BIT 24 (1984), 45–59

COMPUTING A LONGEST COMMON SUBSEQUENCE FOR A SET OF STRINGS

W. J. HSU and M. W. DU

Institute of Computer Engineering, National Chiao Tung University, 1001 Ta Hsueh Road, Hsinchu, Taiwan, Republic of China

Abstract.

The known 2-string LCS problem is generalized to finding a Longest Common Subsequence (LCS) for a set of strings. A new, general approach that systematically enumerates common subsequences is proposed for the solution. Assuming a finite symbol set, it is shown that the presented scheme requires a preprocessing time that grows linearly with the total length of the input strings and a processing time that grows linearly with (K), the number of strings, and ($\|\mathbb{P}\|$) the number of matches among them. The only previous algorithm for the generalized LCS problem takes $O(K) \cdot |S_1| \cdot |S_2| \cdot \ldots \cdot |S_K|)$ execution time, where $|S_i|$ denotes the length of the string S_i. Since typically $\|\mathbb{P}\|$ is a very small percentage of $|S_1| \cdot |S_2| \cdot \ldots \cdot |S_K|$, the proposed method may be considered to be much more efficient than the straightforward dynamic programming approach.

Key Phrases: Analysis of Algorithms, String Merging Problems, LCS Problem.

1. Introduction.

A string is a sequence of symbols. The set of all possible symbols is called the alphabet, denoted Σ. We assume that the alphabet is of bounded size, i.e., $\|\Sigma\| \leqslant C$ for some constant C, where $\|X\|$ denotes the cardinality of a set X. A *subsequence* of a string is obtained by deleting 0 or more (not necessarily consecutive) symbols from the string. For example, 'cbcbd' is a string, and 'ccb' is a subsequence of it. A common subsequence (CS) of a set of strings is a subsequence of all the 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, a set of strings may possess more than one LCS. For example, both 'abd' and 'acd' are LCSs of 'abcd' and 'acbd'. The K-string LCS problem (for short, 'K-LCS problem') is to find a single LCS for K arbitrary input strings.

The 2-LCS problem was first studied by the molecular biologists (e.g., [6, 7]) who found use of LCS in studying similar sequences of amino acids. Subsequently, together with other string problems, the complexity of the 2-LCS problem was analyzed by many computer scientists (see the references). Other cited applications include data compression [1, 4, 5, 14, 29], syntactic pattern

---

Received March 1983. Revised August 1983.

recognition [21], etc. [8, 24, 28], where an LCS of the strings is used to denote a maximal common substructure of the objects represented by the strings. Closely related problems concern the computations of string-to-string distance (string editing) [20, 23, 26, 30, 31], shortest common supersequences [10, 22], string merging [16], pattern matching [17], and tree-to-tree distance [9, 27].

The general K-LCS problem was first addressed in [10, 22], also in [16] as a special type of the string merging problems. In [16], Itoga generalized the dynamic programming approach in [30] to solving the K-LCS problem. The execution time which is required by his algorithm is bounded by $O(K \cdot |S_1| \cdot |S_2| \cdot \ldots \cdot |S_K|)$, where $|S_i|$ denotes the length of the string $S_i (1 \leqslant i \leqslant K)$.

In Section 2, the general LCS problem is first formulated as identifying a longest path in some acyclic directed graph, then special properties of the LCS problem are pointed out to allow more efficient operations. Our scheme is presented after studying a computational model called 'common subsequence tree', whereas suitable data structures are studied in the following section. The result is that whenever the matches among the strings are scarce, i.e., $\|\mathbb{P}\| \ll (|S_1| \cdot |S_2| \cdot \ldots \cdot |S_K|)$, as is the usual case, the new method presented here is very efficient, in terms of both time and space. Comparisons with other recent results are provided in the last part.

To put our subject in perspective, in the following we briefly review some existing algorithms.

The 2-LCS problem has been solved by using dynamic programming, e.g., [11, 26, 30]. The idea is derived from the following observation. Letting $A[1 \ldots m]$ and $B[1 \ldots n]$ denote the strings $a_1 a_2 \ldots a_m$ and $b_1 b_2 \ldots b_n$, and $L_{i,j}$ be the length of LCS of $A[1 \ldots i]$ and $B[1 \ldots j]$, we have

$$\text{If } a_i = b_j \text{ then } L_{i,j} = L_{i-1,j-1} + 1$$

$$\text{else } L_{i,j} = \max \{L_{i,j-1}, L_{i-1,j}\}.$$

$$\text{Clearly, } L_{i,0} = 0 \quad \text{for } i = 0, 1, 2, \ldots, \text{ and}$$
$$L_{0,j} = 0 \quad \text{for } j = 0, 1, 2, \ldots$$

This boundary condition simply means that the length of LCS of any string and a null string is zero.

In [30], Wagner and Fischer employ an array $L[0 \ldots m, 0 \ldots n]$ 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, $A = abcdb$, $B = cbacbaa$. After computing $L_{i,j}$, the L matrix is as shown in Fig. 1,

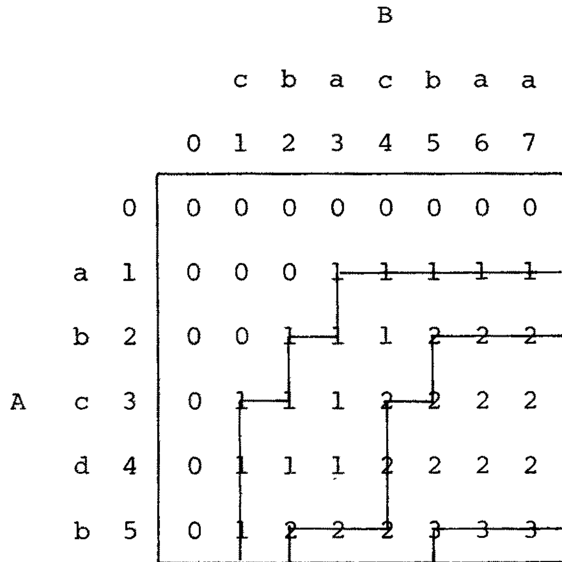


Fig. 1. L matrix.

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 are shown in the figure. In this example, the length of LCS is in $L[5, 7]$, which is 3.

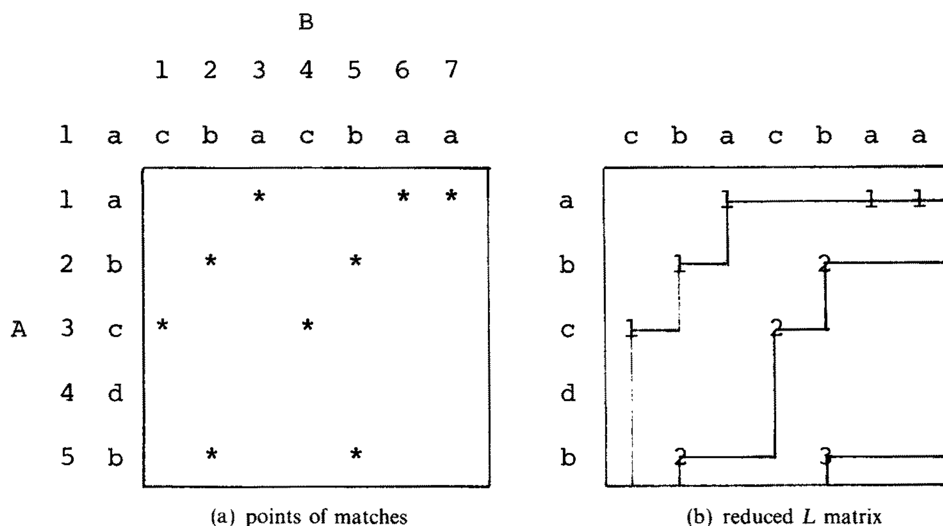
Since all the entries in the L matrix are calculated, and computing each entry requires a constant amount of time, the total time required in computing the length of LCS is bounded by $O(m \cdot n)$, m, n being the lengths of the two strings. An LCS can be generated in $O(m+n)$ time after computing the matrix, so the total time is bounded by $O(m \cdot n)$.

The storage space required in this algorithm is also bounded by $O(m \cdot n)$. In [11] Hirschberg improves the space complexity to $O(m+n)$. But the execution time remains the same.

As pointed out in [1], the straightforward matrix-filling approach for the LCS problem has the disadvantage that it takes the *same* large amount of time on all input strings with the same lengths ($|S_1|, |S_2|, \ldots, |S_K|$), as the algorithm makes no use of the inherent structures of the strings. Therefore we shall look for an algorithm which is more efficient on typical inputs.

Observing that a CS is composed of only matched symbols, we have a possibility. Consider the following diagram (Fig. 2a). By the above argument, we need only consider the matches (*) in the diagram, instead of dealing with all the $m \cdot n$ entries. Note that usually the number of such matches is just a small percentage of $|S_1| \cdot |S_2| \cdot \ldots \cdot |S_K|$.

Consider again the L matrix of the dynamic programming approach now with non-matches omitted (Fig. 2b). Since only matched symbols will constitute an LCS, if we can somehow compute the retained L-values on the matched



(a) points of matches

(b) reduced L matrix

Fig. 2. Matrices with reduced number of entries.

points, we lose no necessary information for generating an LCS. Several algorithms [12–15, 25] for the 2-LCS problem compute some representation of the reduced L matrix. The (worst case) time bound is lowered down to $O(\varrho m \cdot \max(1, \log(n/m)))$, where $m, n (m \leqslant n)$ denote the lengths of the two input strings and ϱ denotes the length of the computed LCS [13].

Unfortunately, one finds difficulties in generalizing the above algorithms to the general K-LCS problem, because these algorithms depend on certain properties unique to the special 2-string cases. More explicitly, the points to be labelled with the same L value are on a "contour" (cf. Fig. 2b) which is easy to define row-by-row or column-by-column. But this contour-defining approach when generalized to higher dimensions ($K > 2$) requires much book-keeping work, hence will, very likely, yield poor time performance and induce great space demand. Consequently, a new, general approach which differs from those mentioned above is pursued.

A note is in order. It has been shown in [10, 22] that the K-LCS problem is NP-complete for variable-length strings and (implicitly) unbounded K, no matter whether the size of the alphabet is bounded or not. Therefore, it is very probable that *every* solution to the K-LCS problem, *in the worst case*, takes an amount of execution time which is an exponential function of the 'input size', taking the lengths of the strings and the number of strings (K) as parameters.

However, for almost every conceivable application, it is also reasonable to assume that the number of the strings under consideration is bounded by some constant. In this sense, the straightforward (generalized) dynamic programming algorithm of [16] already has a polynomial time complexity: $O(K \cdot |S_1| \cdot |S_2| \cdot \ldots \cdot |S_K|)$

$= O(|S_X|^K)$ where $|S_X| = \max\{|S_1|, |S_2|, \ldots, |S_K|\}$, since K is now considered as a constant.

In other words, in typical applications we expect the number of strings under consideration to be bounded by a constant and therefore a solution of the generalized LCS problem for typical applications to be a polynomial time algorithm.

2. A new approach.

Associated with a given set of strings $S_1, S_2, \ldots, S_K$, a *matched point* (a *match*, a *point*) is a K-tuple $(i_1, i_2, \ldots, i_K)$ which denotes two things:

1. a match of the symbols at some positions of the strings, and
2. the matched symbol θ. I.e., $\theta = S_1[i_1] = S_2[i_2] = \ldots = S_K[i_K]$.

A point $P = (i_1, i_2, \ldots, i_K)$ is said to be a *successor* of another point $P' = (i'_1, i'_2, \ldots, i'_K)$, if $i_1 > i'_1$, $i_2 > i'_2, \ldots$, and $i_K > i'_K$. Alternatively, we say that P' is a *predecessor* of P. Clearly, a *CS* of $S_1, S_2, \ldots, S_K$ corresponds to a chain of predecessor-successors and vice versa, and an LCS of the strings corresponds to a longest such chain. Letting $\mathbb{P}$ denote the set of all matched points, the above successor-predecessor relation is a partial ordering [18] defined on $\mathbb{P}$. We may represent the relation by an acyclic digraph having all arcs of length 1. According to the above argument, an LCS corresponds to a maximal length path on the graph, as first observed by Mukhopadhyay in [25].

For convenience we further introduce two dummy matches $\bar{0} = (0, 0, \ldots, 0)$ and $\bar{\infty} = (\infty, \ldots, \infty)$ into $\mathbb{P}$. Since all the other points in $\mathbb{P}$ are successors to $\bar{0}$ and predecessors to $\bar{\infty}$, the above graph is now converted to one with a single 'source' ($\bar{0}$) and a single 'sink' ($\bar{\infty}$) as encountered in many network analysis problems. Indeed, determining an LCS can now be thought of as identifying a *critical path* from $\bar{0}$ to $\bar{\infty}$, as in project-scheduling problems. A critical path for AOE (activity-on-edge) networks can be computed by well-known algorithms, [33], requiring only $O(V+E)$ execution time where V denotes the number of vertices (nodes) and E denotes the number of edges (*arcs*) on the network.

However, depending on the input strings, the number of *arcs* on the digraph representing the partial order relation of the matched points may be as large as the order of $||\mathbb{P}||^2$, $||\mathbb{P}||$ being the number of points, or vertices. Thus, using the known algorithms the required processing time is at least of the order $(||\mathbb{P}||^2)$. Hence in the following we shall deviate from the known graph (critical path) algorithms.

Now we are concerned with a longest path from $\bar{0}$ to $\bar{\infty}$. For a matched point P a longest path from P to $\bar{\infty}$ is of interest. Let R_p denote the length of a longest path from P to $\bar{\infty}$. Clearly, from the above definition, $R_{\bar{\infty}} = 0$, and $R_{\bar{0}} =$ (length of LCS)-1. It is also easy to see that

LEMMA 1

$R_p = \max\{R_{p'} | P'$ is a successor of $P\} + 1$.

As a matter of fact, in determining R_p (the 'R-value') of a point P, not all the successors of P are relevant. We shall characterize a subset of all the successors of a point P so that R_p can be determined from the R-values of these selected successors. To be specific,

$R_p = \max\{R_{p'} | P'$ is a selected successor of $P\} + 1$.

Consider the following property of R-values:

Property (1) Given two points $P' = (i'_1, i'_2, \ldots, i'_K)$ and $P'' = (i''_1, i''_2, \ldots, i''_K)$, where $i'_j \leqslant i''_j$ for $1 \leqslant j \leqslant K$, then $R_{p'} \leqslant R_{p''}$.

PROOF

Every successor of P'' must also be a successor of P', by Lemma 1, and the result follows. ■

For convenience, if a successor of a point P corresponds to a symbol θ, we call it a *θ-successor* of P. Also, for any θ, we define $\bar{\infty}$ to be a *θ-successor* of any other point, while $\bar{\infty}$ has no successor itself.

Now, if the above P' and P'' are two θ-successors to a point P for which R_p is to be determined, then P'' can be ruled out by P', since $R_{P''} \leqslant R_{P'} \leqslant R_P$.

Hence, out of all θ-successors of P, we need only consider *the* one (see the following) which is not ruled out by any other θ-successor of P.

The "*immediate*" *θ-successor* of a point $P = (i_1, i_2, \ldots, i_K)$ $(P \neq \bar{\infty})$ is defined as follows:

$P' = \bar{\infty}$ if $\{i | S_j[i] = \theta$ and $i_j < i \leqslant |S_j|\} = \phi$ for some j $(1 \leqslant j \leqslant K)$. Otherwise $P' = (i'_1, i'_2, \ldots, i'_K)$ where $i'_j = \min\{i | S_j[i] = \theta$, and $i_j < i \leqslant |S_j|\}$ $(1 < j \leqslant K)$.

By the above definition, the immediate θ-successor of a point $P = (i_1, i_2, \ldots, i_K)$ represents the next θ that is closest to i_j in string j $(1 \leqslant j \leqslant K)$, whereas $\bar{\infty}$ simply denotes that no such θ exists. In Section 3, we shall discuss an efficient method for generating an immediate successor for any point under consideration.

Now, by Property (1), since all the other θ-successors of P, if any, have smaller (or equal) R values than that of the immmediate θ-successor, it follows:

LEMMA 2

If $\theta_1, \theta_2, \ldots, \theta_t$ are the distinct symbols which are common to the set of strings, and $P_1, P_2, \ldots, P_t$ are the corresponding immediate successors of P, then $R_P = \max\{R_{P_1}, R_{P_2}, \ldots, R_{P_t}\} + 1$.

Hence, to compute $R_{\bar{0}}$ and, thereby, the length of LCS, we may consider the R-values of all the immediate successors of $\bar{0}$. If the R-value of a selected successor is still unknown, we again consider all its immediate successors. While

if the R-values of all the immediate successors of a point P are known, then R_P can be determined readily, by using Lemma 2. It is easy to see that the above recursive process can indeed determine $R_{\bar{0}}$, hence the length of LCS.

To visualize the computational process, the above scheme can be represented as a tree (cf. Fig. 3), with $\bar{0}$ being the root. Each internal (nonleaf) node in the tree corresponds to a point for which the R value is to be determined, and each (directed) branch connects a parent node to a child node, the latter being the immediate θ-successor of the former. Since *each path from a node (a match) to its descendent node (a match) corresponds to a CS (common subsequence) of the strings*, such a tree will be (called) a *CS tree*.

Intuitively, a *CS* tree enumerates chains of common subsequences by catenating as many common symbols as possible, that is, by considering only *immediate* successors.

Example

Fig. 3 shows a *CS* tree associated with the given strings. It is easy to verify the immediate successor-predecessor relations. Also, by observing that a longest path in the tree corresponds to an LCS, our problem is solved. In this case, $R_{\bar{0}}$ is 4, and acb, bcb, cbb are all LCSs of the strings.

For simplicity, we initially stipulate that all internal nodes have exactly t children. That is, for the t distinct symbols which are common to all the strings, we generate t immediate successors for a point with undefined R value. In other words, a *CS* tree is t-ary. The following property is taken from [18].

LEMMA 3.

A t-ary tree with I internal nodes has $(t-1)I+1$ *external nodes.*

Note that since the R-value of a point need not be computed more than once, to save execution time, on a *CS* tree, each distinct point, if it ever appeared, should be on the internal node no more than once. Such a *CS* tree (with no duplicate internal nodes) is said to be *irredundant* and is abbreviated as an *ICS* tree. Letting $\|\mathbb{P}\|$ denote the total number of all matched points, it follows that

COROLLARY 3.1.

An irredundant t-ary CS tree has at most $t\|\mathbb{P}\|+1$ *nodes.*

PROOF. The number of all internal nodes of an *ICS* tree is bounded by $\|\mathbb{P}\|$. There are then at most $(t-1)\|\mathbb{P}\|+1$, external nodes. Adding these two bounds yields the result.

In Fig. 3 only $\bar{\infty}$ appeared as the leaf nodes. Since certain points (underscored) appeared more than once as the internal nodes, this *CS* tree represents an inefficient computation. Fig. 4 shows an *ICS* tree for the same job. Note that the subtrees rooted at $(5,5)$ and $(3,4)$ are pruned because the two

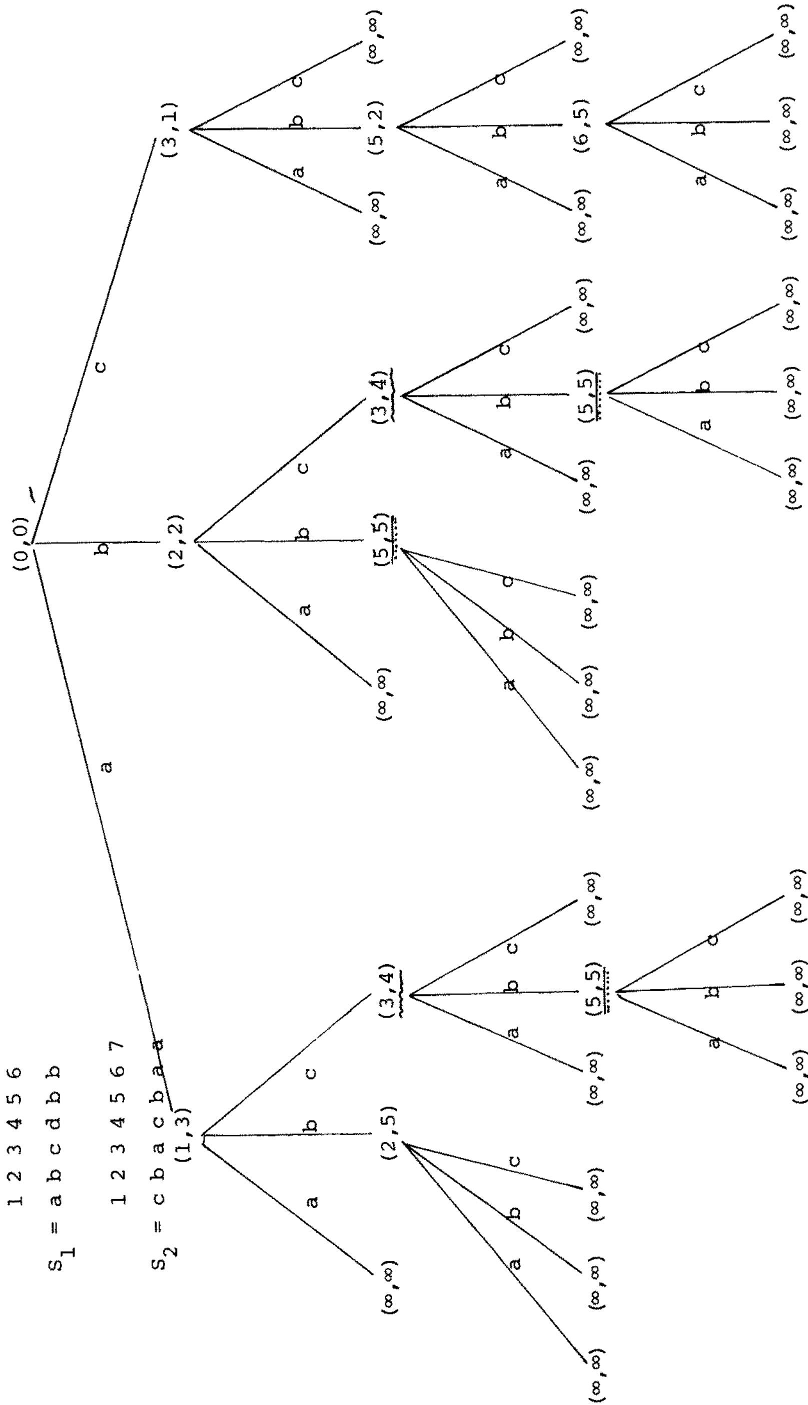


Fig. 3. A CS tree (with redundancy).

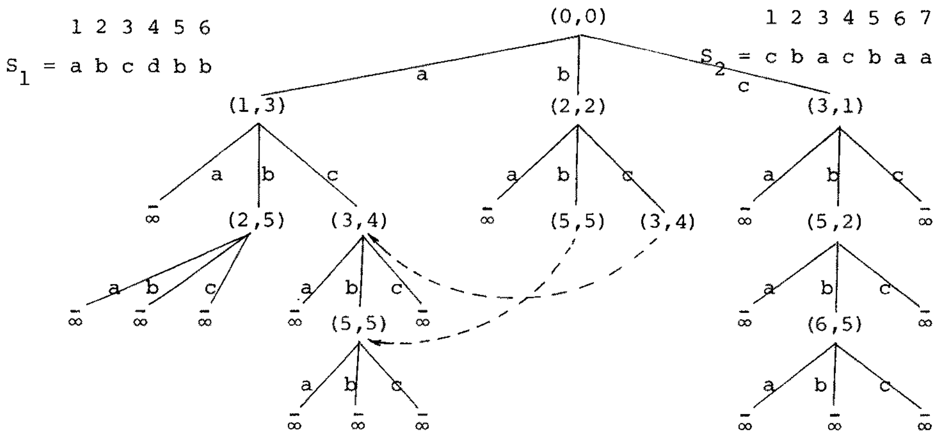


Fig. 4. An ICS tree.

nodes appeared in other places. Here we may think of the dashed lines as references to known results.

As illustrated in the examples, a systematic evaluation of the R values may further require that the branches are considered in a fixed sequence, i.e., we consider only *ordered ICS* trees.

Now, we are able to present a scheme for computing an LCS by constructing an ICS tree associated with a set of input strings.

It is natural to construct an ordered ICS tree in depth-first [18, 33] manner. Specifically, for each node (point) visited, if the R-value is not yet known, the subtree rooted with its first (left-most) child is generated recursively, then the second subtree and so on; if the R-value of a node (point) is known (i.e., the longest path from the point to $\bar{\infty}$ is known), then no further subtrees of it are generated, rather, the R-value of the point is compared with the R-values of the other *siblings* (other children of the same parent of this node), in order to determine the maximum R-value (thus a longest path to $\bar{\infty}$). Fig. 5 presents the tree of Fig. 4 when it is traversed in this order, the integer labels on the nodes corresponding to the sequence of the traversal.

For each internal node, along with its own R value, we will register the identity of one of its immediate successors which has the maximum R-value. Now, if the R-value of $\bar{0}$, the root of a CS tree, is determined, we can easily trace out an LCS by using this linkage information.

From the preceding discussion, we may safely claim that the scheme above solves the LCS problem correctly. Also, from Corollary 3.1, the total number of nodes processed is bounded by $O(t \cdot \|\mathbb{P}\|)$. So if processing each node (generating immediate successors, determining R-value, see next section) takes l time units, then an algorithm realizing the scheme above takes at most $O(l \cdot t \cdot |\mathbb{P}\|)$ processing time. In the next section, we shall examine some data structures in order to determine the parameter of performance mentioned above.

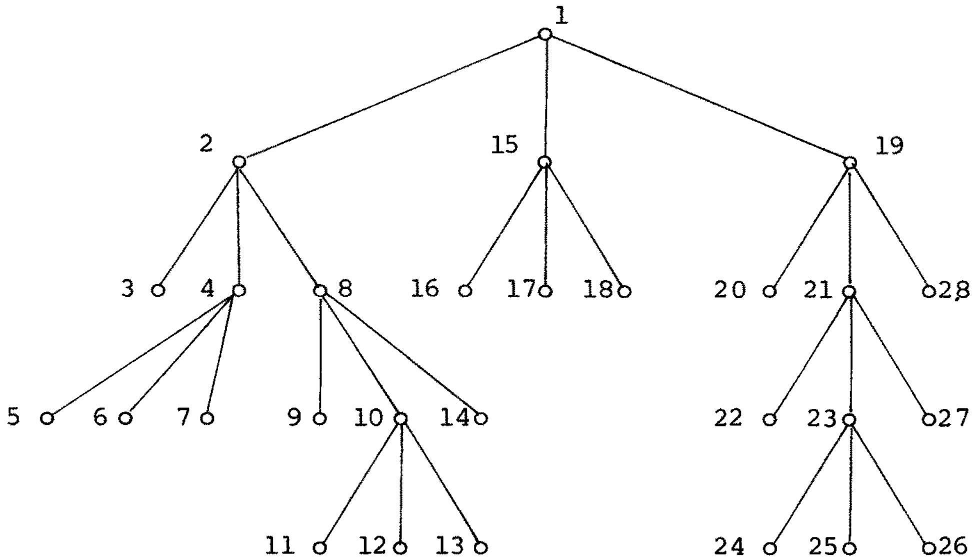


Fig. 5. A depth first search.

3. Implementation.

Provided with some simple preprocessing, we may enumerate the immediate θ-successors for any given point rather efficiently; see the following example. First recall that the jth-tuple of a matched point corresponds to a position in string S_j, and that the immediate θ-successor of a point $P = (i_1, i_2, \ldots, i_k)$, if it exists, is given by $(i'_1, i'_2, \ldots, i'_k)$, where $i'_j = \min\{i|S_j[i] = \theta \text{ and } i > i_j\}$, $(1 \leqslant j \leqslant k)$ (otherwise, the successor is given by $\bar{\infty}$).

Example

For a string S, a 'θ-list' is given by $\{i|S[i] = \theta\}$. Fig. 6 shows the positions of the strings partitioned into θ-lists. Now, for any point, say (3, 1), which is a 'c' match, we may search the two 'b'-lists, and find that (5, 2) is the immediate 'b'-successor of (3, 1).

```
      1 2 3 4 5 6
S1  = a b c d b b

      1 2 3 4 5 6 7
S2  = c b a c b a a
```

Positions in \ θ	'a'	'b'	'c'	'd'
S_1	1	2,5,6	3	4
S_2	3,6,7	2,5	1,4	nil

Fig. 6. θ-lists.

Observe that for a given position in a string, we need to know the next higher position that contains θ. Hence, to avoid the searches on the θ-lists as done in the example above, we rather prepare such 'next higher θ-positions' for each position in a string. Consider the following example.

Example

In Fig. 8a, we prepare three lists of the next higher positions for the 3 distinct symbols: a, b, and c.

The contents of the array A are determined thus:

$$A[i] = \min\{j|S[j] = \text{'}a\text{'} \text{ and } j > i\}.$$

Thus $A[1] = 4$ denotes that position 4 of the string contains the first 'a' that appeared after position 1. Similar statements apply to arrays B and C.

Preprocessing

Here is how the lists of 'next higher θ-positions' are set up:

(Step 1) First, for each symbol, identify the positions in each string containing that symbol, thus forming the θ-lists. For a string S, the 'θ-list' is given by $\{i|S[i] = \theta\}$. Again see Fig. 7b for example.

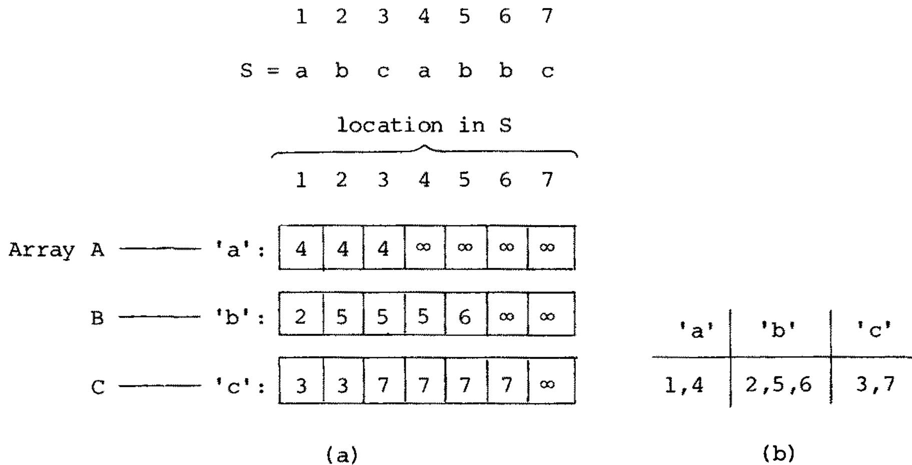


Fig. 7. Representation of 'next' θ's.

(Step 2) Now, for each position i of the string, with i running through 1, 2,..., in that sequence, scan each of the θ-lists, and write the first value in the θ-list which is greater than i into the ith position of the corresponding array (cf. Fig. 7).

Step 1 is called *string-identification* in [1]. Assuming the alphabet is known, the work can be done in $O(|S|)$ time using $O(\|\Sigma\| + |S|)$ space, where S denotes

the length of the string under 'identification' and $\|\Sigma\|$ denotes the size of the alphabet Σ. (If the symbols are not known (given) beforehand, then with the use of balanced search tree techniques [18, 33], the work requires at most $O(|S|\log t)$ time*, where t denotes the number of distinct symbols appearing in the string.) It is also easy to implement Step 2 with at most $O(|S|\cdot t)$ time and $O(|S|\cdot t)$ space. where t still represents the number of distinct symbols appearing in S.

For all K strings, the total time/space required in Steps 1, 2 are bounded by $O(L\cdot t')$ where $L = |S_1|+|S_2|+\ldots+|S_K|$, and t' is the number of the distinct symbols that are common to all the strings. Since $t' \leqslant \|\Sigma\| \leqslant C$ for some constant C, the time bound may be expressed as $O(L)$, that is, the preprocessing takes an execution time which grows linearly with the total length of the input strings.

With this preparation of the 'next higher positions', we may obtain an immediate successor (a K-tuple) of any point in an amount of time which depends only on K, the number of strings under consideration.

Also from this preprocessing, the data structure required in storing the R-value of a point can be determined. We shall use one array for the points matching a distinct symbol. Let $X_{ij} = \{l|S_i[l] = \theta_j,\ 1 \leqslant l \leqslant |S_i|\}$, which denotes all the positions in S_i that contain θ_j, then the cartesian product $\mathbb{P}_j = X_{1j}\times X_{2j}\times\ldots\times X_{Kj}$ is clearly the set of all the points matching the symbol θ_j. We may use an $\|X_{1j}\|\times|X_{2j}\|\times\ldots\times\|X_{Kj}\|$ array to store the R-values of the points in $\mathbb{P}_j$. See the following example.

EXAMPLE

Fig. 8 portrays parts of the θ-lists for some given strings.

Clearly, since $\|X_{11}\| = 4$, $\|X_{21}\| = 2$, $\|X_{31}\| = 3$, a $4\times2\times3$ array B1 can be used to store the R-values for all the points matching θ_1.

Common symbols / Strings	θ_1	θ_2	......
S_1	1,4,7,11		
S_2	3,14		
S_3	2,5,6		

Fig. 8. θ-lists.

* It is reasonable to assume that the representations of the symbols are linearly ordered so that useful operations like test-for-ordering (i.e., $x \geqslant b$?), rather than test for mere equality, ($x = b$?), may be used.

It is also clear that, in this arrangement, each point has a unique index in the array and an access to the array takes a constant amount of time. The total space required is clearly bounded by $O(\|\mathbb{P}\|)$, where $\mathbb{P} = \mathbb{P}_1 \cup \mathbb{P}_2 \cup \ldots \cup \mathbb{P}_t$. For the example above, the R-value of the point (1, 14, 2) is in $B1$ [1, 2, 1].

Concluding these discussions, the arrangements can solve the general LCS problem rather efficiently.

Main theorem

With $O(L \cdot t)$ preprocessing time, the K-string LCS problem can be solved with $O(K \cdot t \cdot \|\mathbb{P}\|)$ processing time and $O(\|\mathbb{P}\| + L \cdot t)$ space, where L denotes the total length of the strings (the input size) and t denotes the number of distinct symbols that are common to the strings under consideration. $\|\mathbb{P}\|$ is the number of K-tuples denoting the matches among the strings.

For the usual applications, the number of the strings (K) and the alphabet (Σ) are both finite, i.e., there exist some constants C_1, C_2, such that $K \leqslant C_1$ and $\|\Sigma\| \leqslant C_2$. Since $t \leqslant \|\Sigma\|$, we have the following asymptotic upper bound for the K-string LCS problem:

COROLLARY *For a constant number of strings and an alphabet of bounded size, the LCS problem can be solved with $O(L)$ preprocessing time and $O(\mathbb{P})$ processing time, where L denotes the input size and $\mathbb{P}$ denotes the number of matches.*

4. Discussion.

A simple, systematic approach for generating common subsequences for a set of strings has been proposed. Assuming a finite symbol set, it is shown that the proposed scheme requires a preprocessing time that grows linearly with the total length of the input strings and a processing time that grows linearly with K, the number of strings, and $\|\mathbb{P}\|$, the number of matches among them. The only previous algorithm for the generalized LCS problem takes $O(K \cdot |S_1| \cdot |S_2| \cdot \ldots \cdot |S_K|)$ execution time, where $|S_i|$ denotes the length of the string S_i. Since typically $\|\mathbb{P}\|$ is a very small percentage of $|S_1| \cdot |S_2| \cdot \ldots \cdot |S_K|$, the proposed method may be considered as much more efficient than the straightforward dynamic programming approach.

Now, consider the conventional 2-LCS case. We first note that in several recent works [12–15, 25] essentially the same string-identification preprocessing as done in ours is required. With the matched points represented as a set of θ-lists, Hirschberg's algorithm in [12] examines every matched point at least once, hence the time complexity of his algorithm, when expressed in terms of $\|\mathbb{P}\|$ (the number of matches), is $\Omega(\|\mathbb{P}\|)$, i.e., is bounded below by $\|\mathbb{P}\|$ asymptotically. Again, let n denote the length of the longer string, the algorithms in [14, 15, 25] require $O(\|\mathbb{P}\| \log n)$ time. Hence, asymptotically, the above algorithms are outrun by the new one. The preceding arguments apply only when t, the total number of distinct symbols that are common to the two strings, is bounded.

Fortunately, this is the usual case. Furthermore, as indicated in Section 1, the new scheme is general enough so that the K strings cases are handled with the same ease. The proposed CS trees is hence a very useful concept for the LCS problem.

Acknowledgements.

The authors are thankful for the comments given by the reviewers (anonymous), which have certainly improved the readability of the paper. They also wish to thank for the assistance generously given by their colleagues, in particular, Prof. C. G. Chung and Prof. W. T. Tsai.

REFERENCES AND BIBLIOGRAPHY

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(1) (Jan. 1976), 1–12.
2. A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, 2nd printing, Addison-Wesley, Reading, Mass., 1976.
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 transl. in Soviet Math., Dokl. 11, 5 (1970), 1209–1210.
4. V. Chavatal, D. A. Klarner, and D. E. Knuth, *Selected combinatorial research problems*, STAN-CS-72-292, Stanford Univ., Stanford, Calif. 1972, p. 26.
5. V. Chvatal and D. Sankoff, *Longest common subsequences of two random sequences*. STAN-CS-75-477, Stanford Univ., Stanford, Calif., Jan. 1975.
6. M. O. Dayhoff, *Computer aids to protein sequence determination*, J. Theoret. Biology 8, (Jan. 1965), 97–112.
7. M. O. Dayhoff, *Computer analysis of protein evolution*, Scientif. Amer. 221, 1 (July 1969), 86–95.
8. M. L. Fredman, *On computing length of the longest increasing subsequences*, Discrete Math. 11, 1 (Jan. 1975), 29–36.
9. K. S. Fu and B. K. Bhargava, *Tree systems for syntactic pattern recognition*, IEEE Trans. Computs. C-22, 12 (Dec. 1973), 1087–1099.
10. J. Gallant, D. Maier and J. A. Storer, *On finding minimal length superstrings*, J. Comput. and Sys. Sci. 20, (1980), 50–58.
11. D. S. Hirschberg, *A linear space algorithm for computing maximal common subsequences*, Comm. ACM 18(6) (June, 1975), 341–343.
12. D. S. Hirschberg, *Algorithms for the longest common subsequence problem*, J. Assoc. Comput. Mach. 24(4) (1977) 664–675.
13. W. J. Hsu and M. W. Du, *A fast algorithm for the longest common subsequence problem*, Yearly Report for NSC Support, March (1982).
14. J. W. Hunt and M. D. McIlroy, *An algorithm for Differential File Comparison*, Computing Science Technical Report 41, 197.
15. J. W. Hunt and T. G. Szymanski, *A fast algorithm for computing longest common subsequences*, Com. ACM 20(5) (May, 1977), 350–353.
16. S. Y. Itoga, *The string merging problem*, BIT 21 (1981), 20–30.
17. D. E. Knuth, J. H. Morris and V. R. Pratt, *Fast pattern matching algorithms*, Technical Report STAN-CS-74-440, Computer Science Dpet., Stanford Univ., Aug. (1974).
18. D. E. Knuth, *The Art of Computer Programming*, Vol. 1: *Fundamental Algorithms*, Addison-Wesley, Reading, Mass., Sec. ed., (1973).
19. D. E. Knuth, *The Art of Computer Programming*, Vol. 3: *Sorting and Searching*, Addison-Wesley, Reading, Mass., (1973).

20. R. Lowrance and R. A. Wagner, *An extension of the string to string correction problem*, J. Assoc., Comput. Mach., 22(2), (1975), 177–183.
21. S. Y. Lu and K. S. Fu, *A sentence-to-sentence clustering procedure for pattern analysis*, IEEE Trans. Syst., Man., Cybern., Vol. SMC-8(5), (May, 1978), 381–389.
22. D. Maier, *The complexity of some problems on subsequences and supersequences*, J. Assoc. Comput. Mach. 25(2), (April, 1978), 322–336.
23. W. J. Masek and M. S. Paterson, *A faster algorithm computing string edit distances*, J. Comput. and Syst. Sci. 20 (1980), 18–31.
24. H. L. Morgan, *Spelling correction in systems programs*, Comm. ACM 13(2), (Feb. 1970), 90–94.
25. A. Mukhopadhyay, *A fast algorithm for the longest-common-subsequence problem*, Inf. Sci. 20, (1980), 69–82.
26. D. Sankoff, *Matching sequences under deletion insertion constraints*,Proc. Nat. Acad. Sci., U.S.A., 69 (1972), 4–6.
27. S. M. Selkow, *The tree-to-tree editing problem*, Inform. Processing Letters, 6, 6 (Dec., 1977), 184–186.
28. P. H. Sellers, *An algorithm for the distance between two finite sequences*, J. Combinatorial Theory Ser. A16: (1974), 253–258.
29. R. A. Wagner, *Common phrases and minimum-space text storage*, Comm. ACM, 16(3), (March, 1973), 148–152.
30. R. A. Wagner and M. J. Fischer, *The string-to-string correction problem*, J. Assoc. Comput. Mach. 21(1), (1974), 168–173.
31. P. A. Wagner, *On the complexity of the extended string-to-string correction problem*. Proc. Seventh Annual ACM Symp. on Theory of Comput., (1975), 218–223.
32. C. K. Wong and A. K. Chandra, *Bounds for the string editing problem*, J. Assoc. Comput. Mach. 28(1) (Feb. 1976), 13–18.
33. E. Horowitz and S. Sahni, *Fundamentals of Data Structures*, Computer Science Press, Potomac, Maryland, (1976).