Algorithmica (1987) 2: 91-95

Algorithmica

# The 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 one of the two input strings contains sets of symbols which may be permuted. This problem arises from a music application.

**Key Words.** Subsequence, Common subsequence, Dynamic programming.

**1. Introduction.** The *Longest Common Subsequence* (LCS) problem can be described as follows: Given two sequences $A=\{a_i\}_{1\le i\le m}$ and $B=\{b_j\}_{1\le j\le n}$, 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].

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

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 $\alpha$.

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. Polphonic 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 guarantee 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

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

Received July 15, 1985; revised March 15, 1986. Communicated by David P. Dobkin.

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 $X = x_1x_2 \dots x_N$ is a string (of elements or sets), $X\langle s:t\rangle$ denotes the substring $x_s \dots x_t$, and $X_t$ denotes the (prefix) substring $x_1 \dots x_t$. Given an instance $(\alpha, B)$ of the SLCS problem, we say that a sequence $\gamma$ is a *candidate*$(i, j)$ if $\gamma$ is a common subsequence of both $B_j$ and some flattening of $\alpha_i$. $\gamma$ is a *solution*$(i, j)$ if $\gamma$ is a candidate$(i, j)$ having maximal length.

## 2. The Algorithm.

Define $\mathscr{L}(i, j)$ as the length of a solution$(i, j)$. Thus, $\mathscr{L}(i, j) = 0$ if either $i = 0$ or $j = 0$, and $\mathscr{L}(r, n)$ is the length of the desired final solution. The following recurrence is crucial to our algorithm.

$$\mathscr{L}(i, j) = \max\{\mathscr{L}(i-1, k) + |\Sigma_i \cap \{b_{k+1}, \dots, b_j\}| \text{ for } 0 \le k \le j\}.$$

Our algorithm computes the values for a matrix $\mathscr{L}[*, *]$, which agree with the theoretical values of $\mathscr{L}(*, *)$. 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 $\mathscr{L}[0, *]$ is identically zero. For each $i > 0$, the algorithm computes the values in row $i$ of $\mathscr{L}$ from the already computed values in row $i-1$, using the rule that, as a function of its second parameter alone, $\mathscr{L}(i, \bullet)$ is the minimum monotone increasing function such that $\mathscr{L}(i, j) \ge peak[j]$ for all $j$. The output of the main algorithm is the array $\mathscr{L}[*, *]$.

**Main Algorithm**

```
$\mathscr{L}[0, j] \leftarrow 0$ for all $0 \le j \le n$
for $i \leftarrow 1$ to $r$ do
    begin
        Findpeaks($i$)
        $\mathscr{L}[i, 0] \leftarrow 0$
        for $j \leftarrow 1$ to $n$ do
            $\mathscr{L}[i, j] \leftarrow \max\{peak[j], \mathscr{L}[i, j-1]\}$
    end
```

The main loop of the algorithm contains an invocation of Findpeaks. The input for Findpeaks$(i)$ is the matrix row $\mathscr{L}[i-1, *]$, and its output is the array *peak*$[*]$.

We give an intuitive explanation of Findpeaks as follows. A solution$(i, j)$ consists of the concatenation of a solution$(i-1, k)$ with a subsequence of $B\langle k+1:j\rangle$, for an appropriate $k$, consisting of elements of $\Sigma_i$. Suppose that $\delta$ is a subsequence of $B\langle k+1:j\rangle$ consisting of distinct elements, each of which is a member of $\Sigma_i$. Appending $\delta$ to any candidate $(i-1, k)$ produces a candidate $(i, j)$. It follows that $\mathscr{L}(i, j) \ge \mathscr{L}(i-1, k) + |\delta|$. The procedure Findpeaks$(i)$ searches for such subsequences and, if one is found, sets the value of *peak*$[j]$ to be the new candidate length for $\mathscr{L}[i,j]$, 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 $k$ varies, $U$ is a list of all members of $\Sigma_i$ which are found in the substring $B\langle k+1\colon n\rangle$ in the order in which they first occur. For any $x \in U$, $first[x]$ is the index of that first occurrence. Findpeaks($i$) finds peaks of candidate lengths for $\mathscr{L}[i,j]$ constructed from $\mathscr{L}(i-1,k)$ plus the number of elements in $U$ (elements of $\Sigma_i$) in $B\langle k+1\colon j\rangle$. However, these peaks are determined only for positions $j$ which contain an element of $U$. Those positions "in between" would have the same value of peak, and these values are filled in by the inner loop of the main program.

```
Findpeaks($i$)
$U \leftarrow$ empty stack
for $k \leftarrow n$ down to 0 do
    begin
        $peak[k] \leftarrow length \leftarrow \mathscr{L}[i-1, k]$
        for $x \leftarrow$ elements of $U$, from Top($U$) to Bottom($U$), do
            begin
                $length \leftarrow length + 1$
                $peak[first[x]] \leftarrow \max\{length, peak[first[x]]\}$
            end
        $x \leftarrow b_k$
        if $x \in \Sigma_i$ then
            begin
                $Push(x, U)$
                $first[x] \leftarrow k$
            end
    end
```

*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 $\mathscr{L}[i,j]$ is assigned the value of $\mathscr{L}[i,j-1]$ within the inner loop of the main program. Inclusion of these backpointers does not increase the time complexity of the algorithm, and recovery of a solution($i,j$) takes time $O(\mathscr{L}(i,j))$. 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. One method represents $U$ as a singly linked list, while maintaining a vector of pointers, one per symbol in alphabet $\Sigma$. The pointer or symbol $\sigma$ is $\lambda$(NIL) if $\sigma$

is not on the stack, otherwise it points at the element above $\sigma$'s occurrence in $U$. It will take only $O(1)$ time to push an element $x$ onto $U$ since we can find where $x$ is located in the linked list by an array lookup, delete $x$ from the list and then insert $x$ at the top end of $U$.

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 $\Sigma_i$ 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.

The foregoing assumes a finite alphabet, since there must be an array indexed by elements of $\Sigma$. The algorithm does extend to the case of arbitrary size alphabet but requires considerably more care and an $O(m \log m)$ preprocessing sort in order to translate symbols in $\Sigma$ to indices in the range $[1, m]$.

*Space Complexity.* The algorithm, as presented, requires $O(rn)$ space for the array $\mathscr{L}$, and $O(m)$ space for the unique stack. The space complexity for the algorithm that recovers a solution sequence can be reduced to $O(m+n)$ by using a straightforward extension of the divide-and-conquer technique developed for the LCS problem [H].

*Correctness.* We will omit the bulk of the details of the proof of correctness, since they are fairly straightforward. Essentially, that proof consists of verification of the following two-part loop invariant, which we leave as an exercise to the reader.

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($i$): $peak[j] \leq \mathscr{L}(i, j)$.

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

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

Correctness of the algorithm follows immediately from S, since the values of row $i$ of $\mathscr{L}$ are assigned once and are never reassigned.

## 3. 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, the time for the SLCS problem might also be reduced by a logarithmic factor. However, it may seem unlikely that the approach in [MP] will extend to the SLCS problem since the values of a submatrix of $\mathscr{L}$ are not dependent on just the values at two of its borders.

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 $P$?

3. Can any lower bounds be placed on the performance of algorithms (real-time, on-line, or off-line) for the SLCS and Set-Set LCS problems?
4. Is it possible to solve the SLCS problem in time $O(rn+m\log m)$?

**Acknowledgment.** The authors wish to thank an anonymous referee for suggesting the elegant method of maintaining the unique stack.

## References

[BD] J. J. Block and R. B. Dannenberg, Real-time computer accompaniment of keyboard performances, *Proceedings of the 1985 International Computer Music Conference*, August 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.

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