JOURNAL OF COMPUTER AND SYSTEM SCIENCES 29, 133–152 (1984)

New Algorithms for the LCS Problem*

W. J. HSU AND M. W. DU

Institute of Computer Engineering,
National Chiao Tung University, Hsinchu, Taiwan, Republic of China

Received June 23, 1982; revised August 10, 1983

The LCS problem is to determine a longest common subsequence (LCS) of two symbol sequences. Two algorithms which improve two existing results, respectively, are presented. Let m, n be the lengths of the two input strings, with $m \leqslant n$, ρ being the length of the LCS, and s being the number of distinct symbols appearing in the two strings. It is shown that the first algorithm presented requires at most $O(n \log s)$ preprocessing time and $O(\rho m \log(n/m) + \rho m)$ processing time to solve the problem. This bound is better than that of previous algorithms especially when n is much greater than m. The algorithm also exhibits desirable properties under conditions of sparse matches. The second scheme achieves essentially the same bound $(O(\rho m \log(n/\rho) + \rho m))$ by employing efficient merging methods in the computations. It also outperforms existing algorithms designed for sparsely-matched situations. Together, the two algorithms provide interesting contrasts of different approaches to one problem; they also offer improved alternatives for actual implementation.

I. INTRODUCTION

A string is a sequence of symbols. For computer manipulations, each of the symbols must be some representable information, such as an integer or an English letter. A *subsequence* of a string is obtained by deleting 0 or more (not necessarily consecutive) symbols from the string. For example, "*abacd*" is a string, and "*bcd*" is a subsequence of it. A common subsequence (CS) of two strings, is a subsequence of both strings. A longest common subsequence (LCS) is one with the greatest length. For example, "top" is a CS of "entropy" and "topology", while "topy" is *the* LCS of the two strings. Notice that, in general, two strings may possess more than one LCS. For example, both "*abd*" and "*acd*" are LCSs of "*abcd*" and "*acbd*." The LCS problem is to find an LCS for two arbitrary input strings.

The LCS problem was first studied by molecular biologists [8, 9, 34, 36]; they found a use for LCS in studying similar amino acids. Subsequently, together with other string problems, the complexity of the LCS problem was analyzed by many computer scientists (see the References). Other cited applications include data compression [2, 12, 28], pattern recognition [11, 27], etc., where the LCS of two strings is used as a certain similarity measure of the objects represented by the

* This work was supported in part by the National Science Council of the Republic of China during the academic years 1980–1981.

0022-0000/84 $3.00

strings. Closely related problems concern the computations of string-to-string distance (string editing) [26, 30, 41, 42], shortest common supersequences [12, 28], string merging [20], pattern matching [23, 32], and tree-to-tree distance [11, 37].

The algorithms presented here are actually improvements to two recent algorithms, respectively. So the closely related results that precede us are reviewed first, then followed by our algorithms. Since the LCS problem is well explored, properties that are already known are just pointed out and explained, rather than proved, to avoid unnecessary formalism. We do hope this approach evokes more intuitions and interests.

II. Previous Results

We shall use A and B to denote the two input strings. Let $|A| = m$, $|B| = n$, without loss of generality, we assume that $m \leqslant n$. We also assume that the comparison of two symbols can be done in a constant amount of time, although a more realistic cost function should be logarithmic to the length of the codes. This assumption is still reasonable for almost every conceivable application.

The problem has been solved by using dynamic programming, e.g., [13, 35, 40]. The idea is based on the following observation. Consider two strings $A = a_1 a_2 \cdots a_i$ and $B = b_1 b_2 \cdots b_j$, if $a_i = b_j$, then by definition, an LCS of A and B must contain this coincident symbol as the last symbol; if $a_i \neq b_j$ then the last symbol of an LCS can be either a_i or b_j (but cannot be both) or even neither. In this case, an LCS can still be obtained even if we drop the "redundant" symbol from the string containing it.

Let $A[1 \cdots i]$ and $B[1 \cdots j]$ denote the strings $a_1 a_2 \cdots a_i$ and $b_1 b_2 \cdots b_j$, and $L_{i,j}$ be the length of LCS of $A[1 \cdots i]$ and $B[1 \cdots j]$. By the above observations, we have

$$\begin{aligned} &\text{If } a_i = b_j \quad \text{then} \quad L_{i,j} = L_{i-1,j-1} + 1 \\ &\qquad\qquad \text{else} \quad L_{o,j} = \max\{L_{i,j-1}, L_{i-1,j}\}. \end{aligned}$$

We also have the following boundary conditions:

$$\begin{aligned} L_{i,0} &= 0 \qquad \text{for} \quad i = 0, 1, 2, \ldots, \\ L_{0,j} &= 0 \qquad \text{for} \quad j = 0, 1, 2, \ldots, \end{aligned}$$

which simply means that the length of LCS of any string and a null string is zero.

In [40], Wagner and Fischer employ an array $L[0 \cdots m, 0 \cdots n]$ such that $L_{i,j}$ is kept in $L[i,j]$. With column 0 and row 0 of this array set to 0 initially, $L[i,j]$ are computed row by row according to the above recursion. The desired value of $L_{m,n}$ is in $L[m, n]$.

For example, $A = abcdbb$, $B = cbacbaaba$. After computing $L_{i,j}$, the "L matrix" is shown in Fig. 1. Since row i ($i > 0$) of the L matrix corresponds to $A[i]$, the ith symbol of A, and column j corresponds to $B[j]$, the corresponding symbols were

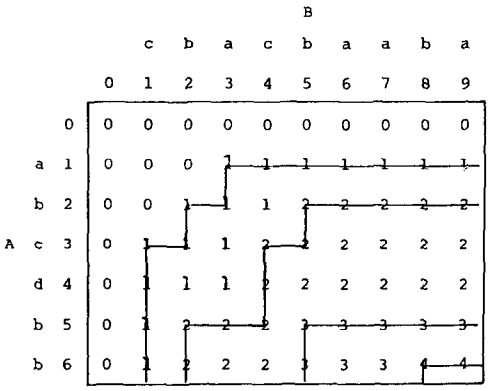


FIG. 1. L matrix.

shown. In this example, the length of LCS is in $L[6, 9]$, which is 4. Observe the regularity in Fig. 1; the region of $(i-1)$ is well separated from region $(i+1)$ by the region of i. We shall elaborate on this later.

Since we need to compute each of the $m \cdot n$ entries in the L matrix and computing each entry requires a constant amount of time, the total time required in computing the length of LCS is bounded by $O(m \cdot n)$, m, n being the lengths of the two strings. An LCS can be enumerated in $O(m+n)$ time after computing the matrix, so the total time is bounded by $O(m \cdot n)$. The storage required in this algorithm is also bounded by $O(m \cdot n)$.

Hirschberg [13] improves the space complexity to $O(m+n)$ by observing that in the calculation of $L_{i,j}$, only $L_{i-1,j-1}$, $L_{i-1,j}$, and $L_{i,j-1}$ are relevant, i.e., row i does not depend on row $(i-2)$ or any other previous rows, hence row $(i-2)$ can be erased by now i without affecting the compution of row $(i+1)$ which again will erase row $(i-1)$. In all, only 2 rows, instead of m rows, are needed in the computation. Note that the time required is still $O(m \cdot n)$, since as many entries are computed.

As pointed out in [2], the straightforward dynamic programming algorithm for the LCS problem has the disadvantage that it takes the same quadratic amount of time on all inputs because they make no use of the inherent structures of the strings. Thus we prefer an algorithm which is more efficient for typical inputs.

In the foregoing approach, every entry in the $m \times n$ matrix is computed. To develop algorithms that perform better than $O(mn)$, we must reduce the number of entries to be considered. Observing that a CS is composed of only matched symbols, we have a chance. See the following example:

Given $A = abcdbb$, $B = cbacbaaba$, Fig. 2(a) shows an * on (i,j), whenever $A[i] = B[j]$. By this argument, we need only consider the *'s on the diagram, instead of dealing with all the $m \times n$ entries.

Consider again the "L matrix" in a dynamic programming approach. Now with nonmatches abstracted away, see Fig. 2(b); since only matched symbols will constitute an LCS, if we can somehow compute the above $L_{i,j}$ values, we lose no

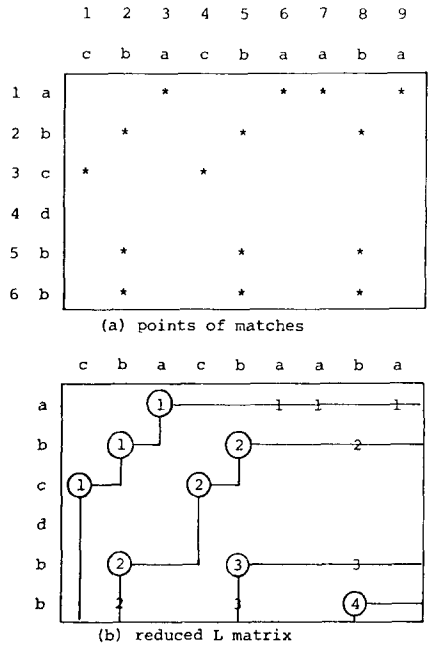


FIG. 2. Matrices with reduced entries.

necessary information for computing an LCS. The regularity exhibited in the L matrix in the dynamic programming approach suggests useful structural property which might ease the computations.

In the following, we first examine how to represent the *'s (the matches), then study the basic facts underlying each of the approaches. Two algorithms and our improvements are discussed subsequently.

Preprocessing

We may represent the points by keeping all the ordered pairs. For example, if $A = acab$, $B = ababacb$, the points are $\{(1, 1), (1, 3), (1, 5), (2, 6), (3, 1), (3, 3), (3, 5), (4, 2), (4, 4), (4, 7)\}$.

But the column/row regularity of the matches suggests better strategy. Specifically, we have the following property:

Let $PB_i = \{j \mid (i,j)$ is a match$\}$ be the set of all positions in string B that match $A[i]$, then $PB_s = PB_t$ if and only if $A[s] = A[t]$.

So it suffices to record a single copy of the set of *j-values* (positions in B) corresponding to each distinct symbol. See the following example.

EXAMPLE. For $A = abcdbb$, $B = cbacbaaba$, we may record the following information:

		j-Values	
	θ	Coincident Positions in B	$N[\theta]$:
"θ-lists" of String B	a	3, 6, 7, 9	4
	b	2, 5, 8	3
	c	1, 4	2
	d	nil	0

Now, given $A[1] = a$, we know, by inspection of the a list in the table, that (1, 3), (1, 6), (1, 7), (1, 9) are the matches on the first row of Fig. 2.

Similarly all other matches may be enumerated easily. To identify the coincident positions within a string is called a *string identification* problem in [2]. The known results may be used [14, 18].

We can set up the "θ-lists" for a string of length n in $O(n \log s)$ time, using $O(n)$ space, where s is the total number of distinct symbols appearing in the string [14]. Here a linear ordering of the symbols is assumed.

If the symbol set is known beforehand, this can be done even more efficiently by applying simple techniques, such as indexing-into-array, suggested in [2]. In this case, only $O(n)$ time and $O(n + \|\Sigma\|)$ space are needed. ($\|\Sigma\|$ denotes the cardinality of the alphabet Σ.)

In subsequent discussions, we assume that the θ-lists are already provided. $PB[\theta, 1]$, $PB[\theta, 2]$,..., $PB[\theta, N[\theta]]$ is the ordered list of j-values (ascending) that correspond to θ.

Preliminary Facts

For convenience, an ordered pair (i, j) with $A[i] = B[j]$ is now referred to as a "point," and the value of $L_{i,j}$ is called the *rank* of the point. Also, we would refer to the set of points that have the same rank as an "equipotential class", or "class K" (denoted C_K), if K is the rank of the members; (0, 0) constitutes class 0.

We now consider the regularity exhibited in the geometry of a class, then study the alternatives for determining the ranks of the points. For this purpose, we define a point (i, j) to be a *successor* of another point (i', j') iff $i' < i$ and $j' < j$, which simply means that the symbol corresponding to the point (i', j') precedes the symbol corresponding to (i, j) in both strings. Clearly, any point which is a successor to a point P must have a higher rank than p; also, any point in class K ($K \geqslant 1$) must be a successor to some point in class $(K - 1)$.

We may imagine that a point (i, j) divides the plane into four regions, as shown in Fig. 3a. Since (i, j) is a successor to any point in region II, any point in that region cannot be in the same class, otherwise (i, j) should have even greater rank. Similarly,

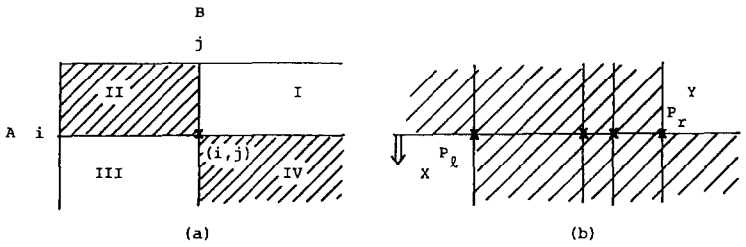


FIG. 3. Geometric properties.

points in region IV must be a successor to (i,j), so they must have higher ranks. The conclusion is, other members of the same class of (i,j), if they exist, must reside either in region I or in region III, or on the two axes (row i and column j).

Now suppose several class K points are known to be on the same row, as shown in Fig. 3(b). The leftmost point dictates that the region IV delineated by it (shadowed, under the row of points) cannot have any class K members; while the rightmost point also rules out the region II delineated by it. So the only regions left to be considered for class K are the unshadowed regions (X and Y) shown above. The above arguments apply to every row, hence the locations of points in the same class always shift to the left from row to row (cf. Fig. 2(b)).

Further, when determining a class, if we scan from row 1, row 2,..., in that sequence, with the upper portion being examined already, then only the lower-left region (region X in the above diagram) remains to be considered. The region X is completely determined by the leftmost point of this class considered so far, i.e., the position of this very point acts as a "threshold." The vertical break lines on Fig. 2(b) correspond to the "thresholds."

To closely correspond to the aforesaid ideas, we let C_k^i denote the partially determined class K associated with i, which is given by $C_k^i \equiv \{(s,t) \in C_k,\ 0 \leqslant s \leqslant i\}$; and let $P_k^i \equiv \min\{j \mid (i,j) \in C_k^i\}$, if $C_k^i \neq \varnothing$, to denote the "threshold" position of C_k at i (the leftmost position of rank K points considered so far). As already mentioned, P_k^i plays an important role in the determination of other class K members. We define $P_k^i = \infty$, if $C_k^i = \varnothing$. Clearly,

$$\infty = P_k^0 \geqslant P_k^1 \geqslant P_k^2 \geqslant \cdots \geqslant P_k^m \qquad \text{for any class } C_k \quad (k > 0).$$

The above left-shifting property says that if (i,j) is in class K then $j \leqslant P_k^{i-1}$. Moreover, $P_{K-1}^{i-1} < j$, since there must exist a point in class $(K-1)$, so that (i,j) is its successor. It can be seen that the above argument is also reversible. Hence, we have

THEOREM 1. *A point (i,j) is in class K, if and only if $P_{K-1}^{i-1} < j \leqslant P_K^{i-1}$.*

In [14], Hirschberg also pointed out that instead of keeping all points of all ranks, a subset called the "minimal candidates" is sufficient in detrmining a single LCS.

A point (i,j) is a minimal K-candidate iff $P_{K-1}^{i-1} < j < P_K^{i-1}$ and $j = P_K^i$. (cf. Lemma 3 in [14]).

The "minimal candidates" are circled in Fig. 2(b). We shall refer to a "minimal K-candidate" as a "k-key", for short. The above characterization says that (i,j) is a K-key, if and only if it is in class K; besides, it causes a change of the threshold position of the class. In Fig. 2(b) the keys are located at the corners of the "contours" [14]. It can be seen that each "contour" has at least one such turning point, i.e., if KEY(K) denotes the set of K-keys, then

$$\text{KEY}(K) = \varnothing \qquad \text{iff} \quad C_K = \varnothing.$$

III. Recent Developments

Approach 1

The algorithm in [14] by Hirschberg is very efficient and deserves some detailed study. Actually, it would be impossible to appreciate the merits without going into the delicate points. Besides, our first algorithm is closely related to it.

His algorithm considers class 1, then class 2, class 3,..., in that sequence, until classes of higher ranks are found to be empty. In considering a class, the algorithm scans points on a row from right to left testing for the conditions in Theorem 1, then proceeds to the next row.

Hirschberg's algorithm is based on the structural property that the locations of class K members on a given row are to the left side of all the class members in the last row. He avoids searching on the region that is bound to have no class K members.

He observes that if $A[i1] = A[i2] = \theta$, with $i1 < i2$, since $P_K^0 \geqslant P_K^1 \geqslant P_K^2 \geqslant \cdots \geqslant P_K^m$, we must have $P_K^{i1} \geqslant P_K^{i2-1}$. Suppose that we are determining which points at row $i2$ are members of class K—or, we are finding $(i2,j)$ with $P_{K-1}^{i2-1} < j \leqslant P_K^{i2-1}$; since $P_K^{i2-1} \leqslant P_K^{i1}$, a point $(i2,j)$ with $P_K^{i1} < j$ need not be considered. Recall that all the points at each row are kept in the order of (ascending) "*j-values*," i.e., if (i,j_1), (i,j_2),..., (i,j_t) are the points at row i, then the list $j_1, j_2,...,j_t$ is kept by our preprocessing. Since $A[i1] = A[i2] = \theta$, the lists of j-values corresponding to $i1$ and $i2$ actually coincide (cf. the discussion on preprocessing). So if at $i1$, we register the greatest u with $j_u \leqslant P_K^{i1}$, then, by the above argument, at row $i2$, we may just check the following list: $j_u, j_{u-1}, j_{u-2},...$, and disregard the other part of the original j-value list. Note that the above upper limit (u) in the search for class K members applies only if $A[i1]$ and $A[i2]$ contain the same symbol. So Hirschberg employs an array *high* $[\theta]$ to register, *for each distinct θ, the current upper bound of a search at a row containing θ.*

The lower limit of serach for class K members at row i in Hirschberg's algorithm is provided by P_{K-1}^{i-1}, which is also kept in an array.

It is precisely through such observations and arrangements, in Hirschberg's algorithm, when identifying a class K that the j-values examined at row $i1$ need not be examined again at row $i2$ ($i1 < i2$, $A[i1] = A[i2] = \theta$) excepting the j_u which joints

the two ranges of searches, The *exact* arrangements are much more complicated and are referred to the original paper.

Here is a simplified description of Hirschberg's algorithm. $PB[\theta, 1]$, $PB[\theta, 2]$,..., $PB[\theta, N[\theta]]$ is the (ordered) list of all j-values corresponding to θ, which is provided by the preprocessing, KEY(K) denotes the set of K-keys, which is empty initially.

$$K \leftarrow 0; \qquad P_0^0 \leftarrow 0;$$

repeat

$K \leftarrow K + 1$; KEY(K) $\leftarrow \phi$; $P_K^0 \leftarrow \infty$;
high$[\theta] \leftarrow N[\theta]$ for all θ,
For $i \leftarrow 1$ to m do {Scan the points from right to left}.
$\theta \leftarrow A[i]$,
while $PB[\theta, \text{high}[\theta]] > P_K^{i-1}$ Do high$[\theta] \leftarrow$ high$[\theta] - 1$;
while $P_{K-1}^{i-1} < PB[\theta, \text{high}[\theta]]$ Do $P_K^i \leftarrow PB[\theta, \text{high}[\theta]]$;
$\{P_K^i = \min\{j \mid (i,j) \in C_K^i\}$
high$[\theta] \leftarrow$ high$[\theta] - 1$
IF $P_K^i \neq P_K^{i-1}$ then KEY(K) $\Leftarrow (i, P_K^i)$

until KEY(K) $= \phi$.

Since different θ's occupy different positions in string B, the total number of all j-values canot exceed $|B|$, or n. Recall that at each row, when considering a class, only one j-value examined before is checked again, so the total number of j-values examined when considering a class can be bounded by $(|B| + |A|)$, or $(n + m)$, where $|A|$, or m, accounts for the number of "overlapped" j-values examined at all rows.

Hence at most $\rho(m + n)$ j-values are examined for ρ classes. Since examining a point takes a constant amount of time, his algorithm takes at most $O(\rho(m + n))$ time for computing the ranks. Again, since $m \leqslant n$ is assumed, so the time complexity is $O(\rho n)$.

Since only $O(m)$ time is required for the postprocessing (to enumerate an LCS), we shall concentrate on improving the major part of the whole work.

Improvement

In Hirschberg's algorithm, the lower bound for the determination of a K-key at a row, say row i, is set by the value of P_{K-1}^{i-1}. Essentially, the algorithm searches *sequentially* through the j-values that are located between the lower bound and the upper bound (high$[\theta]$, if $A[i] = \theta$). We propose an algorithm which preserves the same basic scheme (that avoids duplicated searches) but may benefit additionally from the power of binary searches.

First, we observe that a point cannot belong to more than one class, hence, for points that have been determined the ranks should not be considered again for higher ranks; rather, they should be "annihilated" in later computations. Since by Theorem 1, $(i,j) \in C_K$, if and only if $P_{K-1}^{i-1} < j \leqslant P_K^{i-1}$, for all ranks and all rows. So

the points of lower ranks will always be annihilated from the left portion of a row. *We use* LOW$[i]$ *as an index to the smallest j-value remaining to be considered at row i.*

Again, if $A[i] = \theta$, we use HIGH$[\theta]$ *to point to the greatest j-value to be considered at i.* Note that, in the algorithm shown below, HIGH$[A[i]]$ is set to keep the old LOW$[i]$ (in Step 6.4). This fact is important in our following analysis.

THRESH, whose value is a function of i, *denotes* P_K^{i-1}, *the threshold position of the class under consideration.*

$PB[\theta, 1]$, $PB[\theta, 2]$,..., $PB[\theta, N[\theta]]$, as before, is the ordered list of j-values (produced by the preprocessing phase) that correspond to θ, $N[\theta]$ being the number of θ's in B.

With the above devises, for a class C_K under consideration we iterate the following steps on i $(1 \leqslant i \leqslant m)$ (cf. Step 6 of the Algorithm A1),

(1) *Decide if there is a key of* C_K *for this i.* By Theorem 1, this can be done by checking $PB[\theta, \text{LOW}[i]]$, which is the minimum j-value of the remaining points at this row, against P_K^{i-1} (current THRESH). This is handled in 3 cases in Step 6.4.

Since it is possible that all points of a row are annihilated before—LOW$[i]$ is set to $N[A[i]] + 1$ (see next step). We will check, in Step 6.2, LOW$[i]$ against $N[A[i]]$, for the number of points in this row before using LOW as an index.

(2) *Annihilate points, if any, that belong to* C_K *by adjusting* LOW$[i]$. By our definition of LOW$[i]$, this can be done by finding t *the greatest integer such that* $PB[A[i], t] \leqslant P_K^{i-1}$, i.e. $t \equiv \max\{l \mid PB[A[i], l] \leqslant P_K^{i-1}\}$, then *setting* LOW$[i]$ *to* $1 + t$. We denote the search of t by a procedure FIND $(t, i,$ HIGH, LOW, THRESH$)$. We shall show that t can be found between HIGH and LOW, i.e., LOW$[i] \leqslant t \leqslant$ HIGH$[\theta]$, where $\theta = A[i]$.

(3) Set P_K^i (new THRESH) according to the result of (1), (2).

ALGORITHM A1 (A Variant of the Algorithm in [14]). Only the improved portion is shown. The length of LCS can be determined by executing this algorithm.

Steps:

```
(0)        LOW[i] ← 1                    for i = 1 to m;
(1)               K ← 0;
     Repeat
(2)     HIGH[A[i]] ← N[A[i]]             for i = 1 to m;
(3)               K ← K + 1;
(4)        KEY(K) ← ∅;
(5)        THRESH ← ∞;                   {∵ C_K^0 = ∅, P_K^0 = ∞, K > 0}
(6)     For i = 1 to m do
   (6.1)          θ ← A[i];
```

(6.2) IF LOW$[i] \leqslant N[\theta]$ then Some points at this row remain to be considered

Begin

(6.3) $j \leftarrow PB[\theta, \text{LOW}[i]]$; j is the minimum j-value at this i remaining to be considered

(6.4) **Case** 1. $j > \text{THRESH}$ $j > P_K^{i-1}$, by Theorem 1, none of the points at this row belongs to C_K

HIGH$[\theta] \leftarrow$ LOW$[i]$; $P_K^i < PB[\theta,$ high$[\theta]]$ after this operation

Case 2. $j = \text{THRESH}$ $j = P_K^{i-1}$, by Theorem 1, at this row, exactly one point belongs to C_K

HIGH$[\theta] \leftarrow$ LOW$[i]$; Hence, $P_K^i = PB[\theta,$ high$[\theta]]$ after this operation

LOW$[i] \leftarrow$ LOW$[i] + 1$; Delete this point from later considerations

Case 3. $j < \text{THRESH}$ j is the minimum j-value that has $P_K^{i-1} < j < P_K^{i-1}$; $\therefore$ (i,j) is a key, and $P_K^i = j$

KEY$(K) \Leftarrow (i,j)$;

FIND$(t, i,$ LOW, HIGH, THRESH); $\{t \equiv \max\{l \mid PB[\theta, l] \leqslant P_K^{i-1}\}\}$

THRESH $\leftarrow j$; $\{P_K^i = \min\{j \mid (i,j) \in C_K^i\}\}$

HIGH$[\theta] \leftarrow$ LOW$[i]$; $\{P_K^i = PB[\theta,$ high$[\theta]]$ after this operation$\}$

LOW$[i] \leftarrow t + 1$;

end{IF}

end{For}

until KEY$(K) = \varnothing$;

$\rho \leftarrow K - 1$; number of all nontrivial classes, or equivalently, the length of LCS

next phase: {For generation of an LCS, see [14]}.

As can be noticed, the algorithm developed here essentially follows the same scheme of Hirschberg's algorithm. The complete proof of correctness can be established in much the same manner [16] (basically, this requires Theorem 1 and the observations made earlier, coupled with some inductive arguments), hence is avoided. The operations done in Case 1 and Case 2 are quite simple, and can be verified

easily. The "FIND t" part in Case 3 needs some examination. Here we just show that t, the new LOW$[i]$, can indeed be found between LOW$[i]$ and HIGH$[A[i]]$.

Consider the following facts:

(1) Obviously, LOW$[i] \leqslant t$; because $PB[\theta, \text{LOW}[i]] < P_K^{i-1}$ (Case 3), and $t \equiv \max\{l \mid PB[\theta, l] \leqslant P_K^{i-1}\}$.

(2) Again, $t \leqslant$ HIGH$[A[i]]$: Recall that HIGH$[\theta]$ is initialized to $N[\theta]$ in Step 2; if i is the first (smallest) such that $A[i] = \theta$, then HIGH$[\theta]$ provides a sure upper bound for t. If i is not the first such that $A[i] = \theta$, i.e., there exists i', $i' < i$, and $A[i'] = A[i] = 0$ (besides, no i'', $i' < i'' < i$, such that $A[i''] = \theta$), then from the properties of "threshold" positions, we have $P_K^{i-1} \leqslant P_K^{i'}$. *Again, after Step* 6.4, $P_K^{i'} \leqslant PB[\theta, \text{HIGH}[\theta]]$ *is always satisfied.* (See the comments that accompany each of the assignments that set HIGH$[\theta]$ to LOW$[i]$). Hence, $PB[\theta, t] \leqslant P_K^{i-1} \leqslant P_K^{i'} \leqslant PB[0,$ HIGH$[\theta]]$, i.e., $t \leqslant$ HIGH$[\theta]$.

Therefore, we may realize the FIND t operation by means of a binary serach on the *range* provided by LOW$[i] + 1$,..., HIGH$[\theta]$, such that $t = \max\{l \mid PB[A[i], l] \leqslant$ THRESH$\}$. Note here, since HIGH$[\theta]$ is adjusted to LOW$[i]$ after the search, in considering a class, the ranges of separate binary searches (for separate occurrences of Case 3) are never overlapped. This observation is quite important and it is essential to the efficiency of the algorithm. The detailed analysis of the performance of our algorithm is given in the Appendix. As expected, our algorithm improves the preceding one.

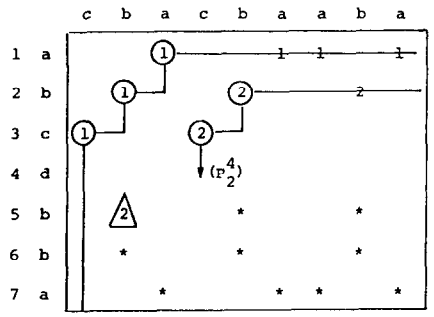
Approach 2

In [18], Hunt and Szymanski compute the ranks in a different manner. Rather than determining class 1, then class 2,..., in succession, they determine all the classes concurrently. Their algorithm works row by row and keeps track of all nonempty classes partially determined. Figure 4 provides a contrast of the ideas of Approaches 1 and 2. The vertical break lines correspond to "threshold positions;" the *'s correspond to points that are still be to "ranked." Other points are labelled with their ranks. Hunt and Szymanski decide the rank of a point $(i + 1, j)$ by searching on the ordered list

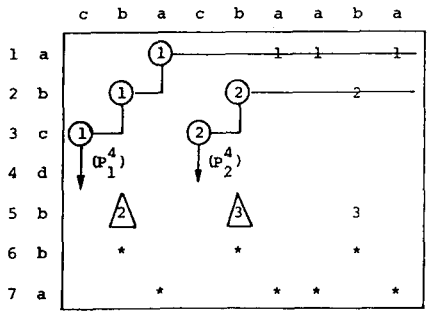
$$0 = P_0^i < P_1^i < P_2^i < \cdots < P_K^i < P_{K+1}^i = \infty.$$

to find the X, such that $P_{X-1}^i < j \leqslant P_X^i$ (hence $(i + 1, j)$ is in class X by Theorem 1). Since there are at most ρ classes, and for each point, it takes a binary search on the list of $P_K^i s$, so if there are $\mathbb{P}$ points, at most $O(\mathbb{P} \log \rho)$ time is used for computing the "ranks," or, equivalently, the length of LCS. When $\mathbb{P}$ is small, for example, $\mathbb{P} = O(n)$ in some situations [18], (i.e., when relatively few symbols of string A coincide with symbols of string B), their algorithm is very efficient—$O(n \log n)$, including the preprocessing.

Recently, we found that Hunt and Szymanski's algorithm can be improved, without sacrificing the desirable features under "sparse" situations. The idea is



a Approach 1 - at row 5, considering class 2



b Approach 2 - at row 5, considering classes 1, 2, and 3

FIG. 4. Two approaches at work.

simple, since $0 = P_0^i < P_2^i < \cdots < P_K^i < P_{K+1}^i = \infty$ for the partially determined classes $C_1^i, C_2^i, \ldots, C_K^i$, also, the points at row $(i+1)$ are kept as the ordered list $PB[\theta, 1] < PB[\theta, 2] < \cdots < PB[\theta, N[\theta]]$, where $\theta = A[i+1]$. So determining the ranks of the points at this row requires a "merge" of the above two ordered lists—the threshold positions and the j-values. Note that once we can determine the abovesaid merged sequence of $P_j^i s$ and PB_i, at most $O(\rho)$ time is needed for identifying the keys and the new thresholds for ρ classes. Hence we may concentrate on bounding the time required in doing the merge.[1]

Many alternatives may determine the merged sequence as required. For example, here are some possibilities:

(1) Given an element x_i from set 1 searches (with a binary search) set 2 to find the index j, so that $y_j < x_i \leqslant y_{j+1}$ (Fig. 5(a)).

(2) Same as (1), but reverse the direction of searches (Fig. 5(b)).

[1] It is pointed out to the authors that Hunt and McIlroy were the first to observe that a merge is required (see [17]). But it is also pointed out that we went further by using efficient merging algorithms, rather than using binary searches to accomplish the merge and restricting efficiency to special cases of "sparse matches."

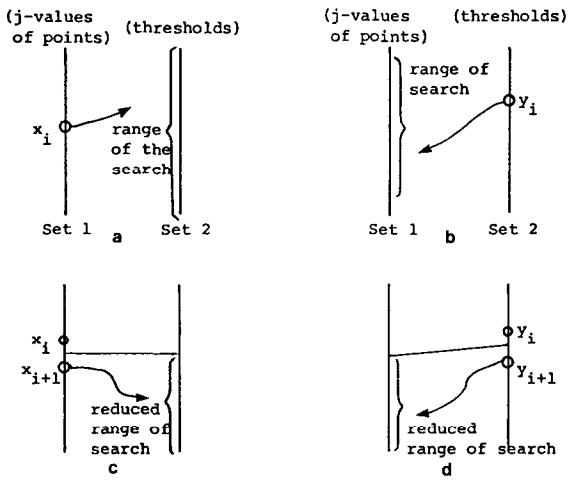


FIG. 5. Four different schemes.

(3), (4) Same as (1) and (2), respectively, but register the result of the last search, then avoid searching in vain (Figs. 5(c) and (d)).

Scheme (1) (or (2)) may characterize several approaches already known for solving the LCS problem, e.g., [17, 18, 33]. For a row with n points and ρ classes under consideration, using scheme (1) requires $O(\rho \log n)$ time, and (2) requires $O(n \log \rho)$ time. Clearly, (3) and (4) are already improvements over (1) and (2), respectively, because of better utilization of the information obtained in the searches. It would be interesting to investigate other alternatives and analyze the effects to the performance. But now that a *merge* is required, we can save by using the *most* efficient *merging algorithms* that are already existing, e.g., [5, 19, 29], and obtain a definite improvement over any of the preceding alternatives. Note that, in the context of the LCS problem, what we require from a "merge" is actually the information regarding the relative ordering of $PB[\theta, X]$'s and $P_K^i s$ obtained through the comparisons of elements of the two sets, and is not the actual output of the merged sequence. Hence, it is reasonable to consider just how many comparisons a merging algorithm requires, and neglect the "move" operations that a merging algorithm is usually supposed to do.

Therefore, with n points (j-values) to "merge" with ρ classes (threshold positions) at a row, at *most* $O(\rho \log(n/\rho) + \rho)$ *time, rather than* $O(n \log \rho)$ *or* $O(\rho \log n)$ *time, is required*, provided one efficient merging method such as [5, 19, 29] is adopted.[2] Very

[2] For practical uses, simple, efficient merging methods such as a linear (tape) merge, which is already optimal for two sets with similar cardinalities [25], is suggested. For situations where sets with disparate sizes occur often, variants of binary searches (e.g., which can alternate direction of searches according to $\|S_1\| \gg \|S_2\|$ or $\|S_2\| \gg \|S_1\|$) may be better. But for unpredictable environment, Hwang–Lin merge is recommended, as it possesses the merits of the above approaches in both the above-mentioned situations (See [25]).

roughly, for m rows, the total time needed is bounded by $O(mp \log(n/p) + mp)$. Coincidently, it takes almost the same form as obtained in our (deliberately devised) Algorithm 1. Note that we did not make use of intricate, special purpose data structures, such as HIGH$[\theta]$, LOW$[i]$, which are indispensable to our first approach.

On the other hand, the first approach allows the possibility of parallel processing. Consider the following arrangement. Let processor 1 take care of class 1, processor 2 take care of class 2, and so on. With processor $(i+1)$ lags a row behind processor i and simple communications (with LOW$[i]$ being a common variable), each processor can determine the class. So $O(m \log(n/m) + m)$ time at most is required. Incidentally, there are also parallel merging algorithms, which may speedup our second approach.

IV. Discussions

The worst case time complexity of our Algorithm 1 is shown to be $O(pm \log\lfloor n/m) + pm)$ (Theorem 2, Appendix). When $n = m$, this is $O(pn)$, the same as that in [14]. When $n > m$, the complexity can be expressed as $O(pm \log(n/m))$. Let $\alpha = n/m > 1$, our bound is better off by a factor $\alpha/\log \alpha$, which can be arbitrarily large. Interestingly, $\alpha/\log \alpha$ seems to be the direct consequence of replacing a sequential search by a binary search. But we are not able to predict when another search scheme is used.

Our algorithm also exhibits a desirable property which is now explained. By our analysis (Theorem 2, Appendix), the time complexity of our algorithm can also be expressed as $O(\mathbb{K} \log(pn/\mathbb{K}) + \mathbb{K})$, or, more loosely, $O(\mathbb{K} \log n)$, where $\mathbb{K}$ being the number of all *keys* (the *minimal K-candidates*, as defined in [14]). Hence if the number of matches is rather few ("sparse" matches) our algorithm is *very* efficient. It even outperforms the $O(\mathbb{P} \log n)$ algorithm in the same sparse situations, since $\mathbb{K} \leqslant \mathbb{P}$ (total number of matches). Furthermore, even in some very "dense" situation (which is not uncommon, if two objects represented by the strings are very much similar), such as in Fig. 6, our algorithm is still very efficient, as the running time is determined by the number of keys ($\mathbb{K}$) which does not increase the same way as $\mathbb{P}$ does.

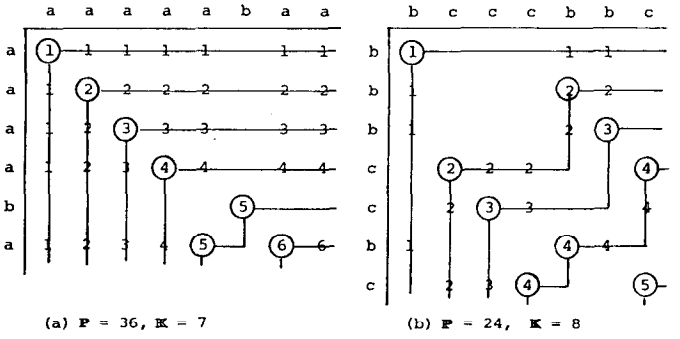


FIG. 6. "Dense" matrices.

Recall that in our algorithm, the binary searches are initiated only when a key is encountered; otherwise, only very simple bookkeeping operations are performed. But the $O(\mathbb{P} \log n)$ algorithm requires a binary search for *every* point, hence it may become one that requires as much as $O(mn \log n)$ time.

Up to the present, the best asymptotic upper bound is due to Masek and Paterson, in [30]. The approach is basically that of Arlazarov *et al.* [4], usually known as the Four Russians, in computing transitive closures of Boolean matrices. They proved that if an $n \times n$ matrix was split up into submatrices of a certain size and computations on all possible submatrices of this size were first performed, the problem could be solved using $O(n^3/\log n)$ operations, instead of $O(n^3)$ operations of the simple common approach. Likewise, Masek and Paterson's algorithm works by splitting up the computation of the "L matrix", mentioned in the dynamic programming approach, into many smaller computations, then putting them together to get the final result. The time complexity of this algorithm is $O(n \cdot \max(1, m/\log n))$ compared with ours of $O(\rho m \cdot \max(1, \log(n/m)))$ (Corollary 2.1, Appendix). Our algorithm may be comparable when ρ is relatively small. It should be cautioned that their time complexity is based on the logarithmic cost criterion, rather than on the usual uniform cost criterion. Also, the above bound applies only asymptotically—when the lengths of the strings are a great many times the size of the alphabet. Hence, the performance of an actual implementation of this method cannot beat that of the previously mentioned approaches in the usual situations. Because the overhead of precomputing all the submatrices often will not be compensated in the later "referencing" phase. Nonetheless, the approach *per se* is interesting enough, besides, the bound is the best.

Aho *et al.* [1, 2] investiagted the lower bound achievable for a class of LCS algorithms. They showed that, under a slightly restricted model of computation, where the "equal-not-equal" comparisons (is $a = b$?) of two symbols were the only information sources, every algorithm takes at least $O(mn)$ operations to solve the LCS problem. In this sense, Wagner and Fischer's algorithm [40] is already optimal. Hence, we can do no better than $O(nm)$ without using extra information. Wong and Chandra in [42] derived a similar lower bound for the string editing problem, which is also applicable to the LCS problem. But their model of computation is also restricted, hence the bound need not apply to all algorithms.

For many conceivable applications, it is reasonable to assume a linear ordering on the symbol set. This provides extra information, as it makes useful operations such as test-for-greater comparisons (is $a \geqslant b$?) possible. Recall that the string identification problem, mentioned in Section II, can be solved in $O(n \log n)$ time rather than in $O(n^2)$, when an ordering of symbols is assumed, n being the length of the string. When the symbol set is further assumed to be known beforehand, then only $O(n)$ time is required. Now, for the LCS problem, with the symbol set given, the best algorithm requires $O(nm/\log n)$, but the best lower bound for the algorithms under this assumption is $O(n)$ [42]; some investigation in filling the gap still seems profitable [30].

Our measure of "efficiency" of an algorithm was necessarily in terms of worst case

(rather than average case) time complexity (rather than space complexity). The reason for this lack of generality is due to the absence of "average case" literature, and to the fact that "time" is the relatively more valuable resource. Indeed, to define a meaningful probability distribution for the inputs is considered a difficult problem [2]. Specific applications may justify further studies.

APPENDIX

Now we analyze the worst-case time complexity of Algorithm 1: The loop from Step 2 to Step 6 is executed $\rho + 1$ times, ρ being the number of nontrivial classes. Step 2 takes $O(m)$ per execution. The total time is therefore bounded by $O(\rho m)$. Steps 3–5 take constant time per execution, hence bounded by $O(\rho)$.

Step 6.1 takes $O(\rho m)$ totally; the test in Step 6.2 takes similar amounts of time. Step 6.3 is executed at most ρm times (totally), thus bounded by $O(\rho m)$. It takes similar amounts of time to decide which one of the three cases in Step 6.4 is to be executed.

When Case 1 is encountered, only constant time is consumed, and therefore $O(\rho m)$ time at most in total. Similarly, the total time for Case 2 is $O(\rho m)$. Case 3 needs some careful treatment.

Let X_e be the range (upon which a binary search is initiated) when Case 3 is encountered the eth time, and q_k be the total number of occurrences of Case 3 when considering C_K. As our discussion following Algorithm A1 reveals, the ranges of separate binary searches are nonoverlapping, hence

$$X_1 + X_2 + \cdots + X_{q_k} \leqslant N(\theta_1] + N[\theta_2] + \cdots + N[\theta_s] \leqslant n,$$

where n is the number of all j-values and s is the number of all distinct symbols in string A ($N[\theta] = 0$ if θ does not appear in B).

Note that a binary search is done if and only if a key is present, so

$$q_k = \|\mathrm{KEY}(K)\|, \qquad \text{where} \quad \|X\| \text{ denotes the cardinality of set } X.$$

Now, the time complexity of a binary search can be measured by the number of comparisons done in the worst case. Let W_k denote the total number of comparisons devoted to the binary searches when considering C_K, then[3]

$$\begin{aligned} W_k &\leqslant \lceil \log(x_1 + 1) \rceil + \lceil \log(x_2 + 1) \rceil + \cdots + \lceil \log(x_{q_k} + 1) \rceil \\ &= \lfloor \log x_1 \rfloor + 1 + \lfloor \log x_2 \rfloor + 1 + \cdots + \lfloor \log x_{q_k} \rfloor + 1 \\ &\leqslant (\log x_1 + \log x_2 + \cdots + \log x_{q_k}) + q_k. \end{aligned}$$

[3] All logarithms are base 2.

By simple application of Lagrange multiplier method [43], it can be shown that $(\log x_1 + \log x_2 + \cdots + \log x_{q_k})$ achieves a maximum when all x_i's are equal. By setting $x_i = n/q_k$, we have,

$$W_k \leqslant q_k \log \frac{n}{q_k} + q_k.$$

Thus, in the worst case, for ρ classes

$$W \equiv \sum_{1 \leqslant k \leqslant \rho} W_k = \sum_{1 \leqslant k \leqslant \rho} \left(q_k \log \frac{n}{q_k} + q_k \right).$$

Letting $\sum_{1 \leqslant k \leqslant \rho} q_k = \mathbb{K}$ (total number of keys),

$$W \leqslant \mathbb{K} \log n + \mathbb{K} - \sum_{1 \leqslant k \leqslant \rho} q_k \log q_k.$$

Again, the Lagrange multiplier method can be used to show that $\sum_{1 \leqslant k \leqslant \rho} q_k \log q_k$ is minimum when all q_i's are equal, i.e., $q_k = \mathbb{K}/\rho$, for all k, therefore

$$\begin{aligned} W &\leqslant \mathbb{K} \log n + \mathbb{K} - \rho \cdot \frac{\mathbb{K}}{\rho} \cdot \log \frac{\mathbb{K}}{\rho} \\ &= \mathbb{K} \log \frac{\rho n}{\mathbb{K}} + \mathbb{K}. \end{aligned} \tag{B}$$

Thus, in terms of the number of keys ($\mathbb{K}$) and the length of LCS (ρ), *we may bound all the work done in Case* 3 *by* $O(\mathbb{K} \log(\rho n/\mathbb{K}) + \mathbb{K})$, *or* $O(\mathbb{K} \log n)$, *since* $\rho \leqslant m \leqslant n$.

In order to measure the work in terms of only ρ, m, and n, we reconsider the relation in (A)

$$W_k \leqslant q_k \log \frac{n}{q_k} + q_k.$$

The right-hand side $(q_k \log(n/q_k) + q_k)$ monotonically increases as q_k goes from 1 to $n \cdot \exp(\ln 2 - 1)$—let q^* denote this value—then it decreases again as q_k goes from q^* to n.

Note that q_k, the number of keys in a class, cannot exceed m, as each row contains at most one key for any class (cf. Fig. 2), i.e., $q_k \leqslant m \leqslant n$. Figure 7 depicts the curve for $q \log(n/q) + q$ versus q. From the figure we see that

(1) when $m \leqslant n \cdot \exp(\ln 2 - 1)$,

$$W_k \leqslant q_k \log \frac{n}{q_k} + q_k \leqslant m \log \frac{n}{m} + m; \tag{C}$$

(2) when $n \cdot \exp(\ln 2 - 1) < m \leqslant n$,

$$W_k \leqslant q^* \log \frac{n}{q^*} + q^* < 1.07n < 1.45m. \tag{D}$$

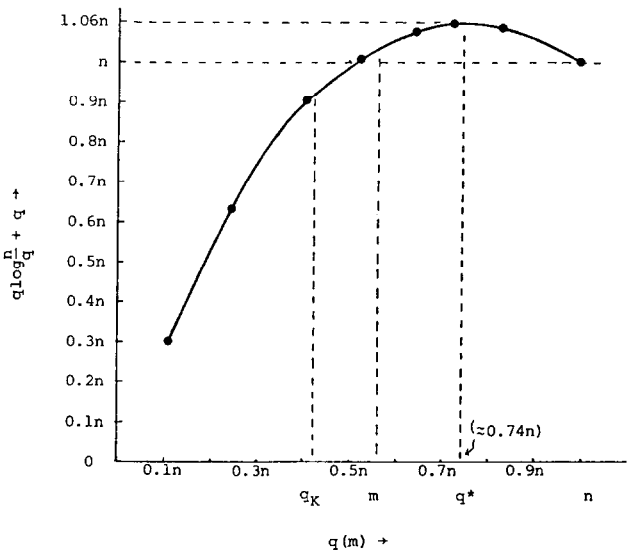


FIG. 7. $q \log(n/q) + q$ versus q.

Combining (C) and (D), we have

$$W_k = O\left(m \log \frac{n}{m} + m\right)$$

and for ρ classes,

$$W = O\left(\rho m \log \frac{n}{m} + \rho m\right).$$

THEOREM 2. *The worst-case time complexity of Algorithm* 1 *may be expressed as* $O(\mathbb{K} \log(n/\mathbb{K}) + \mathbb{K})$ *or* $O(\rho m \log(n/m) + \rho m)$. *The storage space required can be bounded by* $O(\mathbb{K})$, *or* $O(\rho m)$, *if linked list structure is adopted for keeping* KEY(K).

COROLLARY 2.1. *The time complexity of Algorithm* A1 *can be expressed as* $O(\rho m \cdot \max(1, \log(n/m))$ *or* $O(m^2 \cdot \max(1, \log(n/m))$.

ACKNOWLEDMENTS

The referees (anonymous) pointed out several flaws in our previous expositions and indicated possible enhancements, which helped us greatly. We are indebted to them.

REFERENCES

1. A. V. AHO, D. S. HIRSCHBERG, AND J. D. ULLMAN, Bounds on the complexity of the maximal common subsequence problem, *in* "Proceedings, 15th Ann. IEEE Sympos. on Switching and Automata Theory, 1974," pp. 104–109.
2. 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** (1) (1976), 1–12.
3. A. V. AHO, J. E. HOPCROFT, AND J. D. ULLMAN, "The Design and Analysis of Computer Algorithms," 2nd ed., Addison–Wesley, Reading, Mass. 1976.
4. 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 [Russian]; *Soviet Math. Dokl.* **11** (5) (1970), 1209–1210.
5. M. R. BROWN AND R. E. TARJAN, A fast merging algorithm, *J. Assoc. Comput. Mach.* **26** (2) (1979), 211–266.
6. V. CHVATAL, D. A. KLARNER, AND D. E. KNUTH, Selected combinatorial research problems, Technical Report No. STAN–CS–72–292, p. 26, Stanford Univ., Stanford, Calif., 1972.
7. V. CHVATAL AND D. SANKOFF, Longest common subsequences of two random sequences, Technical Report No. STAN–CS–75–477, Stanford Univ., Stanford, Calif., Jan. 1975.
8. M. O. DAYHOFF, Computer aids to protein sequence determination, *J. Theoret. Biol.* **8** (1965), 97–112.
9. M. O. DAYHOFF, Computer analysis of protein evolution, *Sci. Amer.* **221** (1) (1969), 86–95.
10. M. L. FREDMAN, On computing length of the longest increasing subsequences, *Discrete Math.* **11** (1) (1975), 29–36.
11. K. S. FU AND B. K. BHARGAVA, Tree systems for syntactic pattern recognition, *IEEE Trans. Comput.* **C-22** (12) (1973), 1087–1099.
12. J. GALLANT, D. MAIER, AND J. A. STORER, On finding minimal length superstrings, *J. Comput. System Sci.* **20** (1980), 50–58.
13. D. S. HIRSCHBERG, A linear space algorithm for computing maximal common subsequences, *Comm. ACM* **18** (6) (1975), 341–343.
14. D. S. HIRSCHBERG, Algorithms for the longest common subsequence problem, *J. Assoc. Comput. Mach.* **24** (4) (1977), 664–675.
15. D. S. HIRSCHBERG, "The Longest Common Subsequence Problem," Ph.D. thesis, Princeton Univ. Princeton, N. J., 1975.
16. W. J. HSU AND M. W. DU, "A Fast Algorithm for the Longest Common Subsequence Problem," Yearly Report for NSC support, March 1982.
17. J. W. HUNT AND M. D. MCILROY, "An Algorithm for Differential File Comparison," Computing Science Technical Report No. 41, 197.
18. J. W. HUNT AND T. G. SZYMANSKI, A fast algorithm for computing longest common subsequences, *Comm. ACM* **20** (5) (1977), 350–353.
19. F. K. HWANG AND S. LIN, A simple algorithm for merging two disjoint linearly ordered sets, *SIAM J. Comput.* **1** (1972), 31–39.
20. S. T. ITOGA, The string merging problem, *BIT* **21** (1981), 20–30.
21. B. W. KERNIGHAN AND L. L. CHERRY, A system for typesetting mathematics, *Comm. ACM* **18** (3) (1975), 151–157.
22. D. E. KNUTH, Big omicron and big omega and big theta, *SIGACT News*, **8** (2) (1976), 18–24.
23. D. E. KNUTH, J. H. MORRIS, AND V. R. PRATT, "Fast Pattern Matching Algorithms," Technical Report No. STAN–CS–74–440, Computer Science Dept., Stanford Univ., Stanford, Calif., 1974.
24. D. E. KNUTH, "The Art of Computer Programming, Vol. 1: Fundamental Algorithms," Addison–Wesley, Reading, Mass., 2nd ed., 1973.
25. D. E. KNUTH, "The Art of Computer Programming, Vol. 3: Sorting and Searching," Addison–Wesley, Reading, Mass., 1973.

26. R. LOWRANCE AND R. A. WAGNER, An extension of the string to string correction problem, *J. Assoc. Comput. Mach.* **22** (2) (1975), 177–183.
27. S. Y. LU AND K. S. FU, A sentence-to-sentence clustering procedure for pattern analysis, *IEEE Trans. Systems Man Cybernet.* **SMC-8** (5) (1978), 381–389.
28. D. MAIER, The complexity of some problems on subsequences and supersequences, *J. Assoc. Comput. Mach.* **25** (2) (1978), 322–336.
29. G. K. MANACHER, Significant improvements to the Hwang–Lin merging algorithm, *J. Assoc. Comput. Mach.* **26** (3) (1979), 434–440.
30. W. J. MASEK AND M. S. PATERSON, A faster algorithm computing string edit distances, *J. Comput. System Sci.* **20** (1980), 18–31.
31. H. L. MORGAN, Spelling correction in systems programs, *Comm. ACM* **13** (2) (1970), 90–94.
32. J. H. MORRIS AND V. R. PRATT, "A Linear Pattern-Matching Algorithm," Technical Report No. TR–40, Comput. Center, U. of California, Berkeley, Calif., 1970.
33. A. MUKHOPADHYAY, A fast algorithm for the longest-common-subsequence problem, *Inform Sci.* **20** (1980), 69–82.
34. S. B. NEEDLEMAN AND C. S. WUNSCH, A general method applicable to the search for similarities in the amino acid sequence of two proteins, *J. Molecular Biol.* **48** (1970), 443–453.
35. D. SANKOFF, Matching sequences under deletion insertion constraints, *Proc. Nat. Acad. Sci. U.S.A.* **69** (1972), 4–6.
36. D. SANKOFF AND R. J. CEDERGREN, A test for nucleotide sequence homology, *J. Molecular Biol.* **77** (1973), 159–164.
37. S. M. SELKOW, The tree-to-tree editing problem, *Inform. Process. Lett.* **6** (6) (1977), 184–186.
38. P. H. SELLERS, An algorithm for the distance between two finite sequences, *J. Combin. Theory Ser. A* **16** (1974), 253–258.
39. R. A. WAGNER, Common phrases and minimum-space text storage, *Comm. ACM* **16** (3) (1973), 148–152.
40. R. A. WAGNER AND M. J. FISCHER, The string-to-string correction problem, *J. Assoc. Comput. Mach.* **21** (1) (1974), 168–173.
41. P. A. WAGNER, On the complexity of the extended string-to-string correction problem, *in* Proceedings, 7th Ann. ACM Sympos. on Theory of Comput. 1975, pp. 218–223.
42. C. K. WONG AND A. K. CHANDRA, Bounds for the string editing problem, *J. Assoc. Comput. Mach.* **28** (1) (1976), 13–18.
43. F. S. HILLIER AND G. J. LIEBERMAN, "Introduction to Operations Research," Holden-Day, San Francisco.