. It saves each submatrix's final vectors $\delta(i + m, j + 1)$,..., $\delta(i + m, j + m)$ and $\delta(i + 1, j + m)$,..., $\delta(i + m, j + m)$ to be used later.

To compute the final vectors for each possible submatrix we must first be able to enumerate the submatrices. We assume the alphabet is finite, so listing all length m strings is easy; however, listing all m-length initial vectors may take too long. As we get further into the matrix the values tend to increase, so listing all initial vectors may become uneconomical. However, under a modest restriction on the costs assigned to edit operations, there are only a finite number of differences between consecutive matrix values for all edit matrices using the same cost function and alphabet. We will operate with these differences instead. Define a *step* to be the difference between any two horizontally or vertically adjacent matrix elements and a *step vector* as a vector of steps. Corollary 1 expresses Theorem 2 in terms of steps.

COROLLARY 1 (of Theorem 2).

$$\delta_{i,j} - \delta_{i-1,j}$$
$$= \min\{R_{A_i, B_j} - (\delta_{i-1,j} - \delta_{i-1,j-1}), D_{A_i}, I_{B_j} + (\delta_{i,j-1} - \delta_{i-1,j-1}) - (\delta_{i-1,j} - \delta_{i-1,j-1})\},$$

$$\delta_{i,j} - \delta_{i,j-1}$$
$$= \min\{R_{A_i, B_j} - (\delta_{i,j-1} - \delta_{i-1,j-1}), D_{A_i} + (\delta_{i-1,j} - \delta_{i-1,j-1}) - (\delta_{i,j-1} - \delta_{i-1,j-1}), I_{B_j}\}.$$

Now each (i, j, k) submatrix may be determined by a starting value $\delta(i, j)$, two initial step vectors $\delta(i, j + 1) - \delta(i, j), \ldots, \delta(i, j + k) - \delta(i, j + k - 1)$ and $\delta(i + 1, j) - \delta(i, j), \ldots, \delta(i + k, j) - \delta(i + k - 1, j)$, along with the two strings $A^{i+1, i+k}$ and $B^{j+1, j+k}$. Then our algorithm can compute the final step vectors for each possible submatrix efficiently. To enumerate all possible submatrices we will enumerate all pairs of length m strings and all pairs of length m step vectors.

The initial phase of our algorithm in which all submatrices are computed can now be presented. Assuming some fixed ordering on the alphabet Σ and on the finite set of possible step sizes we enumerate all length m strings and all length m step vectors in lexicographic order. Then for each pair of strings C, D and pair of step vectors R, S Algorithm Y calculates a submatrix of steps according to Corollary 1. There are two classes of steps to consider, the ones moving horizontally and the ones moving vertically. Therefore our algorithm computes two matrices of steps: T consisting of the vertical steps and U consisting of the horizontal steps. The function *Store* saves R' and S', the final step vectors of the edit submatrix determined by C, D, R, and S so that they can be easily recovered given C, D, R, and S.

Algorithm Y

```
for each pair C, D of strings in Σ^m and
    each pair of length m step vectors R and S
do
    begin
        for i = 1 to m do
            begin
                T(i, 0) := R(i);
                U(0, i) := S(i);
            end;
        for i = 1 to m do
        for j = 1 to m do
            begin
                T(i, j) := min{R_{C_i, D_j} − U(i − 1, j), D_{C_i},
                                I_{D_j} + T(i, j − 1) − U(i − 1, j)};
                U(i, j) := min{R_{C_i, D_j} − T(i, j − 1),
                                D_{C_i} + U(i − 1, j) − T(i, j − 1), I_{D_j}}
            end;
        R' := ⟨T(1, m),..., T(m, m)⟩;
        S' := ⟨U(m, 1),..., U(m, m)⟩;
        Store (R', S', R, S, C, D);
    end;
```

Algorithm Y takes time $O(m^2 \log m)$ to calculate and store each submatrix. The $O(m^2)$ operations to calculate each submatrix each require time $O(\log m)$ to address their operands. Since we know beforehand exactly how many submatrices we must precalculate we can implement the *Store* function in time $O(m^2)$ by indexing into the RAM memory. If we assume there are a finite number of possible differences between costs of edit instructions, calculating all of the final step vectors takes total time $O(c^m m^2 \log m) = O(k^m)$ for some k depending only on the number of steps and Σ, but not m. Now we observe that the size of steps in a matrix is bounded independently of the strings involved.

LEMMA 3. *Let* $I = \max\{I_a \mid a \in \Sigma\}$, $D = \max\{D_a \mid a \in \Sigma\}$. *For all* A, B, i, j *such that* $1 \leqslant i \leqslant \mid A \mid, 1 \leqslant j \leqslant \mid B \mid$

   (i) $-I \leqslant \delta_{i,j} - \delta_{i-1,j} \leqslant D,$

   (ii) $-D \leqslant \delta_{i,j} - \delta_{i,j-1} \leqslant I.$

Proof. (i) A^i may be taken to B^j by first deleting A_i then taking A^{i-1} to B^j; therefore

$$\delta_{i,j} \leqslant \delta_{i-1,j} + D_{A_i} \quad \text{and so} \quad \delta_{i,j} - \delta_{i-1,j} \leqslant D_{A_i} \leqslant D.$$

Again A^{i-1} may be taken via A^i to B^j; thus

$$\delta_{i-1,j} \leqslant I_{A_i} + \delta_{i,j} \quad \text{or} \quad \delta_{i,j} - \delta_{i-1,j} \geqslant -I_{A_i} \geqslant -I.$$

Part (ii) follows by a similar argument. ∎

Let $\Omega = \{D_a \mid a \in \Sigma\} \cup \{I_a \mid a \in \Sigma\} \cup \{R_{a,b} \mid a, b \in \Sigma\}$. The set Ω is *discrete* only if there exists some constant r such that every element of Ω is some integral multiple of r. For finite alphabets, cost functions mapping into the integers or the rational numbers are always discrete while functions mapping into the real numbers may not be. We will show that if the set of edit costs is discrete then there is a finite set of steps occurring in the submatrices independent of the strings we are using for this computation. Hence Algorithm Y is applicable.

LEMMA 4. *If Ω is discrete then the set of possible steps in edit matrices is finite.*

Proof. Any element of an edit matrix is the sum of the costs of a series of edit operations. Therefore the steps are merely linear integral combinations of Ω. By Lemma 3, there is some real number b such that $-b < s < b$ for any possible step s. Since Ω is discrete, there is some real number $r > 0$ such that every step is a multiple of r. Hence there are at most $2\lfloor b/r \rfloor + 1$ possible steps. ∎

2.2. Computing the Edit Distance

The last stage of our algorithm, Algorithm Z, pieces together the (i, j, k) submatrices generated by Algorithm Y to form the edit matrix of steps. Then the actual edit costs can be calculated by summing the steps along any path to the end. Assume Fetch(R, S, C, D) returns a pair of final vectors of the submatrix determined by strings C and D,

and initial step vectors R and S. P and Q are matrices of length m vectors. Graphically P is the matrix of initial and final column vectors of $m \times m$ submatrices and Q is the corresponding matrix of row vectors. Define the function Sum(vector) to be the sum of a vector's components. Finally assume m divides $|A|$ and $|B|$.

Algorithm Z

for $i = 1$ to $|A|/m$ do $P(i,0) := \langle D_{A_{(i-1)m+1}}, ..., D_{A_{im}} \rangle$;

for $j = 1$ to $|B|/m$ do $Q(0,j) := \langle I_{B_{(j-1)m+1}}, ..., I_{B_{jm}} \rangle$;

for $i = 1$ to $|A|/m$ do

    for $j = 1$ to $|B|/m$ do

        $\langle P(i,j),\ U(i,j) \rangle := $ Fetch$(P(i, j-1),\ Q(i-1,j),$

                        $A^{(i-1)m+1,im},\ B^{(j-1)m+1,jm})$;

cost $:= 0$;

for $i = 1$ to $|A|/m$ do cost $:= $ cost $+$ Sum$(P(i,0))$;

for $j = 1$ to $|B|/m$ do cost $:= $ cost $+$ Sum$(Q(|A|/m, j))$;

We will now assume $|A| \geqslant |B|$. Since we know how many vectors Algorithm Y saved we can implement *Fetch* in time $O(m + \log|A|)$ by indexing into the RAM memory. It uses time $O(\log|A|)$ to read the relevant P and Q vectors plus time $O(m)$ to address and then read the saved vectors. The timing analysis of Z is now straightforward; it requires $O(|A| \cdot |B|/m^2)$ fetches and assignments of length m vectors, and hence, $O(|A| \cdot |B| \cdot (m + \log|A|)/m^2)$ basic steps. If we choose $m = \lfloor \log_k |A| \rfloor$ then the entire algorithm (both Y and Z) runs in time $O(|A| \cdot |B|/m)$. (Algorithm Y runs in time $O(k^m)$.)

The $O(|A| \cdot |B|/m)$ time bound is still achieved if m does not divide $|A|$ and $|B|$. Just pad out A and B with a dummy character not in the string, say ϕ, until A and B are multiplies of m. Then set $D_\phi = I_\phi = 0$, and for all a in Σ, $R_{a,\phi} = D_a$ and $R_{\phi,a} = I_a$. Note that if $m > |B|$ we can pad out B until it is length m so that our algorithm runs in time $O(|A|)$.

2.3. Edit Paths

Algorithms Y and Z describe how to compute the minimum edit cost between any two strings. Now we will explain how to recover a sequence of edit operations that achieves this minimum cost.

It is clear from the Wagner–Fischer algorithm that for any pair of strings A and B, it is enough to consider edit sequences S with the following properties. Each initial sequence S^r of S edits A^i to B^j for some i, j, and so corresponds to a matrix element. Furthermore, successive elements correspond to a path through the matrix since for each successive element either i, j, or both i and j, increase by 1. An *edit path* (Fig. 2) is any such sequence of elements through the matrix, not necessarily starting at the $(0, 0)$ cell. The sequences of elements in an edit path model sequences of insertions, deletions, and

				B			
A	λ	a	b	a	b	a	a
λ	<u>0</u>	1	2	3	4	5	6
b	<u>1</u>	2	1	2	3	4	5
a	<u>2</u>	<u>1</u>	2	1	2	3	4
a	3	2	<u>2</u>	2	2	2	3
b	4	3	2	<u>3</u>	2	3	4
a	5	4	3	2	<u>3</u>	<u>2</u>	3
b	6	5	4	3	2	3	<u>4</u>

FIG. 2. Paths in matrices. Assume $R_{a,a} = R_{b,b} = 0$, $R_{a,b} = 1$, $R_{b,a} = 2$, and $I = D = 1$. The underlined entries of the matrix form an edit path from (0, 0) to (5, 5) with cost 10. Its operations consist of D, D, I, R, R, R, I.

replacements according to their direction, and have costs depending on the symbols involved. The *cost* of an edit path is the sum of the costs of its operations.

Wagner and Fischer described an algorithm for recovering the edit sequence from the edit matrix by working backwards through it. The (i, j)th element of the matrix was originally calculated from the $(i, j - 1)$st, $(i - 1, j)$th, and $(i - 1, j - 1)$st elements along with insert, delete, and replace operations. The characters A_i and B_j are known, so it is easy to decide which operations could have given the value $\delta_{i,j}$. The procedure outputs any such operation as the last step of the edit path and remembers the preceding element $(i, j - 1)$, $(i - 1, j)$, or $(i - 1, j - 1)$. They applied this procedure recursively on the remembered element of each "last step" until it reached the starting element (0, 0). There are at most $2n$ steps in any optimal edit path, so this algorithm runs in time $O(n)$. We can use the same idea to recover the edit path from our discrete edit matrix by regenerating the $O(n/\log n) \log n \times \log n$ submatrices crossed by the optimal edit path. This would take time $O(n \log n)$. Alternatively, if Algorithm Y was modified to save every entry of each submatrix, and Algorithm Z saved a pointer to each submatrix whenever it was used, we could do this in linear time on a discrete edit matrix while using space $O(n^2/\log n)$.

2.4. Storage Requirements

Algorithm Y saves the initial and final step vectors for each of the submatrices. By our choice of m, there are only $O(n^{1/2})$ submatrices to save, so Algorithm Y's storage requirements are $O(n^{1/2} \log n)$ words.

The linear-time algorithm to construct an optimal edit sequence which was described in Section 2.3 required $O(n^2/\log n)$ space, but if only the edit distance is required then Algorithm Z may be made more economical of space using an observation of Hirschberg [4]. Since the ith row of submatrices depends only on the $(i - 1)$st row and the strings A and B, Algorithm Z can be modified to overwrite the $(i - 2)$th row with the ith row. Then Algorithm Z, and the whole algorithm would require only linear space.

3. Longest Common Subsequence

Let U and V be strings. U is a subsequence of length n of V if there exist $1 \leqslant r_1 < \cdots < r_n \leqslant |U|$ such that $U_i = V_{r_i}$. We say U is a *longest common subsequence* of A and B if U is a subsequence of both A and B and there is no longer subsequence of both A and B.

$|U|$ may be derived using the following cost function γ:

$$R_{a,b} = 0 \quad \text{if} \quad a = b \in \Sigma,$$
$$\phantom{R_{a,b}} = 2 \quad \text{otherwise}$$
$$D = I = 1.$$

Now $\delta = |A| + |B| - 2|U|$, or $|U| = (|A| + |B| - \delta))/2$. The domain is discrete, so if $|\Sigma|$ is finite, we can compute $|U|$ in time $O(|A| \cdot \max(1, |B|/\log|A|))$, using Algorithms Y and Z. We can compute the actual string U using an algorithm similar to our algorithm for recovering edit sequences. The cost function in this section comes from [7].

4. Discretion Is Necessary

The number of steps in all possible $n \times n$ matrices for a given alphabet and cost function has to be finite or very slowly growing for Algorithm Y to run efficiently. In this section we demonstrate with an example that the set of possible steps can grow linearly with the size of the strings. Algorithm Y has an exponential running time for the strings and edit costs we define.

4.1. The Example

Our example gives a nondiscrete cost function with an unbounded number of steps. Let $\Sigma = \{a, b, c\}$, then for all $\sigma \in \Sigma$ define the cost function γ:

$$R_{\sigma,\sigma} = 0,$$
$$R_{a,b} = R_{b,a} = 1,$$
$$R_{c,b} = R_{c,a} = R_{a,c} = R_{b,c} = \pi,$$
$$I_\sigma = D_\sigma = 5.$$

A and B are generated as follows. First let $A' = baba...$ and $B' = abab...$. We replace some characters of A' and B' with c's so that the number of c's in A^i (and B^i) equals μ_i where $\mu_{2k} = \mu_{2k+1} = \lfloor 2k/(2\pi + 1) \rfloor$. (Note that the c's only replace characters in even positions.) Figure 3 shows the first 50 characters of A and B with the c's inserted.

The edit costs are set up so the optimal paths contain many replacements, i.e., diagonal steps. (Figure 4 shows the edit matrix with $k = 10$.) We define $P(i, j, k)$ to be the minimum cost for an edit path from (i, j) to $(i + k, j + k)$. The *eccentricity* of (i, j) is

$$A^{50} = bababababcbababababcbababbcbababababcbababababcbababcbababa$$

$$B^{50} = abababacababababacababacababababacababababacababacabababab$$

Fig. 3. The first 50 characters of A and B.

B	λ	b	a	b	a	b	a	b	c	b	a
						A					
λ	0	5	10	15	20	25	30	35	40	45	50
a	5	1	5	10	15	20	25	30	35	40	45
b	10	5	2	5	10	15	20	25	30	35	40
a	15	10	5	3	5	10	15	20	25	30	35
b	20	15	10	5	4	5	10	15	20	25	30
a	25	20	15	10	5	5	5	10	15	20	25
b	30	25	20	15	10	5	6	5	10	15	20
a	35	30	25	20	15	10	5	7	$5 + \pi$	11	15
c	40	35	30	25	20	15	10	$5 + \pi$	7	$5 + 2\pi$	$11 + \pi$
a	45	40	35	30	25	20	15	11	$5 + 2\pi$	8	$5 + 2\pi$
b	50	45	40	35	30	25	20	15	$11 + \pi$	$5 + 2\pi$	9

FIGURE 4

$\mid i - j \mid$ and we define $P^*(i, j, k)$ to be the minimum cost for any path from (i, j) to $(i + k, j + k)$ through points all of eccentricity at least $\mid i - j \mid$. Paths consisting solely of replacements move down the diagonals of the edit matrix. There are two types of diagonals, even and odd, corresponding to the parity of the eccentricity of their associated paths. The odd diagonals correspond to paths with many cost 0 replacements (the a for a and b for b replacements) and relatively few large cost replacements (the replacements involving c's) to keep the paths from getting too efficient. The even diagonals correspond to paths with many small costs (the a for b and b for a replacements) and a few 0 cost replacements (the c for c replacements) to keep them efficient enough. Since the c's only occur in even positions they can only match up on the even diagonals.

We will compute $P(0, 0, n)$, $P(0, 1, n)$, and $P(1, 0, n)$ corresponding to the three central diagonal paths. By inserting c's with the precise densities chosen, none of the diagonal paths gets too efficient so they do not interact. However, if we inserted the c's in any periodic pattern one of the diagonal paths would eventually become more efficient and "overpower" another causing the pattern of steps to repeat. This would yield only a bounded number of steps for the example.

4.2. Comparing the Paths

A path along an odd diagonal costs π times the number of c's encountered in the path. A path along an even diagonal costs the length of the path minus the number of c for c replacements in the path, assuming the c's match up. Note that the only place we can guarantee that the c's match up is the center.

LEMMA 5. $P^*(i, j, k) \geqslant k - \mu_{i+k} + \mu_i$ if $i - j$ is even; $P^*(i, j, k) \geqslant (\mu_{i+k} - \mu_i + \mu_{j+k} - \mu_j)\pi$ if $i - j$ is odd.

Proof.   We proceed by induction on k. If $k = 0$, the result is obvious. For $k = 1$ we can enumerate the cases.

(i)   If $i - j$ is even,

$$P^*(i, j, 1) - 1 + \mu_{i+1} - \mu_i = R_{A_{i+1}, B_{j+1}} - 1 + (\mu_{i+1} - \mu_i)$$
$$\geqslant 0$$

since if $R_{A_{i+1}, B_{j+1}} < 1$ then $A_{i+1} = B_{j+1} = c$ and $\mu_{i+1} - \mu_i = 1$.

(ii)   If $i - j$ is odd

$$P^*(i, j, 1) - (\mu_{i+1} - \mu_i + \mu_{j+1} - \mu_j)\pi$$
$$= R_{A_{i+1}, B_{j+1}} - (\mu_{i+1} - \mu_i + \mu_{j+1} - \mu_j)\pi$$
$$= 0 \text{ if neither } A_{i+1} \text{ nor } B_{j+1} \text{ is } c \text{ or if just one of } A_{i+1} \text{ or } B_{j+1} \text{ is } c.$$

Note that $A_{i+1} = B_{j+1} = c$ is impossible.

For $k > 1$ there are two cases to consider.

Case 1. P^* is computed from two shorter paths with the same minimum eccentricity (Fig. 5a). Therefore $P^*(i, j, k) = P^*(i, j, k') + P^*(i + k', j + k', k - k')$ for some $0 < k' < k$ and the result follows directly.

Case 2. P^* is computed from a path with a higher minimum eccentricity (Fig. 5b). Without loss of generality, suppose $i \geqslant j$; then $P^*(i, j, k) = 5 + P^*(i + 1, j, k - 1) + 5$. There are two cases to look at.
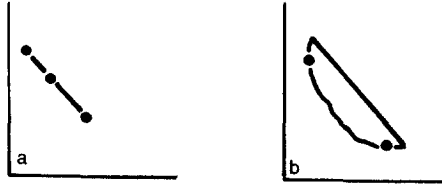

FIG. 5. (a, b) Alternatives to the diagonal paths.

(i)   If $i - j$ is even,

$$P^*(i, j, k) - k + \mu_{i+k} - \mu_i = 10 + P^*(i + 1, j, k - 1) - k + \mu_{i+k} - \mu_i$$
$$\geqslant 10 + (\mu_{i+k} - \mu_{i+1} + \mu_{j+k-1} - \mu_j)\pi - k + \mu_{i+k} - \mu_i$$
$$\geqslant 10 + (2\pi + 1)((k - 2)/(2\pi + 1) - 1) - k$$
$$\geqslant 7 - 2\pi > 0.$$

We are using the inequality $\mu_{r+s} - \mu_r \geqslant (s - 1)/(2\pi + 1) - 1$.

(ii)   If $i - j$ is odd,

$$P^*(i, j, k) - (\mu_{i+k} - \mu_i + \mu_{j+k} - \mu_j)\pi$$
$$= 10 + P^*(i + 1, j, k - 1) - (\mu_{i+k} - \mu_i + \mu_{j+k} - \mu_j)\pi$$
$$\geqslant 10 + k - 1 - \mu_{i+k} + \mu_{i+1} - (\mu_{i+k} - \mu_i + \mu_{j+k} - \mu_j)\pi$$
$$\geqslant 9 + k - (2\pi + 1)((k + 1)/(2\pi + 1) + 1)$$
$$= 7 - 2\pi > 0.$$

We are using the inequality $\mu_{r+s} - \mu_r \leqslant (s + 1)/(2\pi + 1) + 1$. ∎

The diagonal path is optimal for the center diagonal since we showed $P^*(0, 0, k) \geqslant k - \mu_k$ and $k - \mu_k$ is achieved by that path. Now we need to show the two center odd diagonal paths optimal. The path DR^k costs $5 + (\mu_{k+1} - \mu_1 + \mu_k - \mu_0)\pi = 5 + (\mu_{k+1} + \mu_k)\pi$.

LEMMA 6.   *For all k' $0 \leqslant k' \leqslant k$*

$$P^*(0, 0, k') + 5 + P^*(k' + 1, k', k - k') \geqslant 5 + (\mu_{k+1} + \mu_k)\pi.$$

Proof.   The minimal cost path to $(k + 1, k)$ consists of a path to (k', k'), followed by a deletion, followed by a path of eccentricity 1 from $(k' + 1, k')$ to $(k + 1, k)$. That path has cost $P^*(0, 0, k') + 5 + P^*(k' + 1, k', k - k')$.

By lemma 5:

$$P^*(0, 0, k') + 5 + P^*(k' + 1, k', k - k')$$
$$\geqslant k' - \mu_{k'} + 5 + (\mu_{k+1} - \mu_{k'+1} + \mu_k - \mu_{k'})\pi$$
$$= 5 + (\mu_{k+1} + \mu_k)\pi + k' - (1 + \pi)\,\mu_{k'} - \pi\mu_{k'+1}$$
$$\geqslant 5 + (\mu_{k+1} + \mu_k)\pi$$

because if k' is even

$$(1 + \pi)\,\mu_{k'} + \pi\mu_{k'+1} = (2\pi + 1)\,\mu_{k'} \leqslant k'$$

or if k' is odd

$$(1 + \pi)\,\mu_{k'} + \pi\mu_{k'+1} = (1 + \pi)\,\mu_{k'-1} + \pi\mu_{k'+1}$$
$$\leqslant ((1 + \pi)(k' - 1) + \pi(k' + 1))/(2\pi + 1)$$
$$\leqslant k'. ∎$$

When the results of Lemmas 5 and 6 are applied to the three diagonal paths in the center of the matrix we can calculate their string edit distances exactly.

LEMMA 7. *For all k,*

    (a) $\delta_{k,k} = k - \mu_k$ *and*

    (b) $\delta_{k,k+1} = \delta_{k+1,k} = 5 + (\mu_{k+1} + \mu_k)\pi.$

Proof. (a) Lemma 5 showed $P^*(0, 0, k) \geqslant k - \mu_k$ and $k - \mu_k$ is achieved by the center diagonal path.

    (b) Any path computing $\delta_{k+1,k}$ must consist of a path with eccentricity 0 followed by a deletion followed by a path with eccentricity 1. By Lemma 6 any such path must cost at least $5 + (\mu_{k+1} + \mu_k)\pi$. The path consisting of a deletion followed by k replacements (a center odd diagonal path) achieves this bound. ∎

In this example all of our conditions but discreteness hold but Algorithm Y does not work fast enough. Therefore discreteness is a necessary condition for using Algorithm Y.

THEOREM 5. *Discreteness is a necessary condition for Algorithm Y to run in time $O(k^m)$ on length m strings and step sequences.*

Proof. By Lemma 7 in our example the edit distances along the main diagonal form an increasing integer sequence, whereas the immediately adjacent diagonals are sequences increasing by multiples of π. Since π is irrational the number of different step sizes between these diagonals increases linearly with n, the string length. Therefore Algorithm Y's running time would be about $(kn)^m$ for some k, and no effective use could be made of its preprocessing. ∎

5. CONCLUSION

We have presented an algorithm for computing the shortest edit distance between two strings of length n in time $O(n^2/\log n)$. The algorithm works if the alphabet is finite and the domain for the cost function is discrete. Our analysis of an example shows the need for the discreteness condition. We showed that there are cases violating the discreteness condition when the algorithm does not work efficiently. The results in [1] show our algorithm cannot work if the alphabet is infinite. The most important problem remaining is finding a better algorithm for the finite alphabet case without the discreteness condition.

The question of the complexity of the shortest edit distance problem for finite alphabets is open. The best lower bound is linear in n [8], the upper bound is $O(n^2)$ or $O(n^2/\log n)$ depending on the discreteness condition. This gap seems too large and should be improved.

6. REFERENCES

1. A. V. AHO, D. S. HIRSCHBERG, AND J. D. ULLMAN, Bounds on the complexity of the longest common subsequence problem, *J. Assoc. Comput. Mach.* **23**, No. 1 (1976), 1–12.
2. A. V. AHO, J. E. HOPCROFT, AND J. D. ULLMAN, "The Design and Analysis of Computer Algorithms," Addison–Wesley, Reading, Mass., 1974.
3. V. L. ARLAZAROV, E. A. DINIC, M. A. KRONROD, AND I. A. FARADZEV, On economic construction of the transitive closure of a directed graph, *Dokl. Akad. Nauk SSSR* **194** (1970), 487–488 [in Russian]. English translation, *Soviet Math. Dokl.* **11** No. 5 (1970), 1209–1210.
4. D. S. HIRSCHBERG, A linear space algorithm for computing maximal common subsequences, CACM **18**, No. 6 (1975), 341–343.
5. J. E. HOPCROFT, W. J. PAUL, AND L. G. VALIANT, On time versus space and other related problems, *in* Proceedings, 16th Annual Symposium on Foundations of Computer Science, Berkeley, 1975," pp. 57–64.
6. R. LOWRANCE AND R. A. WAGNER, An extension of the string to string correction problem, *J. Assoc. Comput. Mach.* **22**, No. 2 (1975), 177–183.
7. R. A. WAGNER AND M. J. FISCHER, The string to string correction problem, *J. Assoc. Comput. Mach.* **21**, No. 1 (1974), 168–183.
8. C. K. WONG AND A. K. CHANDRA, Bounds for the string editing problem, *J. Assoc. Comput. Mach.* **23**, No. 1 (1976), 13–16.