Information Processing Letters 35 (1990) 317–323
North-Holland

15 September 1990

# AN O(NP) SEQUENCE COMPARISON ALGORITHM

Sun WU, Udi MANBER *, Gene MYERS **

*Department of Computer Science, University of Arizona, Tucson, AZ 85721, USA*

Webb MILLER ***

*Department of Computer Science, The Pennsylvania State University, University Park, PA 16802, USA*

Communicated by G.R. Andrews
Received 13 September 1989
Revised 25 April 1990

Let $A$ and $B$ be two sequences of length $M$ and $N$ respectively, where without loss of generality $N \geqslant M$, and let $D$ be the length of a shortest edit script (consisting of insertions and deletions) between them. A parameter related to $D$ is the number of deletions in such a script, $P = \frac{1}{2}D - \frac{1}{2}(N - M)$. We present an algorithm for finding a shortest edit distance of $A$ and $B$ whose worst-case running time is $O(NP)$ and whose expected running time is $O(N + PD)$. The algorithm is simple and is very efficient whenever $A$ is similar to a subsequence of $B$. It is nearly twice as fast as the $O(ND)$ algorithm of Myers, and much more efficient when $A$ and $B$ differ substantially in length.

*Keywords*: Algorithm, longest common subsequence, sequence comparison, shortest edit script

## 1. Introduction

Let $A$ and $B$ be two sequences of length $M$ and $N$ respectively, where without loss of generality $N \geqslant M$, and let $D$ be the length of a shortest edit script between them. The parameter $D$ is also known as the simple Levenshtein distance between the sequences [6]. The number of deletions and insertions in such a shortest script are also well-defined quantities. In particular, $P$, the number of deletions in a shortest edit script, is always equal to $\frac{1}{2}D - \frac{1}{2}(N - M)$, because there are $D$ insertions and deletions and $N - M$ more insertions than deletions.

* Supported in part by an NSF Presidential Young Investigator Award (grant DCR-8451397), with matching funds from AT&T.
** Supported in part by the National Institutes of Health (grant LM-04970).
*** Supported in part by the National Institutes of Health (grant LM-05110).

The problem of determining a shortest edit script (SES) or a longest common subsequence (LCS) between two sequences of symbols has been studied extensively [2,4,5,7,9,11,14,16]. The classic dynamic programming algorithm, invented by Wagner and Fischer [16] and others [12,15], has $O(MN)$ worst-case running time. Masek and Paterson [7] improved this algorithm by using the "Four-Russians" technique [1] to reduce the worst-case running time to $O(MN \log\log N/\log N)$ and $O(MN/\log N)$ for arbitrary and finite alphabet sets respectively. In terms of the input parameters $M$ and $N$ this bound has not been improved upon, but several recent designs have complexities that depend on output parameters such as $D$ and $P$. For example, Hunt and Szymanski [5] presented an algorithm whose running time is $O(R \log M)$, where $R$ is the total number of ordered pairs of positions at which the two sequences match. Later, Myers [9], Ukkonen [14], and Nakatsu et al. [11] gave algorithms with

0020-0190/90/$03.50 © 1990 – Elsevier Science Publishers B.V. (North-Holland)

worst-case time complexity $O(ND)$, which are efficient when $A$ and $B$ are similar. Such algorithms have been used in file comparison programs [8] and for economically updating the video screen by a text editing program [10]. Our algorithm is an improvement over the algorithms above since $P = \frac{1}{2}D - \frac{1}{2}\Delta$, where $\Delta = N - M$, and in practice our algorithm is always twice as fast as the $O(ND)$ algorithms. Its superiority is even more pronounced when the problem is highly asymmetric, i.e., $\Delta \gg 0$.

Our algorithm is best explained by casting the longest common subsequence problem as a shortest paths problem on a grid-like graph called an *edit graph* (e.g., see [9]). The algorithm improves upon Myers's algorithm [9] by exploring fewer of the vertices in the edit graph. It does so by using a path-compression technique that has been used as a heuristic for shortest paths problems [13]. This technique was also used by Hadlock [2] to give an $O(NP)$ sequence comparison algorithm, however, Hadlock used a version of Dijkstra's algorithm and thus the expected running time of his algorithm is also $O(NP)$, whereas the expected running time of our algorithm is $O(N + PD)$. Our fusion of a notion of compressed distances and Myers's greedy approach give an $O(NP)$ algorithm (when $P > 0$) that is very simple and thus very efficient in practice. The algorithm's dependence on $P$ implies that it is particularly efficient when $A$ is similar to a subsequence of the longer sequence $B$. The algorithm is $O(N)$ when $P = 0$, i.e., when $A$ is a subsequence of $B$. By using Hirschberg's divide-and-conquer technique [3,9], the algorithm can be modified to deliver a shortest edit script using only linear space.

## 2. Preliminaries

Let $A = a_1a_2a_3 \ldots a_M$ and $B = b_1b_2b_3 \ldots b_N$, $N \geqslant M$, be two strings of length $M$ and $N$ respectively. A sequence $C = c_1c_2c_3 \ldots c_L$ is called a *subsequence* of $A$ if $C$ can be derived from $A$ by deleting some characters of $A$. $C$ is called a *common subsequence* of $A$ and $B$ if $C$ is a subsequence of both $A$ and $B$. $C$ is called the *longest common subsequence* of $A$ and $B$ if the length of $C$ is the maximum among all common subsequences of $A$ and $B$. An *edit script* that edits sequence $A$ into $B$ is a list of *delete / insert* instructions where a delete instruction specifies which character of $A$ to delete and an insert instruction specifies which character of $B$ to insert. A *shortest edit script* is an edit script whose length is minimum among all possible edit scripts that edit $A$ into $B$. For example, if

$A$ = 'acbdeacbed' and $B$ = acebdabbabed',

then a longest common subsequence is 'acbdabed', and a shortest edit script is "insert $b_3$, delete $a_5$, delete $a_7$, insert $b_7$, insert $b_8$, insert $b_9$", where $a_i$ denotes the $i$th character of $A$ and $b_i$ denotes the $i$th character of $B$. The problem of finding a longest common subsequence (LCS) and of finding a shortest edit script (SES) are dual problems as reflected in the equality $D + 2L = M + N$ (where $L$ is the size of the LCS and $D$ is the size of the SES).

The edit graph for sequences $A$ and $B$ is a directed graph with a *vertex* at each grid point $(x, y)$, $0 \leqslant x \leqslant M$ and $0 \leqslant y \leqslant N$. Each vertex has a *horizontal* and a *vertical* edge to its right and lower neighbor if they exist. There is also a diagonal edge from $(x, y)$ to $(x+1, y+1)$ if $a_{x+1} = b_{y+1}$. The edit graph is constructed so that a path from *source* (0, 0) to *sink* $(M, N)$ corresponds to an edit script that converts $A$ into $B$: a horizontal edge corresponds to an insertion, a vertical edge corresponds to a deletion, and a diagonal edge represents a common symbol. By assigning cost 1 to the horizontal and vertical edges, and 0 to diagonal edges, the cost of a path equals the number of vertical and horizontal edges in it. Thus the problem of finding an SES/LCS is equivalent to finding a shortest source-to-sink path in the edit graph. Figure 1 shows the edit graph for

$A$ = 'acbdeacbed' and $B$ = 'acebdabbabed'.

A shortest path is highlighted and shows that $D = 6$ and $P = 2$.

Let diagonal $k$ of the edit graph be those vertices $(x, y)$ for which $y - x = k$. With this definition diagonals are numbered from $-M$ to $N$, diagonal 0 contains the source, and diagonal $\Delta = N - M$ contains the sink. The algorithm of

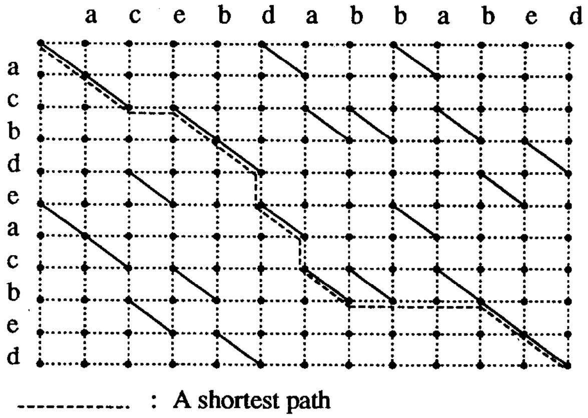


Fig. 1. Edit graph for $A =$ 'acbdeacbed' and $B =$ 'acebdabbabed'.

Myers [9] examines vertices between diagonal $-D$ and $D$, shown as the *D-band* in Fig. 2. Our algorithm only examines vertices in the smaller region between diagonals $-P$ and $\Delta + P$, shown as the *P-band* in Fig. 2. This is possible because any path passing outside the *P-band* must have more than $P$ vertical edges. To wit, if it passes through a vertex on a diagonal below $-P$, then it must traverse greater than $P$ vertical edges to reach the source, and if it passes through a vertex on a diagonal above $\Delta + P$, then it must traverse greater than $P$ vertical edges to reach the sink.

Let the *edit distance* to $(x, y)$, denoted $D(x, y)$, be the cost of the shortest path from the source to $(x, y)$ on diagonal $k = y - x$. Suppose that such a path contains $v$ vertical and $h$ horizontal edges. Then the number of nondiagonal edges is $v + h = D(x, y)$ and the path must end on diagonal $h - v = k$. Thus the number of vertical edges in a shortest path to $(x, y)$, $V(x, y)$, is well defined: it is equal to $\frac{1}{2}(D(x, y) - k)$. Similarly, the number of horizontal edges, $H(x, y)$, is equal to $\frac{1}{2}(D(x, y) + k)$. Let the *compressed distance* to $(x, y)$, $P(x, y)$, be defined as follows:

$$P(x, y) = \begin{cases} V(x, y), & \text{if } (x, y) \text{ is below diagonal } \Delta, \\ V(x, y) + (k - \Delta), & \text{if } (x, y) \text{ is above diagonal } \Delta. \end{cases}$$

The definition of compressed distance is the vertical distance $V(x, y)$ plus a lower bound on the number of vertical edges that must be in a path that continues from $(x, y)$ to the sink vertex. This bound is zero below diagonal $\Delta$ and is $k - \Delta$ above it since at least $k - \Delta$ vertical edges must be traversed to return to diagonal $\Delta$. Figure 3 depicts all *D-values* not greater than $D = 6$ and *P-values* not greater than $P = 2$ for the sequences of Fig. 1.

Like Myers's algorithm, our algorithm centers on computing a set of *farthest* vertices in order of distance until the sink is reached. The *farthest d-point in diagonal k* is the vertex on diagonal $k$ with $D$-value $d$ that has the greatest $y$- (or $x$-) coordinate. Let the $y$-coordinate of this point be denoted by

$$fd(k, d) = \max\{ y \colon D(y - k, y) = d \}.$$

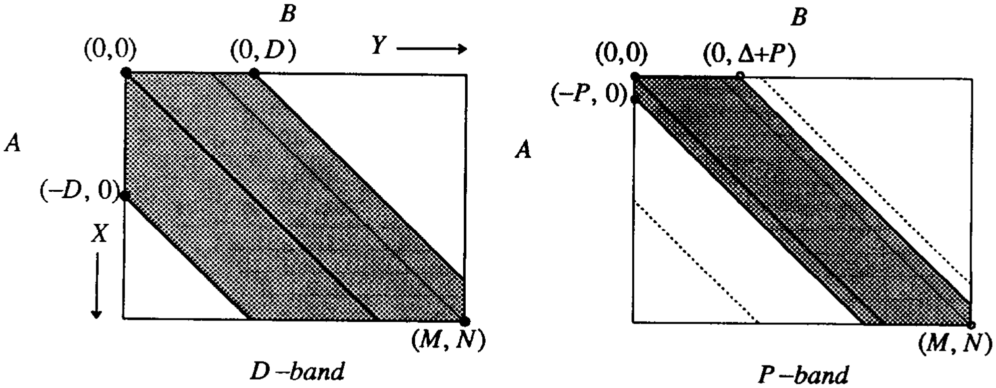


Fig. 2. *D-band* and *P-band* of an edit graph.

| | | a | c | e | b | d | a | b | b | a | b | e | d |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 0 | 0 | 0 | 1 | 2 | | | | | | | | |
| a | 1 | 0 | 0 | 0 | 1 | 2 | | | | | | | |
| c | 2 | 1 | 0 | 0 | 0 | 1 | 2 | | | | | | |
| b | | 2 | 1 | 1 | 0 | 0 | 1 | 2 | | | | | |
| d | | | 2 | 2 | 1 | 0 | 0 | 1 | 2 | | | | |
| e | | | | 2 | 2 | 1 | 1 | 1 | 2 | | | | |
| a | | | | | | 2 | 1 | 1 | 1 | 2 | | | |
| c | | | | | | | 2 | 2 | 2 | 2 | | | |
| b | | | | | | | | 2 | 2 | 2 | 2 | | |
| e | | | | | | | | | | | | 2 | |
| d | | | | | | | | | | | | | 2 |

(a)

| | | a | c | e | b | d | a | b | b | a | b | e | d |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | | | | | | |
| a | 1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | | | | | |
| c | 2 | 1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | | | | |
| b | 3 | 2 | 1 | 2 | 1 | 2 | 3 | 4 | 5 | 6 | | | |
| d | 4 | 3 | 2 | 3 | 2 | 1 | 2 | 3 | 4 | 5 | 6 | | |
| e | 5 | 4 | 3 | 4 | 3 | 2 | 3 | 4 | 5 | 6 | | | |
| a | 6 | 5 | 4 | 3 | 4 | 3 | 2 | 3 | 4 | 5 | 6 | | |
| c | | 6 | 5 | 4 | 5 | 4 | 3 | 4 | 5 | 6 | | | |
| b | | | 6 | 5 | 6 | 5 | 4 | 5 | 6 | | 6 | | |
| e | | | | 6 | | 6 | 5 | 6 | | | | 6 | |
| d | | | | | | | 6 | | | | | | 6 |

(b)

Fig. 3. An example of (a) $P$-values and (b) $D$-values.

The *set of farthest d-points* is

$$FD(d)=\{(y-k,\ y): y=fd(k,\ d) \text{ and } -d\leqslant k\leqslant d\}.$$

The set $FD(d)$ is the frontier of vertices whose edit distance is $d$. In Fig. 3, the farthest points are underlined. Our algorithm uses compressed distance, for which we make the analogous definitions:

$$FP(p)=\{(y-k,\ y):\ y=fp(k,\ p) \text{ and } -p\leqslant k\leqslant p+\triangle\},$$

where

$$fp(k,\ p)=\max\{y:\ P(y-k,\ y)=p\}.$$

## 3. The O($NP$) algorithm

Our algorithm computes the set $FP(p)$ from the set $FP(p-1)$ until $(M,\ N)\in FP(p)$ whereupon $P$ and $D=\triangle+2P$ are known. We first give an operational description of the algorithm and then formalize it in a recurrence that is rigorously proved. Let $q_k^p$ be the farthest $p$-point in diagonal $k$ (i.e., the point $(y-k,\ y)$, such that $y=fp(k,\ p)$). Assume that

$$FP(p-1)=\{q_{-(p-1)}^{p-1},\ q_{-(p-2)}^{p-1},\dots,q_{\triangle+(p-1)}^{p-1}\}$$

has already been found. The algorithm first computes $q_{-p}^p,\ q_{-(p-1)}^p,\dots,q_{\triangle-1}^p$ in this order (we now assume that $k<\triangle$). Vertex $q_k^p$ is found from $q_{k-1}^p$ and $q_{k+1}^{p-1}$ as follows. Let $a$ be the vertex immediately to the right of $q_{k-1}^p$ and $b$ be the vertex immediately below $q_{k+1}^{p-1}$ (see Fig. 4(a)). Both these vertices are on diagonal $k$. From the vertex with greatest $y$-coordinate, we follow diagonal edges until a vertex is reached that has no outgoing diagonal edge or that is on the lower boundary of the edit graph. This vertex is $q_k^p$ as proved in Lemma 1. The algorithm then goes above the diagonal and computes $q_{\triangle+p}^p,\ q_{\triangle+(p-1)}^p,\dots,q_{\triangle+1}^p$ (see Fig. 4(b)), this time using $q_{k-1}^{p-1}$ and $q_{k+1}^p$ to compute $q_k^p$ in the same fashion. (This time $a$ is the vertex immediately to the right of $q_{k-1}^{p-1}$ and $b$ is the vertex immediately below $q_{k+1}^p$.) Finally, $q_\triangle^p$ is computed from $q_{\triangle-1}^p$ and $q_{\triangle+1}^p$.

The procedure for computing $FP(p-1)$ from $FP(p)$ is formalized in Lemma 1 which gives a recurrence expression for $fp(k,\ p)$ in terms of the $y$-coordinates of previously computed farthest points. Let $snake(k,\ y)$ denote the $y$-coordinate of the farthest point on diagonal $k$ that can be reached from $(y-k,\ y)$ by traversing diagonal edges. Formally

$$snake(k,\ y)$$
$$=\max\{z:\ a_{y+1-k}\cdots a_{z-k}=b_{y+1}\cdots b_z\},$$

and informally *snake* models the process of following diagonal edges above. The correctness of the recurrence depends on a proper treatment of the boundary cases: $p=0$, $k=-p$, and

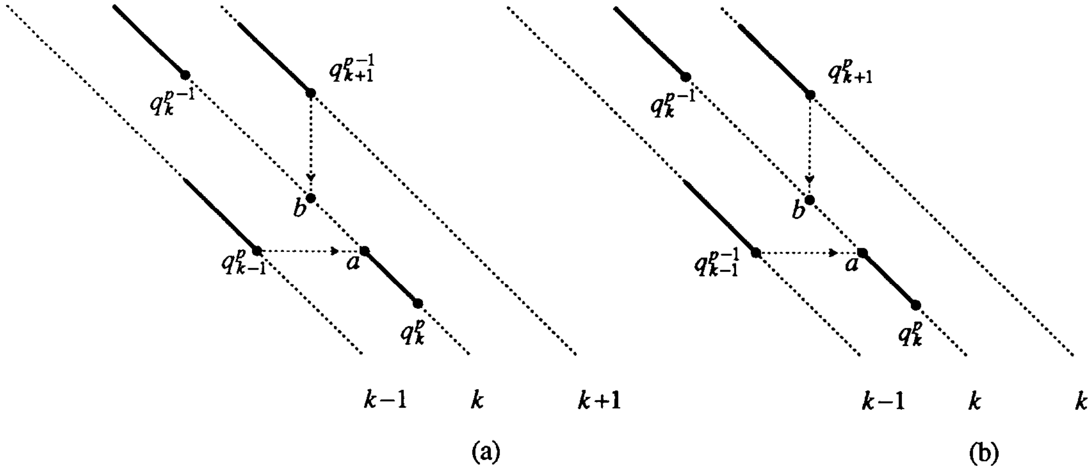


Fig. 4. Generating $FP(p)$ from $FP(p-1)$. (a) below diagonal $\Delta$, (b) above diagonal $\Delta$.

$k=\Delta+p$.These are handled cleanly by defining $fp(k, p)$ to be $-1$ whenever $p<0$ or $k \notin [-p, \Delta+p]$.

**Lemma 1.**

$$fp(k, p)=\begin{cases} snake(k, \max(fp(k-1, p)+1, \\ \qquad fp(k+1, p-1))), \\ \quad \text{if } k \in [-p, \Delta-1], \\ snake(k, \max(fp(k-1, p)+1, \\ \qquad fp(k-1, p))), \\ \quad \text{if } k=\Delta, \\ snake(k, \max(fp(k-1, p-1)+1, \\ \qquad fp(k+1, p))), \\ \quad \text{if } k \in [\Delta+1, \Delta+p]. \end{cases}$$

**Proof.** We give the proof only for the first case, $k<\Delta$; the proof for other cases is similar. Let $g$ be the farthest $p$-point in diagonal $k-1$ (i.e., $g=q_{k-1}^{p}$), and let $q$ be the farthest $(p-1)$-point in diagonal $k+1$ (i.e., $q=q_{k+1}^{p-1}$). Let $a$ be the vertex immediately to $g$'s right, let $b$ be the vertex immediately below $q$, and let $d$ be the farthest vertex reached from the farther of $a$ and $b$ along diagonal edges. The $y$-coordinate of $a$ is $fp(k-1, p)+1$, that of $b$ is $fp(k+1, p-1)$, and that of $d$ is given by the first case of the recurrence of the lemma. Figure 5 shows the two possible cases where $a$ is above $b$ (i.e., $fp(k+1, p)+1 \leqslant fp(k, p-1)$), and $b$ is above $a$. Again we focus just on the case shown in Fig. 5(a); the treatment of the other case is similar. The $P$-value of $d$ must be $p$ because there is a path to $d$ with compressed distance $p$ (i.e., the one passing through $q$ and $b$), and if there were a shorter path, then the vertex $c$ shown in Fig. 5(a) would have $P$-value less than $p-1$ contradicting the choice of $q$. It remains to show that $d$ is the farthest such point. A path of distance $p$ to a farther point cannot pass through $d$, because it would contradict the choice of $d$. But then it must pass through a vertex of distance $p$ on diagonal $k-1$ below $g$ or a vertex of distance $p-1$ on diagonal $k$ below $q$, contradicting the choices of $g$ and $q$, respectively. Thus such a path does not exist and $d$ is the farthest $p$-point in diagonal $k$. □

The simple sequence comparison algorithm in Fig. 6 is obtained directly from Lemma 1. The outer **repeat** loop is executed exactly $P+1$ times. In the $p$th pass of this loop, the upper **for** loop generates the points in $FP(p)$ on diagonals below $\Delta$. Note that by overwriting the $FP(p-1)$ points as it does so, only a single $M+N+1$ element array $fp$ is required for working storage. The lower **for** loop generates the points in $FP(p)$ above diagonal $\Delta$, and the next statement generates the farthest point on $\Delta$. An examination of the recurrence reveals that the points visited in the upper

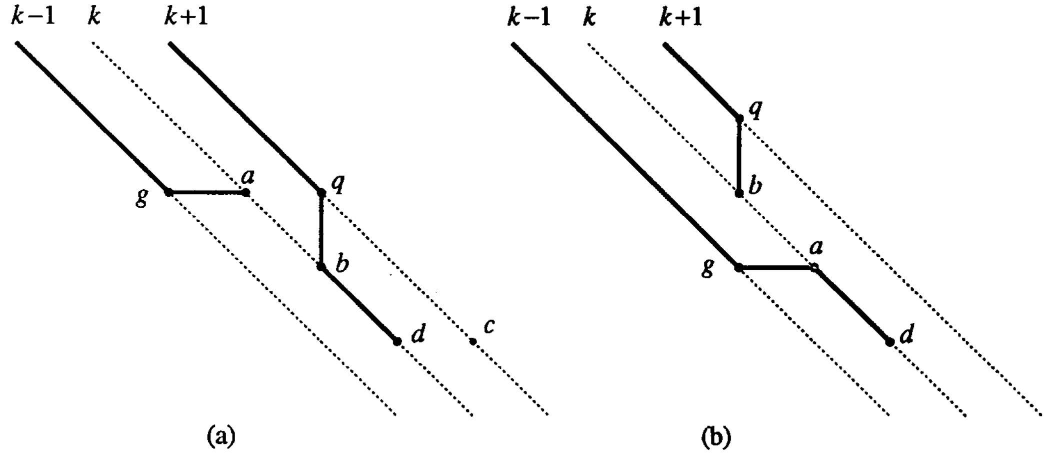


Fig. 5. The two cases of Lemma 1.

**for** loop are strictly increasing in their $y$-coordinate and the points visited in the lower **for** loop are strictly decreasing in their $x$-coordinate. Thus, the total time spent for one pass of the outer **repeat** loop is $O(N)$. So, the worst-case running time of the algorithm is $O(NP)$. To obtain the expected running time of the algorithm we observe that during a pass the total number of points visited in a particular diagonal is the number of diagonal edges traversed plus one (the frontier point). Let the total number of matched edges traversed be $R_P$. Then, the total number of points visited is $O(R_p + PD)$, because at most $D+1$ diagonals are covered in the computation. By an analysis as in [9], we can show that the expected number of traversed matched edges is $O(N + PD)$. The expected time complexity of the algorithm is therefore $O(N + PD)$.

## 4. Implementation

We implemented our algorithm and compared it to Myers's $O(ND)$ algorithm [9]. Table 1 shows the test results for 100 randomly generated strings. Table 1 shows average values over 100 trials on randomly generated strings over an alphabet of size 16. The fifth column of the table shows the number of comparisons (the same as the number of points visited in the edit graph) that were made during the computation for our $O(NP)$ algorithm. The sixth column shows the number of compari-

Table 1
Experimental results

| $M$ | $N$ | Number of deletions | Edit distance | Number of comparisons | | Execution time | |
|---|---|---|---|---|---|---|---|
| | | | | $O(NP)$ | $O(ND)$ | $O(NP)$ | $O(ND)$ |
| 4000 | 5000 | 10 | 1020 | 21564 | 526506 | 0.13 | 2.67 |
| 4000 | 5000 | 50 | 1100 | 59520 | 614391 | 0.41 | 3.12 |
| 4000 | 5000 | 100 | 1200 | 121635 | 737748 | 0.83 | 4.02 |
| 4000 | 5000 | 200 | 1400 | 255157 | 1004952 | 1.62 | 5.87 |
| 4000 | 5000 | 400 | 1800 | 600216 | 1693377 | 3.68 | 8.94 |
| 4000 | 5000 | 600 | 2200 | 1016433 | 2523687 | 6.41 | 14.85 |
| 5000 | 5000 | 200 | 400 | 49202 | 93139 | 0.33 | 0.61 |
| 5000 | 5000 | 600 | 1200 | 398499 | 791815 | 2.34 | 4.65 |

```
Algorithm Compare
  begin
    fp[-M...N] := -1;
    p := -1;
    repeat
      begin
        p := p + 1;
        for k := -p to Δ-1 do
          fp[k] := snake(k, max(fp[k-1]+1,
                                fp[k+1]));
        for k := Δ+p downto Δ+1 by -1 do
          fp[k] := snake(k, max(fp[k-1]+1,
                                fp[k+1]));
        fp[Δ] := snake(k, max(fp[k-1]+1, fp[k+1]));
      end
    until fp[Δ] = N;
    write "The edit distance is:" Δ+2p;
  end

  function snake(k, y: int): int
    begin
      x := y - k;
      while x < M and y < N and A[x+1] = B[y+1] do
        begin
          x := x + 1; y := y + 1;
        end
      snake := y;
    end
```

Fig. 6. Algorithm Compare.

sons made during the computation of the $O(ND)$ algorithm. The last two columns show running times on a VAX 8650 under 4.3bsd UNIX. As can be seen in the table, the speedup is quite large when $A$ and $B$ differ in length but are quite similar. When $A$ is approximately a subsequence of $B$, our algorithm runs in linear time.

## References


[1] 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); *Soviet Math. Dokl.* **11** (1975) 1209–1210 (in English).

[2] F. Hadlock, Minimum detour methods for string or sequence comparison, *Congr. Numer.* **61** (1988) 263–274.

[3] D.S. Hirschberg, A linear space algorithm for computing longest common subsequences, *Comm. ACM* **18** (1975) 341–343.

[4] D.S. Hirschberg, Algorithms for the longest common subsequence problem, *J. ACM* **24** (1977) 664–675.

[5] J.W. Hunt and T.G. Szymanski, A fast algorithm for computing longest common subsequences, *Comm. ACM* **20** (1977) 350–353.

[6] V.I. Levenshtein, Binary codes capable of correcting deletions, insertions, and reversals, *Problems Inform. Transmission* **1** (1965) 8–17.

[7] W.J. Masek and M.S. Paterson, A faster algorithm for computing string edit distances, *J. Comput. System Sci.* **20** (1980) 18–31.

[8] W. Miller and E.W. Myers, A file comparison program, *Software — Practice & Experience* **15** (1985) 1025–1040.

[9] E.W. Myers, An $O(ND)$ difference algorithm and its variations, *Algorithmica* **1** (1986) 251–266.

[10] E.W. Myers and W. Miller, Row replacement algorithms for screen editors, *ACM Trans. Program. Lang. Syst.* **11** (1989) 33–56.

[11] N. Nakatsu, Y. Kambayashi and S. Yajima, A longest common subsequence algorithm suitable for similar text string, *Acta Inform.* **18** (1982) 171–179.

[12] S.B. Needleman and C.D. Wunsch, A general method applicable to the search for similarities in the amino acid sequence of two proteins, *J. Molecular Biology* **48** (1970) 443–453.

[13] R. Sedgewick and J.S. Vitter, Shortest paths in Euclidean graphs, *Algorithmica* **1** (1986) 31–48.

[14] E. Ukkonen, Algorithms for approximate string matching, *Inform. and Control* **64** (1985) 100–118.

[15] T.K. Vintsyuk, Speech discrimination by dynamic programming, *Cybernetics* **4** (1968) 55–57.

[16] R.A. Wagner and M.J. Fischer, The string to string correction problem, *J. ACM* **21** (1974) 168–173.