FIT2004 S1/2019

Assignment 2: Analysis

Nicholas Chong (29808146)

Introduction

This document will give a brief outline of my solution along with the worst-case space and time complexity of my solution. Each section will be about a single task in the assignment.

For reference, let n and m be the size of the first and second text respectively, N be the number of words in the dictionary, M be the maximal size of the words in the list and k be the size of the input string in Task 2. Furthermore, assume that I am discussing the **worst-case complexity for both time and space complexity** for any operations.

Task 1: Finding the longest subsequence of common alphabets

The function works by retrieving the two words from the file and creating a table with dimensions based on the length of the first word and the second word (adding 1 to each for a null row and column). For example, if the two messages were “abcde” and “cdgfee”, the table would have (5 + 1 = 6) rows and (6 + 1 = 7) columns.

Each element in the table represents the length of the longest common subsequence out of two substrings of the original words. Using our previous example, an element in the 3rd row and 4th column would contain the length of the longest common subsequence of “abc” and “cdgf”. We fill out the table based on:

Base case: DP[i][j] = 0if i or j is 0

Recurrence relations:

DP[i][j] = 1 + DP[i-1][j-1]if the letters at the position match

DP[i][j] = max(DP[i-1][j], DP[i][j-1]) otherwise

There will never be a common subsequence of two words if one of the words are empty, so it will always have a length of 0. If the two letters at the position match, that means we have found another letter to add to our building subsequence and so the length increases by 1 and we add the optimal solution as if the letters weren’t there. If the two letters at the position do not match, we consider the better solution of having the first word without the newest letter and the second word without the newest letter.

We then acquire the longest common subsequence using backtracking. Starting at the very bottom right of the table, we add the letter if the elements to the left and above of our current element are the same AND that value being different to the top-left element. If not, we move towards the element with the higher value. This continues until we reach the top-left of the table, getting our decrypted word.

Worst-case Time Complexity

We begin by opening the file's contents and retrieving both lines which will take O(n) and O(m) to iterate and retrieve. To create the table of None values, we make a row of size m which will take O(m). This is done O(n) times. We iterate over every element in the table (O(nm)) and perform operations regardless of the size of the input so it will be O(1). We perform backtracking from the bottom-right element towards the top of the table which will take O(nm + k) = O(nm) operations, where k is the extra operations in moving left and up.

Overall, the worst-case time complexity is …

O(n) + O(m) + O(nm) + O(nm * 1) + O(nm) = **O(nm)**

Worst-case Space Complexity

Holding the first and second words in a variable will take O(n) and O(m) space, respectively. Making a table based on the value of n and m effectively means that our table will be of size O((n + 1) * (m + 1)) = O(nm).

Overall, the worst-case space complexity is…

O(n) + O(m) + O(nm) = **O(nm)**

Task 2: Words separating

The function starts by retrieving every word in the dictionary from the file. We then make a dynamic programming table with the number of rows as the size of the largest word in the dictionary (+ 1) and the number of columns as the size of the decrypted message (+ 1).

The vertical axis (y-axis) represents the length of the largest word so far that can exist in the table. The horizontal axis (x-axis) represents the current letter in question. Overall, each element in the table represents the dictionary word that the letter is associated with, given the specified max number of words that can exist.

We loop over every element in the table and fill the table based on the base case and recurrence relation.

Base case: DP[i][j] = 0 if i or j is 0

Recurrence Relations:

k = j → j – i : DP[i][k] = location if word from i → j is a word in dictionary

DP[i][j] = DP[i-1][j] otherwise

For example, at the row representing a word of length 4, we make a word from our current position with 3 spaces back to our current position. If this word is in the dictionary, we set the last 4 elements to the position of the dictionary word. If it is not in the dictionary, we simply take the value of the element above. By doing all of this, we are maximising the word length to break our message into.

Once the table is filled out, we iterate over the last row and essentially add a space whenever the numbers change (suggesting that the word is different) OR whenever the length of the current word built is bigger than the word it is meant to be (suggesting that the multiple instances of the same word is being repeated).

Worst-case Time Complexity

Getting the words from the dictionary file will take O(N) time, having to retrieve every line from the dictionary. We then get the largest word in the dictionary, which means we iterate over every letter of the dictionary, taking O(NM) time. Making the table where the rows are based on the length of the largest word in the dictionary (M) and the columns are based on the length of the input message (k) will take O(kM) time in creating. We have to iterate over every element in that table O(kM) and in every iteration, have to check if the word belongs in the dictionary, which will take O(NM) time, this means that filling out the table will take O(kM . NM) time. Lastly, we iterate over the last row, O(k) to form the message.

Overall, the worst-case time complexity is …

O(N) + O(NM) + O(kM . NM) + O(k) = **O(kM . NM)**

Worst-case Space Complexity

Containing the dictionary's contents will be O(NM), having to hold all N words with maximal size M. Making the table with the number of rows based on the length of the largest dictionary word (M) and columns based on the input string length (k) will mean that the table will be of O(kM) size. Getting the output message will hold a word of size O(M + j) = O(M) where j is a constant for the spaces in the message.

Overall, the worst-case space complexity is …

O(NM) + O(kM) + O(M) = **O(kM + NM)**

END OF FILE