The Set LCS Probelm

D.S. Hirschberg and L.L. Larmore
University of California, Irvine

Technical Report No. 85-23

August, 1985

The Set LCS Problem

D.S. Hirschberg and *L.L. Larmore*
University of California, Irvine

Abstract

An efficient algorithm is presented that solves a generalization of the Longest Common Subsequence problem, in which one of the two input strings contains sets of symbols which may be permuted. This problem arises from a music application.

1. Introduction

The *Longest Common Subsequence* (LCS) problem can be described as follows: Given two sequences $A = \{a_i\}_{1 \leq i \leq m}$ and $B = \{b_j\}_{1 \leq j \leq n}$, find a longest sequence which is a subsequence of both A and B. The LCS problem has been solved in quadratic time and linear space [H], and there is known a subquadratic time algorithm [MP]. Some special cases of the LCS problem can be solved much faster [H2,M].

In this paper, we discuss a generalization of the LCS problem (suggested by Roger Dannenberg [D]), which we call the *Set LCS* (SLCS) problem. One sequence (in some alphabet Σ) $B = \{b_j\}_{1 \leq j \leq n}$ is given, as before. Instead of a second sequence in Σ, we are given a sequence of subsets of Σ, namely $\alpha = \{\Sigma_k\}_{1 \leq k \leq r}$, where the sum of the cardinalities of the Σ_k is m. We say that a sequence $A = \{a_i\}_{1 \leq i \leq m}$ is a *flattening* of α if A is the concatenation of strings, the k^{th} of which is some permutation of Σ_k.

We define the SLCS problem to be the problem of finding a longest common subsequence of A and B, where B is fixed and A ranges over all possible flattenings of a given string of subsets α.

Authors' address: Department of Information and Computer Science, University of California, Irvine, CA 92717.

The SLCS problem has application to a problem in music [BD]. Computer-driven music accompaniment has been based on matching polyphonic performances (scores) against a solo score. Polyphonic music is a performance in which multiple notes can occur simultaneously, such as in a chord. A polyphonic score can be described as a sequence of sets of notes. The problem is to decide when notes of the solo score are to be played so as to accompany the performance in progress. The simultaneous notes may be matched in any order. In [BD], a heuristic is proposed for solving the SLCS problem. This heuristic obtains reasonable but not always optimal length common subsequences. It might not be possible to guaranteee an optimal solution for the real-time application. We consider the case of an off-line application.

In the next section, we present an $O(mn)$ algorithm which solves the SLCS problem. The algorithm is reminiscent of the classic dynamic programming algorithm for the LCS problem. We refer the reader to [H] for a discussion of that algorithm.

Throughout this paper, we use the following substring notation: if $X = x_1 x_2 \ldots x_N$ is a string (of elements or sets), $X\langle s{:}t\rangle$ denotes the substring $x_s \ldots x_t$, and X_t denotes the (prefix) substring $x_1 \ldots x_t$. Given an instance (α, B) of the SLCS problem, we say that a sequence γ is a *candidate*(i,j) if γ is a common subsequence of both B_j and some flattening of α_i. γ is a *solution*(i,j) if γ is a candidate(i,j) having maximal length.

2. The Algorithm

Define $\mathcal{L}(i,j)$ as the length of the longest sequence which is a candidate(i,j). Thus, $\mathcal{L}(i,j) = 0$ if either $i = 0$ or $j = 0$, and $\mathcal{L}(r,n)$ is the length of the desired final solution.

The algorithm presented in this paper computes the values for a matrix *best*[*,*], which agree with the theoretical values of $\mathcal{L}(*,*)$. We also indicate how to define a system of pointers which will allow recovery of a solution sequence, without increasing the asymptotic time complexity.

The zeroth row $best[0,*]$ is identically zero. For each $i > 0$, the algorithm computes the values in row i of $best$ from the already computed values in row $i-1$. The output of the main algorithm is the array $best[*,*]$.

Main Algorithm

```
best[0,j] ← 0 for all 0 ≤ j ≤ n
for i ← 1 to r do
    begin
        Findpeaks(i)
        Shadow(i)
    end
```

The main loop of the algorithm contains two steps: Findpeaks and Shadow. The input for Findpeaks(i) is the matrix row $best[i-1,*]$, and its output is the array $peak[*]$. The input for Shadow(i) is the array $peak[*]$, and its output is the matrix row $best[i,*]$.

We give an intuitive explanation of Findpeaks as follows. Suppose that δ is a subsequence of $B\langle j+1:k\rangle$ consisting of distinct elements, each of which is a member of Σ_i. Appending δ to any candidate$(i-1,j)$ produces a candidate(i,k). It follows that $\mathcal{L}(i,k) \geq \mathcal{L}(i-1,j) + |\delta|$. The procedure Findpeaks(i) searches for such subsequences and, if one is found, sets the value of $peak[k]$ to be the new candidate length for $best[i,k]$, provided it is larger than the largest previously found candidate length.

Findpeaks makes use of an array *first* and a data structure U, which we call a unique stack. A ***unique stack*** is a stack with the condition that no member can occur twice in the stack. When $Push(x, U)$ is executed for some item x, x is first deleted from U if it is already a member. In Findpeaks(i), as j varies, U is a list of all members of Σ_i which are found in the substring $B\langle j+1:n\rangle$ in the order in which they first occur. For any $x \in U$, $first[x]$ is the index of that first occurrence.

Findpeaks(i)

```
peak[j] ← 0,  for all 0 ≤ j ≤ n
U ← empty stack
```

```
for j ← n downto 0 do
    begin
        length ← best[i−1,j]
        peak[j] ← max{ length, peak[j] }
        for x ← elements of U, from Top(U) to Bottom(U), do
            begin
                length ← length+1
                peak[ first[x] ] ← max{ length, peak[ first[x] ] }
            end
        x ← b_j
        if x ∈ Σ_i then
            begin
                Push(x, U)
                first[x] ← j
            end
    end
```

The procedure Shadow(i) computes $best[i,j]$ for all j, using the rule that, as a function of its second parameter alone, $\mathcal{L}(i,\bullet)$ is the minimum monotone increasing function such that $\mathcal{L}(i,j) \geq peak[j]$ for all j.

Shadow(i)

```
best[i,0] ← 0
for j ← 1 to n do
        best[i,j] ← max{ peak[j], best[i,j−1] }
```

Recovery of a solution sequence. A solution(i,j), for any i and j, can be recovered after the algorithm is finished if an array of backpointers is maintained. Each backpointer is an (i,j) pair, and a new value of a backpointer is needed whenever a new (i.e., higher) value of *peak* is assigned, and also whenever $best[i,j]$ is assigned the value of $best[i,j-1]$ during Shadow(i). Inclusion of these backpointers does not increase the time complexity of the algorithm, and recovery of a solution(i,j) takes time $O(\mathcal{L}(i,j))$. The details, which we leave as an exercise to the reader, are straightforward.

Time Complexity. There are a number of ways to implement the unique stack. We suggest representing U as a doubly linked list, simultaneously maintaining a doubly linked list of all elements of the alphabet Σ which are currently not in U. Thus, it will take only $O(1)$ time to push an element onto U (we can find where an element x is located in the linked lists by an array lookup, and delete x from its list and then insert x at the top of U) or to set U to an empty list.

Each traversal of U requires $O(|\Sigma_i|)$ time, and there are $O(n)$ such traversals during the i^{th} iteration of the main loop of the algorithm. Since the sum of the cardinalities of the Σ_i is m, the total time spent on traversing the unique stack is $O(mn)$. All other parts of the algorithm combined require only $O(rn)$ time.

Space complexity. The algorithm, as presented, requires $O(rn + m)$ space, most of which is for array *best*. The space complexity for the algorithm that recovers a solution sequence can be reduced to $O(m+n)$ in a straightforward manner, by using a recursion technique developed for the LCS problem [H].

3. Proof of Correctness

Correctness of the algorithm follows immediately from the loop invariant below, which consists of two parts, F and S.

Loop invariant.

F(i): After Findpeaks(i) has executed during the i^{th} iteration of the main loop of the algorithm, the following two conditions hold for all $0 \leq j \leq n$.

F1(i): $peak[j] \leq \mathcal{L}(i,j)$

F2(i): There exists some $j_0 \leq j$ such that $peak[j_0] \geq \mathcal{L}(i,j)$

S(i): After i iterations of the main loop of the algorithm, $best[i,j] = \mathcal{L}(i,j)$, for all $0 \leq j \leq n$

We first note that correctness of the algorithm follows immediately from S, since the values of row i of *best* are never reassigned after Shadow(i) has executed.

We prove the loop invariant inductively. S(0) obviously holds. We show that F(i) implies S(i) for all $1 \le i \le r$, and that S($i-1$) implies F(i) for for all $1 \le i \le r$.

Proof that F(i) $\Rightarrow$ S(i). Execution of Shadow(i) causes

$$best[i,j] \ge peak[j], \text{ for all } j \qquad (1)$$

and $$best[i,j] \ge best[i,j-1], \text{ for all } j > 0 \qquad (2)$$

Suppose

$$best[i,j] < \mathcal{L}(i,j), \text{ for some } j \qquad (3)$$

By F2(i), there exists some $j_0 \le j$ such that

$$peak[j_0] \ge \mathcal{L}(i,j) \qquad (4)$$

Then,

$$\begin{aligned} \mathcal{L}(i,j) &\le peak[j_0], && \text{by (4)} \\ &\le best[i,j_0], && \text{by (1)} \\ &\le best[i,j], && \text{by (2)} \\ &< \mathcal{L}(i,j), && \text{by (3)} \end{aligned}$$

which is a contradiction. On the other hand, let j be the minimum index such that $best[i,j] > \mathcal{L}(i,j)$. Since $best[i,j-1] = \mathcal{L}(i,j-1)$ and $\mathcal{L}(i,\bullet)$ is monotone increasing (and thus $\mathcal{L}(i,j) \ge \mathcal{L}(i,j-1)$), $best[i,j] > best[i,j-1]$. Shadow(i) thus assigns the value of $peak[j]$ to $best[i,j]$, which contradicts F1(i).

Proof that S(i−1) $\Rightarrow$ F1(i). We show that F1(i) is a loop invariant of the loop (indexed by j) of Findpeaks(i). F1(i) holds before the first iteration, since *peak* is initialized to zero. Suppose that during the iteration indexed by j, $peak[j_0]$ is increased. The new, higher, value must be *length.* Let γ be a longest candidate($i-1,j$), and let δ be the string of all symbols in U from the top down to $x = B[j_0]$. (Note that $first[x] = j_0$.) The concatenation $\gamma\delta$ is a candidate(i,j_0). The value of *length*, when $peak[j]$ is reassigned, is the length of $\gamma\delta$, which cannot exceed $\mathcal{L}(i,j_0)$.

Proof that S(i−1) $\Rightarrow$ F2(i). We will show that, for arbitrary j, there exists some $j_0 \le j$ for which $peak[j_0] \ge \mathcal{L}(i,j)$, assuming S($i-1$). Let ς be the longest candidate(i,j), and thus $|\varsigma| = \mathcal{L}(i,j)$. We can write $\varsigma = \gamma\delta$ such that γ is a candidate($i-1,j$) and δ consists of distinct members of Σ_i. Let j_1 ($\le j$) be the index in B of the last item of γ.

(If γ is empty, we default j_1 to 0.) Note that δ is a subsequence of $B\langle j_1+1:j\rangle$. The length of γ must equal $\mathcal{L}(i-1,j_1)$, for if γ' were a candidate$(i-1,j_1)$ longer than γ, $\gamma'\delta$ would be a candidate(i,j) longer than ς. If δ is empty, we are done, since then $peak[j]$ will be assigned a value such that

$$\begin{aligned} peak[j] &\geq best[i-1,j], \text{ since } \textit{length} \text{ is initialized to this} \\ &\geq best[i-1,j_1], \text{ since } \textit{best} \text{ is non-decreasing} \\ &= \mathcal{L}(i-1,j_1), \text{ by } S(i-1) \\ &= |\gamma| = |\varsigma|, \quad \text{since } \delta \text{ is empty} \\ &= \mathcal{L}(i,j). \end{aligned}$$

Suppose, on the other hand, that δ is non-empty. Write $d = |\delta|$. Let θ be the subsequence of $B\langle j_1+1:n\rangle$ consisting of the first instance of every item of $B\langle j_1+1:n\rangle$ which is also a member of Σ_i. Let j_0 be the position of the d^{th} item of θ. By definition of θ, $B\langle j_1+1:j_0-1\rangle$ contains only $d-1$ distinct members of Σ_i. It follows that $j_0 \leq j$, since the items in the string δ are d distinct items of $B\langle j+1:j\rangle$ which are members of Σ_i. During the iteration of the main loop of Findpeaks(i) indexed by j_1, $U = \theta$. During the d^{th} iteration of the inner (descent) loop, x will be the d^{th} item of θ, $first[x]$ will be j_0, and the value of *length* will be $|\gamma| + d = |\gamma| + |\delta| = |\varsigma| = \mathcal{L}(i,j)$. It follows that the value of $peak[j_0]$ will be at least $\mathcal{L}(i,j)$ after that iteration.

4. Some Open Questions

1. Since there is an algorithm [MP] to solve the LCS problem in time $O(n^2/\log n)$, where n is the length of each string, we suggest that the time for the SLCS problem could also be reduced by a logarithmic factor.

2. Consider a generalization of the SLCS problem, the Set-Set LCS problem, in which two sequences of sets are given and the problem is to find the longest sequence which is a common subsequence of flattenings of the two sequences of sets. Is this problem even in the class $\mathcal{P}$?

3. Can any bounds be placed on the performance of real-time algorithms for the

SLCS and Set-Set LCS problems?

References

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

[D] R.B. Dannenberg, personal communication to D.S. Hirschberg, June 1985.

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

[H2] D.S. Hirschberg, "Algorithms for the longest common subsequence problem," *Journal ACM 24*,4 (Oct. 1977), 664-675.

[M] E.W. Myers, "An O(ND) difference algorithm and its variations," unpublished manuscript, Dept. of Computer Science, U. of Arizona, 1985.

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