Algorithmica (1989) 4: 503-510

Algorithmica
© 1989 Springer-Verlag New York Inc.

The Set-Set LCS Problem

D. S. Hirschberg[1] and L. L. Larmore[1]

Abstract. An efficient algorithm is presented that solves a generalization of the Longest Common Subsequence problem, in which both of the input strings consists of sets of symbols which may be permuted.

Key Words. Subsequence, Common Subsequence, Dynamic programming.

1. Introduction. The *Longest Common Subsequence* (LCS) problem can be described as follows: given two sequences $\alpha = \{a_i\}_{1 \le i \le m}$ and $\beta = \{b_j\}_{1 \le j \le n}$, find a longest sequence which is a subsequence of both α and β. The reader is referred to [M] and [S] for a thorough background of the LCS problem.

In this paper we discuss a generalization of the LCS problem which we call the *Set-Set LCS* problem. Given a sequence of sets $\mathbf{A} = \{A_i\}_{1 \le i \le p}$, we say that a sequence $\alpha = \{a_k\}_{1 \le k \le m}$ is a *flattening* of $\mathbf{A}$ if α is the concatenation of strings, the ith of which is some permutation of A_i. An instance of the Set-Set LCS problem of size $m \times n$ consists of two sequences of subsets of some alphabet Σ, $\mathbf{A} = \{A_i\}_{1 \le i \le p}$ and $\mathbf{B} = \{B_j\}_{1 \le j \le q}$, where the sum of the cardinalities of the A_i is m, and the sum of the cardinalities of the B_j is n.

We define the Set-Set LCS problem to be the problem of finding a longest common subsequence of α and β, where α ranges over all possible flattenings of $\mathbf{A}$, and β ranges over all possible flattenings of $\mathbf{B}$. As an example, if $\mathbf{A} =$ ({comp}, {uter}, {science}, {degree}) and $\mathbf{B} =$ ({greedy}, {algorithm}, {cou}, {rse}) then a Set-Set LCS of $\mathbf{A}$ and $\mathbf{B}$ is (morticser).

The case where all the sets of both $\mathbf{A}$ and $\mathbf{B}$ are singletons is that of the LCS problem. The LCS problem has been solved in time $O(mn)$ and linear space [H], and there is known a subquadratic-time algorithm [MP]. The case where all the sets of just $\mathbf{B}$ are singletons was solved in $O(mn)$ time in a previous paper [HL]. That case arose from the problem of computer-driven music accompaniment, matching polyphonic performances against a solo score [BD]. In this paper we present an $O(mn)$-time algorithm which solves the general Set-Set LCS problem. This problem corresponds to the generalization of the problem of computer-driven music accompaniment, where now the score, as well as the live performance, may have embedded chords. Note that, in accordance with the meaning of chords, the sets are simple sets. However, as detailed at the end of Section 3, the algorithm can handle multisets.

[1] Department of Information and Computer Science, University of California at Irvine, Irvine, CA 92717, USA.

Received August 1985; revised June 1987. Communicated by David Dobkin.

Notation. A capital letter denotes a set. A greek letter denotes a string.

2. Abstract Description of the Algorithm.

2. Abstract Description of the Algorithm. For convenience we define $A_0 = B_0 = \emptyset$. We define *Entries*(i, j) to be the set of triples (k, F, G) such that:

(1) k is the length of γ, a common subsequence of some flattening of $A_1 \cdots A_i$ and some flattening of $B_1 \cdots B_j$,
(2) *free set* $F \subseteq A_i$ is the set of symbols of A_i which are not used by γ, and
(3) free set $G \subseteq B_j$ is the set of symbols of B_j not used by γ.

We refer to such a triple as an *entry*. The length of the LCS of some flattening of $A_1 \cdots A_p$ and some flattening of $B_1 \cdots B_q$ is then, by definition, the largest k such that $(k, F, G) \in$ *Entries*(p, q) for some F and G. *Entries*$(0, 0)$ contains just one entry, namely $(0, \emptyset, \emptyset)$, while *Entries*$(i, j)$ can be computed dynamically from *Entries*$(i-1, j)$ and *Entries*$(i, j-1)$. The problem is that the cardinality of *Entries*(i, j) could become very large, making such an algorithm exponential in the worst case.

If $e = (k, F, G) \in$ *Entries*$(i-1, j)$ and S is any subset of $A_i \cap G$, then $e' = (k+|S|, A_i - S, G - S)$ is an element of *Entries*(i, j) and we say that *e vertically generates e'*. That $e' \in$ *Entries*(i, j) is shown by the following. If α is a length k common subsequence of some flattening of $A_1 \cdots A_{i-1}$ and some flattening of $B_1 \cdots B_j$, where $F \subseteq A_{i-1}$ and $G \subseteq B_j$ are the free sets, and β is a sequence consisting of the elements of S written in any order, then $\alpha\beta$ (having length $k+|S|$) is a common subsequence of a flattening of $A_1 \cdots A_i$ and a flattening of $B_1 \cdots B_j$, with free sets $A_i - S$ and $G - S$.

Similarly, if $(k, F, G) \in$ *Entries*$(i, j-1)$ and $S \subseteq F \cap B_j$, we say that (k, F, G) *horizontally generates* $(k+|S|, F - S, B_j - S) \in$ *Entries*(i, j).

LEMMA 1. *If $e \in$ Entries(i, j) for $i+j > 0$, then e is generated by some element of either Entries$(i-1, j)$ or Entries$(i, j-1)$.*

PROOF. Let $(k, F, G) \in$ *Entries*(i, j), and let γ be a common subsequence of some flattening of $A_1 \cdots A_i$ and some flattening of $B_1 \cdots B_j$, with free sets $F \subseteq A_i$ and $G \subseteq B_j$. Flattenings of both $(A_i - F)$ and $(B_j - G)$ appear as suffixes of γ, therefore either $(A_i - F) \subseteq (B_j - G)$ or $(B_j - G) \subseteq (A_i - F)$. The cases are symmetric, and we consider the case that $(B_j - G) \subseteq (A_i - F)$. Let β be the suffix of γ which is a flattening of $S = B_j - G$, and let α be the prefix of γ such that $\alpha\beta = \gamma$. Then α is a common subsequence of some flattening of $A_1 \cdots A_i$ and some flattening of $B_1 \cdots B_{j-1}$, with free sets $F \cup S$ and some $G' \subseteq B_{j-1}$. In this case, $(k - |S|, F \cup S, G')$, which is an element of *Entries*$(i, j-1)$, generates (k, F, G). In the symmetric case, (k, F, G) is generated by an entry of *Entries*$(i-1, j)$. $\square$

The following algorithm is a dynamic programming algorithm in which the boundary conditions are set and then the internal entries are determined:

The algorithm—initial description

```
  for all $i$, $Entries(i,0) \leftarrow \{(0, A_i, \emptyset)\}$
  for all $j$, $Entries(0,j) \leftarrow \{(0, \emptyset, B_j)\}$
  for $i = 1$ to $p$
    for $j = 1$ to $q$
      $Entries(i,j) \leftarrow$ {all entries vertically generated from $Entries(i-1,j)$}
                        $\cup$ {all entries horizontally generated from $Entries(i,j-1)$}

  $max\_k \leftarrow$ the largest $k$ such that $(k, F, G) \in Entries(p,q)$ for some $F$, $G$
```

The above algorithm may be very time-consuming because of a proliferation of triples. We will speed the algorithm by eliminating consideration of many triples.

If (k, F, G), $(k', F', G') \in Entries(i,j)$, we say that (k, F, G) *dominates* (k', F', G') if the following conditions hold:

1. $k \geq k'$.
2. $|F' - F| \leq k - k'$.
3. $|G' - G| \leq k - k'$.

LEMMA 2. *Any element of Entries(i,j) which is not maximal with respect to the relation "dominates" can be discarded during execution of the algorithm without affecting the final value of max_k.*

PROOF. By downward double induction on i and j. The value of max_k is obtained in the last step from just one maximal element of $Entries(p,q)$, and all other elements may be discarded with no effect. Suppose $i+j<p+q$, and suppose $e' \in Entries(i,j)$ is not maximal. Then e' is dominated by some maximal $e \in Entries(i,j)$. We will show that any maximal element of $Entries(i+1,j)$ or $Entries(i,j+1)$ which is generated by e' is also generated by e. By the inductive hypothesis, e' can then be discarded.

Write $e = (k, F, G)$ and $e' = (k', F', G')$. Suppose that e' horizontally generates $f' = (k' + |S'|, F' - S', B_{j+1} - S')$ for some $S' \subseteq F' \cap B_{j+1}$. Let $S = S' \cap F$, and let $f = (k + |S|, F - S, B_{j+1} - S)$, which is horizontally generated by e. Since e dominates e', we know that $\delta = k - k' \geq 0$ and that $x = |F' - F| \leq \delta$. Then $y = |S' - S| = |S' - (S' \cap F)| = |S' - F| \leq x \leq \delta$. We see that $z = (k + |S|) - (k' + |S'|) = \delta - y \geq 0$. Also, $|(F' - S') - (F - S)| = |(F' - F) - (S' - S)| \leq x - y \leq \delta - y = z$. And also $G' - G = \emptyset$, since $S \subseteq S'$. Thus f dominates f', and so either f' is not maximal or $f = f'$.

The vertical case is similar. □

If $e = (k, F, G) \in Entries(i,j)$, we define the *horizontal child* of e to be $hor(e) = (k + |F \cap B_{j+1}|, F - B_{j+1}, B_{j+1} - F)$, and define the *vertical child* of e to be $ver(e) = (k + |A_{i+1} \cap G|, A_{i+1} - G, G - A_{i+1})$. We define $MaxEntries(i,j)$ to be the set of maximal elements of $Entries(i,j)$ under the dominance relation.

LEMMA 3. *Any entry horizontally generates at most one maximal entry and vertically generates at most one maximal entry.*

PROOF. Let $e=(k, F, G) \in \mathit{Entries}(i,j)$. The only entry horizontally generated by e which could possibly be maximal is $\mathit{hor}(e)$, since it dominates any other entry horizontally generated by e. Similarly, $\mathit{ver}(e)$ dominates any other entry vertically generated by e. □

We say that (k, F, G) *strongly dominates* (k', F, G) if $k>k'$. If $S \subseteq \mathit{Entries}(i,j)$, define $\mathit{Dom}(S) \subseteq S$ to be the set obtained by deleting every element of S which is strongly dominated by another element of S.

We now inductively define sets $\mathit{Chain}(i,j) \subseteq \mathit{Entries}(i,j)$ by:

1. $\mathit{Chain}(i,0)=\{(0, A_i, \emptyset)\}$.
2. $\mathit{Chain}(0,j)=\{(0, \emptyset, B_j)\}$.
3. $\mathit{Chain}(i,j)=\mathit{Dom}(\{\mathit{ver}(e) \mid e \in \mathit{Chain}(i-1,j)\} \cup \{\mathit{hor}(e) \mid e \in \mathit{Chain}(i,j-1)\})$.

We refer to entries in $\mathit{Chain}(i,j)$ as *weakly maximal.* We observe the following lemma.

LEMMA 4. $\mathit{MaxEntries}(i,j) \subseteq \mathit{Chain}(i,j)$.

PROOF. By double induction. For $i=0$ or $j=0$, the two sets are identical. For $i, j>0$, and $e \in \mathit{MaxEntries}(i,j)$ must be a horizontal or vertical child of some maximal entry, which is weakly maximal by induction. It follows that e must be weakly maximal, since it is maximal and thus cannot be deleted by the operator *Dom*. □

Making use of Lemmas 2-4, we modify the algorithm as follows:

```
The algorithm—using weakly maximal entries
  for all $i$, $Chain(i,0) \leftarrow \{(0, A_i, \emptyset)\}$
  for all $j$, $Chain(0,j) \leftarrow \{(0, \emptyset, B_j)\}$
  for $i=1$ to $p$
    for $j=1$ to $q$
      begin
        $Chain(i,j) \leftarrow \emptyset$
        for all $(k, F, G) \in Chain(i,j-1)$
          insert $(k+|F \cap B_j|, F-B_j, B_j-F)$ into $Chain(i,j)$
        for all $(k, F, G) \in Chain(i-1,j)$
          insert $(k+|A_i \cap G|, A_i-G, G-A_i)$ into $Chain(i,j)$
        delete all nonweakly maximal elements from $Chain(i,j)$
      end
  $max\_k \leftarrow$ the maximum value of $k$ such that $(k, F, G) \in Chain(p,q)$ for
                                                        some $F$ and $G$
```

3. Efficient Implementation of the Algorithm. We now develop a logical structure on $Chain(i,j)$ that will help obtain an efficient implementation of the algorithm. We begin by defining a transitive reflexive relation $\lhd$ on $Entries(i,j)$. We say that $(k, F, G) \lhd (k', F', G')$ if $F \subseteq F'$ and $G \supseteq G'$.

LEMMA 5.

(a) *If e, $e' \in Entries(i, j-1)$, and if $e \lhd e'$, then $hor(e) \lhd hor(e')$.*
(b) *If e, $e' \in Entries(i-1, j)$, and if $e \lhd e'$, then $ver(e) \lhd ver(e')$.*
(c) *If $e \in Entries(i, j-1)$ and $e' \in Entries(i-1, j)$, then $hor(e) \lhd ver(e')$.*

PROOF. (a) $hor(e) = (k, F - B_j, B_j - F)$ and $hor(e') = (k', F' - B_j, B_j - F')$. It follows from $F \subseteq F'$ that $F - B_j \subseteq F' - B_j$ and $B_j - F \supseteq B_j - F'$, i.e., $hor(e) \lhd hor(e')$.

(b) Similar to the proof of (a).

(c) Write $e = (k, F, G)$ and $e' = (k', F', G')$. Then $hor(e) = (k, F - B_j, B_j - F)$ and $ver(e') = (k', A_i - G', G' - A_i)$. We see that $F \subseteq A_i$ since $e \in Entries(i, j-1)$, and that $G' \subseteq B_j$ since $e' \in Entries(i-1, j)$. As a result, $F - B_j \subseteq A_i - G'$ and $B_j - F \supseteq G' - A_i$, i.e., $hor(e) \lhd ver(e')$. □

LEMMA 6. *The relation $\lhd$ imposes a total order on $Chain(i,j)$.*

PROOF. It suffices to show that for any distinct f, $f' \in Chain(i,j)$, either $f \lhd f'$ or $f' \lhd f$, but not both. If $f \lhd f'$ and $f' \lhd f$, then f and f' would have the same free sets, which implies they must be identical, else the one with the smaller value of k would not be weakly maximal. Thus, we need only show that f and f' are comparable. We do this by induction on i and j.

$Chain(0,j)$ contains just one entry, namely $(0, \emptyset, B_j)$, and hence is ordered. Similarly, $Chain(i, 0)$ contains only the entry $(0, A_i, \emptyset)$.

Suppose $i, j > 0$, and f, $f' \in Chain(i,j)$. Both f and f' must be generated by maximal entries e and e', respectively. We consider three cases. If f and f' are the horizontal children of e and e', respectively, then by induction, e and e' are comparable, hence f and f' are comparable by Lemma 5(a). If f and f' are the vertical children of e and e', the proof is similar, using Lemma 5(b). If f is the horizontal child of e and f' is the vertical child of e', then f and f' are comparable by Lemma 5(c). □

COROLLARY. *$Chain(i,j)$ has cardinality at most $1 + |A_i| + |B_j|$.*

PROOF. If $e = (k, F, G) \in Entries(i,j)$, define the *signature* of e to be $|F| - |G|$, which must lie in the range $[-|B_j|, |A_i|]$. Since $Chain(i,j)$ is ordered under the relation $\lhd$, each entry must have a different signature. □

The following small example illustrates the generation of $Chain(i,j)$. Sets of characters are indicated without the usual braces and commas. The empty set is indicated by "-":

	red	algorithm	course
			(2,mp,urse)
comp	(0,comp,red)	(2,cp,algrith)	(3,p,ourse)
			(5,t,s)
			(6,t,os)
		(4,ue,algih)	(6,-,cors)
uter	(2,ut,d)	(3,u,algorihm)	(4,-,corse)

The horizontally generated entries are aligned horizontally. In this example, no entries were eliminated as a result of strong dominance. The length of the Set-Set LCS is 6. Note that, in accord with Lemma 5(c), the horizontally generated entries $\lhd$ the vertically generated entries.

We now develop an efficient data structure to represent the lists $\mathit{Chain}(i,j)$.

Respecting Orderings. If A is a finite set, and if C is a set of subsets of A, we say that an ordering $a_1, \ldots, a_{|A|}$ of A *respects* C if, for each $S \in C$, $S = \{a_1, \ldots, a_{|S|}\}$. We remark without proof:

LEMMA 7. *There exists an ordering of A which respects C if and only if C is strictly ordered by proper inclusion.*

LEMMA 8. *If A is ordered in a manner which respects C, then each element of C may be represented by an integer in the range* $[0, |A|]$.

Notation. If $\alpha = (a_1, \ldots, a_n)$ is a list of distinct items, and if S is a set, then we let $\alpha \uparrow S$ denote the list consisting of all a_i which are members of S, in the ordering inherited from α. We let $\alpha \downarrow S$ denote the list consisting of the remaining items of α, also in the inherited order. For example, if $\alpha = (s, h, a, d, e)$ and $S = \{t, h, e\}$, then $\alpha \uparrow S = (h, e)$ and $\alpha \downarrow S = (s, a, d)$.

Implementation of $\mathit{Chain}(i,j)$. Let $\mathit{Chain}(i,j) = (k_1, F_1, G_1) \lhd \cdots \lhd (k_r, F_r, G_r)$. Then $\mathit{Chain}(i,j)$ will be represented by the following three lists:

(1) α_{ij}, an ordering of A_i which respects $\{F_1, \ldots, F_r\}$.
(2) β_{ij}, an ordering of B_j which respects $\{G_1, \ldots, G_r\}$.
(3) γ_{ij}, the list of ordered triples of integers $(k_1, |F_1|, |G_1|), \ldots, (k_r, |F_r|, |G_r|)$.

The values of all free sets for all weakly maximal entries in $\mathit{Chain}(i,j)$ are determined by the above information, which takes $O(|A_i| + |B_j|)$ space.

Initialization. $\mathit{Chain}(i, 0)$ is initialized by choosing α_{i0} to be an arbitrary ordering for A_i. The single element of $\mathit{Chain}(i, 0)$ is represented by the triple $(0, |A_i|, 0)$. $\mathit{Chain}(0, j)$ is initialized similarly.

Construction of Chain(i,j). For $i, j > 0$, we can define

$$\alpha_{ij} = (\alpha_{i,j-1} \downarrow B_j) \bullet (\beta_{i-1,j} \uparrow A_i)^{\mathrm{R}},$$
$$\beta_{ij} = (\beta_{i-1,j} \downarrow A_i) \bullet (\alpha_{i,j-1} \uparrow B_j)^{\mathrm{R}},$$

where $\bullet$ denotes concatenation and α^{R} denotes the reverse of list α.

We expend $O(|A_i|+|B_j|)$ time to compute and store an index function *index*, which has the property that

$$\alpha_{i,j-1}[1\,..\,l] \downarrow B_j \text{ has length index}(l) \qquad \text{for all} \quad 1 \leq l \leq |A_i|.$$

Suppose $e=(k, F, G) \in Chain(i, j-1)$ is represented by the triple $(k, |F|, |G|)$ in the list $\gamma_{i,j-1}$. Then $hor(e) = (k+|F \cap B_j|,\ F - B_j,\ B_j - F)$ is represented by the triple

$$(k+|F| - index(|F|),\ index(|F|),\ |B_j| - |F| + index(|F|))$$

which can be computed in constant time. Similarly, the triple representing the vertical child of $e \in Chain(i-1, j)$ can be computed in constant time. The list γ_{ij} is obtained by considering all such horizontal and vertical children, but keeping only those triples (k, f, g) where k is maximum for given f and g. The culling process can be done in $O(1)$ time for each triple culled, since the triples are generated in monotone order with respect to the relation $\lhd$. Thus, $Chain(i,j)$ can be generated in $O(|A_i|+|B_j|)$ time altogether.

The entire algorithm thus takes $O(mn)$ time, and the longest common subsequence can be recovered by maintaining a parent pointer from each weakly maximal entry to the entry that generated it.

Reduced Space. If $|A_i| \leq a$ and $|B_j| \leq b$ for all i and j, the longest common subsequence can be found in $O(bn)$ time or $O(am)$ time, whichever is smaller, by using a variation of Hirschberg's technique, from [H]. Without loss of generality, $am \leq bn$. The weakly maximal entries, together with their parent pointers, form a tree T rooted at $(0, \emptyset, \emptyset) \in Chain(0, 0)$. For $e = (k, F, G) \in Chain(i, j)$, we define

$$depth(e) = |A_1| + \cdots + |A_i| - |F|.$$

Note that the depth of an entry must be at least as great as the depth of its parent. We define an entry e to have *middling* depth if $depth(parent(e)) \leq n/2 < depth(e)$. Note that each e whose depth exceeds $n/2$ has a unique ancestor in T of middling depth, which we call $Mid(e)$.

The Space Saving Algorithm. We run the algorithm, discarding the data structure for $Chain(i,j)$ as soon as all children of its elements have been computed, saving only $Chain(p, q)$ and also saving all weakly maximal entries of middling depth and their parents. The space needed is $O(am)$. In addition, for entries whose depth exceeds $n/2$, we compute the middling ancestor. Finally, let $best \in Chain(p, q)$ be the entry of maximum k, let $e = Mid(best)$, $e' = parent(e)$, and *root* $= (0, \emptyset, \emptyset)$. The path from *best* to *root* can be recovered by running the algorithm recursively twice, once to recover the path from e' to *root*, and once from *best* to e. These subproblems have size, together, at most half the size of the original problem. Thus the total time complexity is still $O(mn)$.

Multisets. All the methods of this paper can be applied to the more general problem where the A_i and B_j are multisets. If L is a list (of not necessarily distinct items) and S is a multiset, the list $L \uparrow S$ is defined as the list whose items are those elements of S that appear in L, in the order that they appear in L. If $x \in S$ of multiplicity s and x appears l times in L, then $L \uparrow S$ contains the first $\min\{s, l\}$ appearances of x. We then define $L \downarrow S$ to be the list consisting of the items of L after the items of $L \downarrow S$ have been removed.

For example, if $L = (a, b, c, b, a, d)$ and $S\{a, b, d, d\}$ then $L \uparrow S = (a, b, d)$ and $L \downarrow S = (c, b, a)$.

References

[BD] J. J. Bloch and R. B. Dannenberg, Real-time computer accompaniment of keyboard performances, *Proc. 1985 Int. Computer Music Conf.* (Aug. 1985).

[H] D. S. Hirschberg, A linear space algorithm for computing maximal common subsequences, *Comm. ACM* **18**, 6 (1975), 341-343.

[HL] D. S. Hirschberg and L. L. Larmore, The Set LCS problem, *Algorithmica* **2** (1987), 91-95.

[MP] W. J. Masek and M. S. Paterson, A faster algorithm for computing string-edit distances, *J. Comput. System Sci.* **20**, 1 (1980), 18-31.

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

[S] D. Sankoff and J. B. Kruskal (eds.), *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison*, Addison-Wesley, Reading, MA, 1983.