National University
of computer and emerging sciences

Time Complexity Analysis of Algorithm

Submitted To:-

Respected: Ma'am Noor

Course : Design and Analysis of Algorithms

Submitted By:-

Name : Rizwan Haidar

Roll No # i180536

Section : E

Department: Computer Science

Date : 16-06-2020

Efficient Algorithms for Longest Common Subsequences & Dynamic Programming for LCS

Complexity Analysis

Dynamic Programming for Longest Common Subsequence

In order to find out the complexity of brute force approach, we need to first know the number of possible different subsequences of a string with length n, i.e., find the number of subsequences with lengths ranging from 1,2,..n-1. Recall from theory of permutation and combination that number of combinations with 1 element are nC1. Number of combinations with 2 elements are nC2 and so forth and so on. We know that nC0 + nC1 + nC2 + … nCn = 2n. So a string of length n has 2n-1 different possible subsequences since we do not consider the subsequence with length 0. This implies that the time complexity of the brute force approach will be O(n * 2n). Note that it takes O(n) time to check if a subsequence is common to both the strings. This time complexity can be improved using dynamic programming.
It is a classic computer science problem, the basis of diff (a file comparison program that outputs the differences between two files), and has applications in bioinformatics.

Constrained Longest Common Subsequence

1. Introduction

Lets say two strings $A = a_1a_2a_3 \cdots a_m$ and B= $b_1b_2b_3 \cdots b_n$, the *longest common subsequence* (LCS) problem is that of finding the longest common part of *A* and *B* by deleting no or more characters from *A* and *B then* also finding a D of length , by comparing the LCS from A and B with C as a constrained string or length r. D will be known as Constrained Longest Common Subsequence where t is a common part in A, B and C(May be full as it is a constraint)

Problem:

Given two strings *A* and *B* and a constraint sequence *C* with lengths *m*, *n* and *r* respectively, the *Constrained longest common subsequence* (CLCS) problem is that of finding the LCS of *A* and *B* containing *C* as a subsequence. In 2003, the very first algorithm that was proposed has complexity $O(m^2n^2r)$.Meanwhile same year, another algorithm was proposed that had $O(mnr)$ time and space complexity. Recently, Gotthilfet al, Chen and Chao proposed the related variant which excludes the given constraint as a subsequence.

Definition:

Given two strings *A* and *B* of lengths *m* and *n*, respectively, and a constraint sequence *C* formed by ordered strings $(C^1, C^2, C^3, \ldots, C^l)$ of total length *r*, where C^i is called the *i*th partition of the constraint and each $C^i = c^i\, c^i \cdots c^i$, the SSCLCS problem is that of finding the LCS of *A* and *B* such that *D* contains substrings $C^1, C^2, C^3, \ldots, C^l$ and the partition order is retained.

Consider an example $A = atcatatgag$, $B = atcatctagg$ and $C = (acat)$. *acat* is an CLCS of the second variant, but it is not the first one. Here, we only consider the monotonically increasing case. If two neighboring partitions have the containing relation in the CLCS, it means one of the partition is a substring of the other. In this case, there is no use for the shorter string, and we can spend $O(r^2)$ time to preprocess the input constraints to filter the contained ones out. The required time is varies from $O(mnr)$ to $O(mn + (m+n)(|\Sigma| + r)$

Proof

Let's say C $c_1c_2c_3...c_r$. Chen and Chao proposed an algorithm with $O(mnr)$ time for solving the string inclusion CLCS problem by calculating a 3D lattice directly with the dynamic programming technique applying to A, B and C . the string inclusion CLCS problemis a special case of the SSCLCS problem with only one partition. Because many cells in their lattice are not used.

For example, consider A= *atcatatgag*, B = *atcatctagg* and C = *tag*. We have a_2, b_5, c_1, so we can find the best SSCLCS containing C starting at (2, 5) by jumping through (4, 8), (8, 9). On the other hand, we can wait until the last character of C is matched, and then we find the nearest occurrence of the previous character reversely. Since we perform the dynamic programming approach, we should not refer to cells that have not yet been calculated. Thus, we will perform the matching process in the backward (reverse) way.

We use LCS(S1, S2) to denote the LCS between S1 and S2 and LCS(S1, S2) to denote its length. Ai..j is also used to denote the substring of a string A starting at position i and ending at position j. It is easy to obtain the following fact.

The PrevMatch table can be constructed in O(|S| |Σ|) time and space, where S denotes the input string and Σ denotes the alphabet set of S. A PrevMatch table for A = atcatatgag is shown in output, where −1 means that the character never appears. In actual PrevMatch is opposite to the NextMatch Table. Then we have to create the Startpos table using the PrevMatch table. PrevMatch table can be constructed in O(|S| |Σ|) time and space, where S denotes the input string and Σ denotes the alphabet set of S. he time required for constructing the two StartPos tables is O((m+n)r), since each position requires at most r lookups in the PrevMatch table and each lookup takes only constant time.
Then one pass is made through Dynamic Programming and second pass is made through the below mentioned formulae. While both of them will take **O(m*n)**
As mentioned extracting values from Startpos table will take O(1) time

$$M[i,j,1] = \max \begin{cases} -\infty & \text{if } i \le 0 \text{ or } j \le 0; \\ M[i-1,j-1,1]+1 & \text{if } a_i = b_j; \\ M[\zeta_A[i]-1, \zeta_B[j]-1, 0]+r & \text{if } a_i = b_j = c_r; \\ M[i-1,j,1] & \\ M[i,j-1,1] & \text{otherwise.} \end{cases}$$

In summary, we first construct the PrevMatch tables for A and B in **O((m + n) |Σ|)** time and space. Second, we use the PrevMatch tables to construct the StartPos tables ζA and ζB in O((m + n)r) time and space. With ζA and ζB, each cell in the M table can be obtained in constant time. So the time complexity of our algorithm is **O(mn + (m + n)(|Σ| + r)),** which improves a lot on that of Chen and Chao's method with **O(mnr)** time. Our space complexity is **O(mn + (m + n)|Σ|).**

Apart from this analysis I have provided 1022 examples for Dynamic Programming Algorithm and 1002 examples for Constrained Longest Common Subsequence Algorithm. Which shows that my implemented code's correctness.