Problem Set 2 - Exercise 5

Fabio Pernisi

January 2024

Problem Description

Given two binary sequences of lengths n and m, respectively, find the length of their longest common subsequence.

Running time $O(n^2 + m)$.

Solution

The problem at hand is a modification of the classical Longest Common Subseuqence (LCS). Instead of storing the LCS length directly, our dynamic programming table $dp[i][\ell]$ represents the *smallest index j in sequence B for which the LCS of $A[1..i]$ and $B[1..j]$ has a length of at least ℓ.*

This redefinition allows us to use the properties of binary sequences to update the table entries more efficiently.

To understand the optimization, consider the nature of binary sequences: each element is either '0' or '1'. This allows us to preprocess B to quickly determine the next occurrence of a matching character (obtained via 2). For each position in B, we can store the next index where '0' and '1' appear. With this information, when we wish to extend an LCS by one element (say $A[i] = 1$), we can directly jump to the next occurrence of '1' in B beyond the current LCS boundary.

The table is filled by iterating over the lengths of possible subsequences in A and for each, determining the minimum extension needed in B. The update rule for $dp[i][\ell]$ leverages the precomputed positions of '0's and '1's in B, stored in the array T, and uses the previous subsequence information to find the smallest index j efficiently.

By only considering extensions of the LCS when a match is found and quickly skipping non-matching characters, we avoid the $O(m)$ per-entry cost typical of standard LCS algorithms, thus achieving the $O(n^2 + m)$ running time.

Algorithm 1 Binary Longest Common Subsequence

```
function BinaryLCS(A, B)
    n ← length of A
    m ← length of B
    T ← PreprocessB(B)                                  ▷ Use the preprocessing function described earlier
    Let dp be (n + 1) × (n + 1) array filled with ∞
    dp[0][0] ← 0
    for i ← 1 to n do
        for l ← 1 to i do
            j ← dp[i − 1][l − 1]
            nextJ ← T[j]
            while nextJ ≤ m and B[nextJ] ≠ A[i − 1] do
                nextJ ← T[nextJ]                        ▷ Find the next occurrence of A[i-1]
            end while
            if nextJ ≤ m then
                dp[i][l] ← min(dp[i − 1][l], nextJ + 1) ▷ Add 1 to move to the position after nextJ
            else
                dp[i][l] ← dp[i − 1][l]                 ▷ No occurrence found, copy previous value
            end if
        end for
    end for
    return max({l | dp[n][l] ≠ ∞})
end function
```

Algorithm 2 Binary Longest Common Subsequence Preprocessing

```
function PreprocessB(B)
    m ← length of B
    Let T be a new array of length m filled with m + 1
    for i ← m − 1 down to 0 do
        if i == m − 1 or B[i + 1] ≠ B[i] then
            T[i] ← i + 1
        else
            T[i] ← T[i + 1]
        end if
    end for
    return T
end function
```