Computing the Number of Longest Common Subsequences

Ronald I. Greenberg
Dept. of Mathematical and Computer Sciences
Loyola University
6525 N. Sheridan Rd.
Chicago, IL 60626-5385
http://www.cs.luc.edu/~rig

Abstract

This note provides very simple, efficient algorithms for computing the number of distinct longest common subsequences of two input strings and for computing the number of LCS embeddings.

Keywords: longest common subsequences, edit distance, shortest common supersequences

1 Background and Terminologies

Let $A = a_1a_2 \ldots a_m$ and $B = b_1b_2 \ldots b_n$ ($m \leq n$) be two sequences over an alphabet Σ. A sequence that can be obtained by deleting some symbols of another sequence is referred to as a *subsequence* of the original sequence. A *common subsequence* of A and B is a subsequence of both A and B. A longest common subsequence (LCS) is a common subsequence of greatest possible length. A pair of sequences may have many different LCSs. In addition, a single LCS may have many different *embeddings*, i.e., positions in the two strings to which the characters of the LCS correspond.

Most investigations of the LCS problem have focused on efficiently finding one LCS. A widely familiar $O(mn)$ dynamic programming approach goes back at least as far as the early 1970s [5, 7, 8], and many later studies have focused on improving the time and/or space required for the computation. Methods have also been developed to efficiently generate a listing of all distinct LCSs or all LCS embeddings in time proportional to the output size (plus a preprocessing time of $O(mn)$ or less) [1, 2, 6, 3]. Here we show that the simplest scheme [3] can be simplified even further if we seek only a count of the number of distinct LCSs (or of the number of LCS embeddings). We obtain a running time of $O(mn)$ and a space bound of $O(m)$. (While the number of LCSs (or LCS embeddings) can grow very large as input size increases [4], the results here are based on the standard assumption of unit time for any arithmetic operation without worrying about the possible magnitude of the operands.)

2 Computing the Number of LCSs or LCS embeddings

The familiar $O(mn)$ method for computing the length of an LCS is a "bottom-up" dynamic programming approach based on the following recurrence for the length $L[i,j]$ of an LCS of $a_1a_2 \ldots a_i$ and $b_1b_2 \ldots b_j$:

$$L[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ L[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } a_i = b_j \\ \max\{L[i-1, j], L[i, j-1]\} & \text{otherwise} \end{cases} \tag{1}$$

We can use a similar approach to devise an $O(mn)$ algorithm to compute the number of distinct LCSs $D[m,n]$ of $a_1a_2\ldots a_m$ and $b_1b_2\ldots b_n$:

```
1   for j ← 0 to n do
2       for i ← 0 to m do
3           if i = 0 or j = 0 then D[i, j] ← 1
4           else
5               D[i, j] ← 0
6               if a_i = b_j then D[i, j] ← D[i − 1, j − 1]
7               else
8                   if L[i − 1, j] = L[i, j] then D[i, j] ← D[i, j] + D[i − 1, j] endif
9                   if L[i, j − 1] = L[i, j] then D[i, j] ← D[i, j] + D[i, j − 1] endif
10                  if L[i − 1, j − 1] = L[i, j] then D[i, j] ← D[i, j] − D[i − 1, j − 1] endif
11              endif
12          endif
13      endfor
14  endfor
```

(Note that there is always at least one LCS, since the empty string ϵ is always considered to be a common subsequence of the input sequences.)

In the pseudocode above, line 5 could have been moved inside the **else** clause beginning at line 7, but the pseudocode as written is particularly easy to modify for computation of the number of LCS embeddings rather than the number of distinct LCSs; just replace that **else** with the **endif** from line 11. (The test in line 10 is never satisfied with $a_i = b_j$, but it is harmless and concise to write the code this way.)

We may also note that $O(m)$ space suffices for the computation, since we really only need a portion of two columns of the L and D arrays at any time. Here is a rewrite of the code to achieve $O(m)$ space that also introduces the necessary change to switch from computing the number of distinct LCSs to computing the number of LCS embeddings $E[m]$; we also include the efficient computation of the L values:

```
for j ← 0 to n do
    for i ← 0 to m do
        if i = 0 or j = 0 then L[i] ← 0, oldL ← 0, E[i] ← 1, and oldE ← 1
        else
            newL ← max{L[i − 1], L[i]} and newE ← 0
            if a_i = b_j then newL ← oldL + 1 and newE ← oldE endif
            if L[i − 1] = newL then newE ← newE + E[i − 1] endif
            if L[i] = newL then newE ← newE + E[i] endif
            if oldL = newL then newE ← newE − oldE endif
            oldL ← L[i], oldE ← E[i], L[i] ← newL, and E[i] ← newE
        endif
    endfor
endfor
```

References

[1] Stephen F. Altschul and Bruce W. Erickson. Optimal sequence alignment using affine gap costs. *Bulletin of Mathematical Biology*, 48(5/6):603–616, 1986.

[2] Osamu Gotoh. Optimal sequence alignment allowing for long gaps. *Bulletin of Mathematical Biology*, 52(3):359–373, 1990.

[3] Ronald I. Greenberg. Fast and simple computation of all longest common subsequences. Eprint arXiv:cs.DS/0211001, Comp. Sci. Res. Repository, `http://arXiv.org/abs/cs.DS/0211001`, 2002.

[4] Ronald I. Greenberg. Bounds on the number of longest common subsequences. Technical Report arXiv:cs.DM/0301030, Comp. Sci. Res. Repository, `http://arXiv.org/abs/cs.DM/0301030`, 2003.

[5] Saul B. Needleman and Christian D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48:443–453, 1970.

[6] Claus Rick. Efficient computation of all longest common subsequences. In *Proceedings of the 7th Scandinavian Workshop on Algorithm Theory*, volume 1851 of *Lecture Notes in Computer Science*, pages 407–418. Springer-Verlag, 2000.

[7] David Sankoff. Matching sequences under deletion/insertion constraints. *Proceedings of the National Academy of Science USA*, 69(1):4–6, January 1972.

[8] Robert A. Wagner and Michael J. Fischer. The string-to-string correction problem. *Journal of the ACM*, 21(1):168–173, 1974.