Department of Mathematics and Computer Science
University of Southern Denmark

December 13, 2016
RF

Dynamic Programming: Hirschberg's Trick

"Hirschbergs trick" is a method for saving space in dynamic programming algorithms, invented in 1975 by Daniel Hirschberg. We here illustrate it using the standard dynamic programming algorithm for Longest Common Subsequence (LCS) of two strings.

Longest Common Subsequence

A *subsequence* of a string $X = x_1x_2x_3 \dots x_m$ is simply some of the characters of X, taken in order. In other words, it is a string $x_{t_1}x_{t_2}x_{t_3} \dots x_{i_k}$ where $i < j$ implies $t_i < t_j$. A *common subsequence* of two strings $X = x_1x_2x_3 \dots x_m$ and $Y = y_1y_2y_3 \dots y_n$ of lengths m and n, respectively, is a string which is a subsequence of both X and Y. Let $\mathrm{LCS}(X, Y)$ be the length of a *longest common subsequence* of X and Y (there may be many longest common subsequences, but they all have the same length).

The value $\mathrm{LCS}(X, Y)$ may be seen as a similarity measure between strings. The problem has applications in bioinformatics, and forms the basis of file comparison utility programs such as `diff` and methods for reconciling changes between file versions in version control systems such as Git.

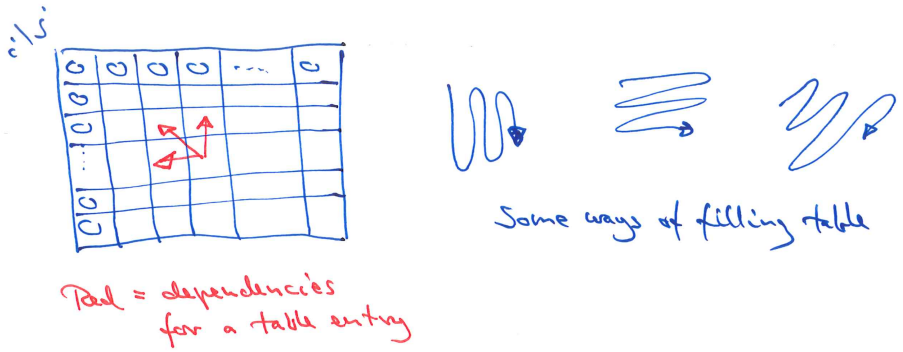
The classic dynamic programming algorithm is based on the following recursion, where $\mathrm{lcs}(i, j)$ is shorthand for $\mathrm{LCS}(X[1..i], Y[1..j])$ for $1 \leq i \leq m$ and $1 \leq j \leq n$:

$$\mathrm{lcs}(i,j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ \mathrm{lcs}(i-1, j-1) + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j \\ \max\{\mathrm{lcs}(i-1, j), \mathrm{lcs}(i, j-1)\} & \text{if } i, j > 0 \text{ and } x_i \neq y_j \end{cases}$$

Let S be a longest common subsequence of $X[1..i]$ and $Y[1..j]$. The first case of the recursion is correct because at least one of the strings is empty, hence S is empty. The second case is correct because we can assume that S has a

last character corresponding to both x_i and y_j, since any longest common subsequence with last character corresponding to only one of these can be converted to one corresponding to both (by changing the corresponding character in one of the strings), whereas a common subsequence ending in none of these cannot be longest, as it can be extended. Removing the last character of S, the remainder of S will lie in both $X[1..(i-1)]$ and $Y[1..(j-1)]$, and must be longest common subsequence there (else S itself could be be extended, hence would not be longest). The third case is correct because the last character of S cannot correspond to both x_i and y_j, since these are different. Hence, S must lie in either $X[1..(i-1)]$ and $Y[1..j]$, or in $X[1..i]$ and $Y[1..(j-1)]$. As no common subsequence in either of these two pairs of strings can be longer than S (else, S would no be a longest in $X[1..i]$ and $Y[1..j]$), the maximum gives exactly $\mathrm{lcs}(i,j)$.

From this recursive formula, the $O(mn)$ size table of $\mathrm{lcs}(i,j)$ for $1 \leq i \leq m$ and $1 \leq j \leq n$ can be filled in one of several ways



using $O(1)$ time per table entry, hence $O(mn)$ time in total. The value $\mathrm{LCS}(X,Y)$ is simply the last table entry $\mathrm{lcs}(m,n)$. To find this, it is enough to store $O(\min\{m,n\})$ table entries during the execution of the algorithm, since only the last row (or column, or diagonal, depending on choice of table fill order), is needed for finding the next row. Hence, the space usage for finding the *length* of a longest common subsequence is $O(\min\{m,n\})$.

For finding an *actual longest common subsequence*, the usual method is to backtrack from the last table entry, moving one diagonal step backwards in the third case (and outputting a last character of the longest common subsequence), while moving one horizontal or vertical step backwards in the second case (outputting no character). The backtracking stops when the

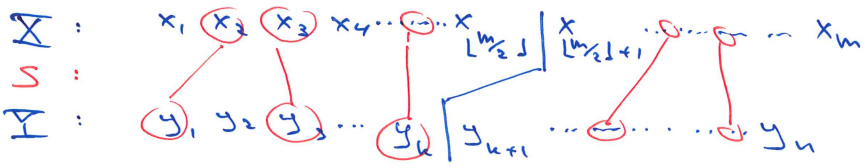
first case is met. However, this method requires the entire filled table to be kept in order to choose between horizontal and vertical steps in case two, since the backwards path taken is not known beforehand. Hence, with this method, time as well as space usage of finding an actual longest common subsequence is $O(mn)$. Hirschberg's trick is a method for reducing the space to $O(\min\{m, n\})$ while keeping the time at $O(mn)$.

First note that the recursion above is based on scanning the strings X and Y from left to right. An entirely symmetric argument based on scanning from right to left gives the following recursion, where where lcs'(i, j) is shorthand for LCS$(X[i..m], Y[j..n])$ for $1 \leq i \leq m$ and $1 \leq j \leq n$:

$$\text{lcs'}(i,j) = \begin{cases} 0 & \text{if } i = m \text{ or } j = n \\ \text{lcs'}(i+1, j+1) + 1 & \text{if } i < m,\ j < n \text{ and } x_i = y_j \\ \max\{\text{lcs'}(i+1, j), \text{lcs'}(i, j+1)\} & \text{if } i < m,\ j < n \text{ and } x_i \neq y_j \end{cases}$$

Using this method we can fill the table in the other direction, and the value LCS(X, Y) is the first table entry lcs'(m, n). Nothing new is achieved yet.

Hirschberg's idea is to use divide and conquer on one of the strings. We here explain it using divide and conquer on X. Look at a longest common subsequence S for X and Y. Some characters (possibly none) of S correspond to characters from $X[1..\lfloor m/2 \rfloor]$ and the rest correspond to characters from $X[\lfloor m/2 \rfloor + 1..m]$. Let y_k be the rightmost character in Y which via S corresponds to a character in $X[1..\lfloor m/2 \rfloor]$ (if no characters of S correspond to characters from $X[1..\lfloor m/2 \rfloor]$, let k be zero).



We claim that

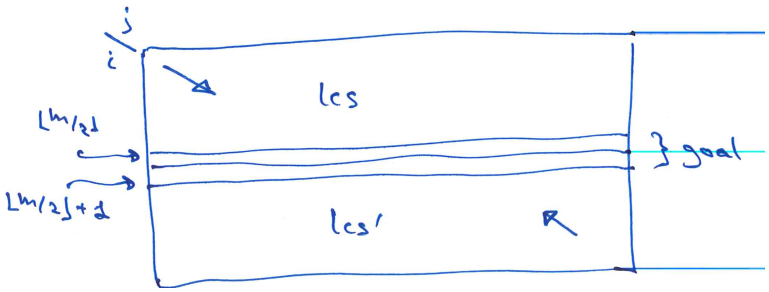
$$\text{LCS}(X,Y) = \max_{0 \leq l \leq n} \{\text{lcs}(\lfloor m/2 \rfloor, l) + \text{lcs'}(\lfloor m/2 \rfloor + 1, l + 1)\}$$

Clearly, by the definition of k a first part of S is a common subsequence of $X[1..\lfloor m/2 \rfloor]$ and $Y[1..k]$, and the remaining part of S is a common sub-

sequence of $X[\lfloor m/2 \rfloor + 1..m]$ and $Y[k+1..n]$. This shows the left hand side smaller than the right hand side. Conversely, for any l a common subsequence of $X[1..\lfloor m/2 \rfloor]$ and $Y[1..l]$ concatenated with a common subsequence of $X[\lfloor m/2 \rfloor + 1..m]$ and $Y[l+1..n]$ is a common subsequence of X and Y. This shows the left hand side larger than the right hand side. Hence, the claim is proved.

This claim shows that to find an actual longest common subsequence of X and Y, it suffices to: *i*) first find the value k' of l which maximizes the right-hand side, *ii*) then find a longest common subsequence of $X[1..\lfloor m/2 \rfloor]$ and $Y[1..k']$ and a longest common subsequence of $X[\lfloor m/2 \rfloor + 1..m]$ and $Y[k'+1..n]$, and *iii*) finally concatenate them.

To perform *i*), we find $\text{lcs}(\lfloor m/2 \rfloor, l)$ for all $0 \leq l \leq n$. This is the middle row in the table for $\text{lcs}(i,j)$. We also find $\text{lcs'}(\lfloor m/2 \rfloor + 1, l+1)$ for all $0 \leq l \leq n$. This is the middle row in the table for $\text{lcs'}(i,j)$.



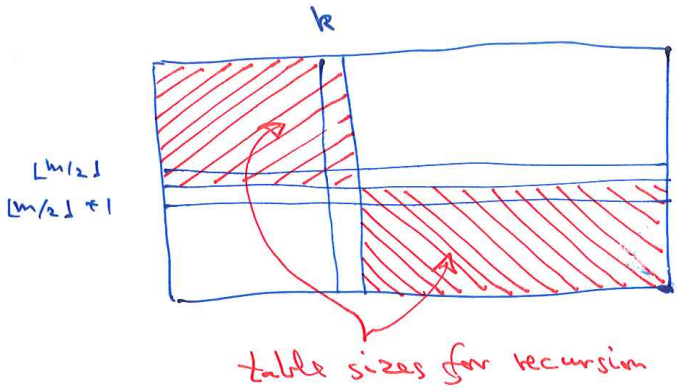
Both can be found in $O(mn)$ time and $O(n)$ space. Afterwards, k' can be found in a scan of the two rows in $O(n)$ time.

To perform *ii*), we recurse on the instance $X[1..\lfloor m/2 \rfloor]$ and $Y[1..k']$, and the instance $X[\lfloor m/2 \rfloor + 1..m]$ and $Y[k'+1..n]$. The longest common subsequences returned by each call are contatenated to form the output. A base case is when the length of the first string becomes one (which will be reached eventually when repeatedly halving an integer length). This string is a single char, and an actual longest common subsequence between it and any other string is either empty or a single character, which can be determined by a scan of the other string. Another base case is when the length of the second string becomes zero, in which case the empth string is returned.

Performing *iii*) is straight-forward if the output is returned as a linked list (since the length of the longest common subsequence is known before the

recursive steps, it is not hard to see that it is also easy to return the output as an array).

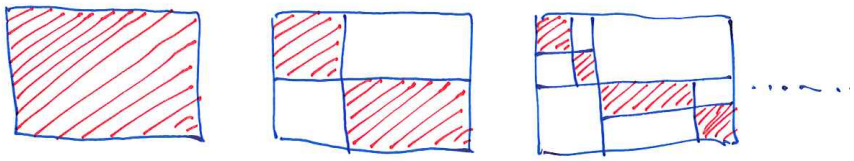
We now analyse the time. In the recursion tree of the method, the local work of the root node is the $O(mn)$ during i), which is the area of the table it needs to fill out. As is clear from the following figure, the combined area of the tables of its children (and hence their combined work) is a factor of two smaller.



This is the crux of the method. From this, it follows that the sum of the local work across one level of the recursion tree is a factor of two smaller than the sum of the local work across the previous higher level of the recursion tree. As the local work of the root is $O(mn)$, the combined work summed over the entire recursion tree is bounded by

$$O(mn) \cdot \sum_{i=0}^{\infty} (1/2)^i = O(mn) \cdot 2 = O(mn)$$

Below is an illustration of the table sizes for each level of the recursion. The red area for each box depicts the total work on one level of the recursion tree.



Additionally, there is scan work at the leaves of the recursion tree (the base cases), but all scanning in the leaves take place on non-overlapping parts of the string, hence in total this scan work is bounded by $O(n+m)$.

The space usage is bounded by the recursion stack and the space usage of the active node in the recursion tree. Since each node on the stack contains two string indices for each of the two strings, this former is $O(\log m)$. The latter is bounded by the space usage of the root, which is $O(n)$. Since we may choose either of the input strings to be X, this means a space usage of $\min\{n+\log m, m+\log n\}$, which can be seen to be $O(\min\{m,n\})$ in the very realistic case that $\max\{\log m, \log n\} \leq \min\{m,n\}$