CSCI311-DNAProject-Team15

Per Astrom, Caroline Frank, Christina Steinberg, and Will Zhao

November 3, 2021

Introduction

We implemented a basic sequence alignment system using four algorithms: Longest Common Substring, Longest Common Subsequence, Edit Distance, and the Needleman-Wunsch Algorithm. The Longest Common Substring and Longest Common Subsequence algorithms have the added functionality of being able to produce the longest common substring or longest common subsequence. Below are descriptions of the algorithmic structure and runtime for each of our four algorithms and notes regarding the changes we made to our code after the demonstration.

1 Longest Common Substring

The Longest Common Substring algorithm is to find the longest string that is a substring of multiple strings. It is fed two strings, x, y, that are compared. An array called c is initialized of size (length of string x) + 1 by (length of string y) + 1. The purpose of this array is to store the answers to our substrings. We have two nested for loops that iterate through each letter of the string. We have i as the position in string x and j as the position in string j. There nested for loops are $O(mn)$ where m is the length of string x and n is the length of string y. We have (m+1)(n+1) cells that each have $O(1)$ work.

We have two cases:

Case 1: The characters in both strings are the same. Cell [i][j] in c is set to the value of the cell to the upper left plus one. There is a nested if for this first case. If the cell is greater than the max, then maxi and maxj are set to i and j so that the new max value is stored.

Case 2: If there are no more characters to go through, then c[i][j] is 0.

Finally, we find the longest sub string's last letter and set i to maxi and j to maxj. We set to longest substring to the new string. We go back up the diagonal until the string is complete. Lastly, we return the longest common substring and the algorithm is complete. Finally, adding the runtimes together we get O(1) +O(mn)O(1) +O(min(m, n)) giving us a runtime of O(mn).

2 Longest Common Subsequence

The Longest Common Subsequence (LCS) algorithm is fed two strings, x, y, to compare. It then initializes two arrays, c and b, of size (length of string x) + 1 by (length of string y) + 1 to store the answers to our subproblems. Array c holds the length of the longest common subsequence for that subproblem and array b holds a character, 'd', 'l', or 'u', indicating whether the subproblem represented in this cell depends on the cell diagonal, to the left of, or above it. Note everything thus far is $O(1)$. We then have 2 nested for loops that begin to iterate through the letters in the string, where i is the position in string x and j is the position in string y. These nested for loops are $O(mn)$ where m is the length of string x and n is the length of string y because we have $(m+1)(n+1)$ cells each with $O(1)$ work.

We have three cases:

Case 1: The characters in both strings are the same. Then, cell [i][j] in c is set to the value of the cell to the upper left plus one and cell [i][j] in b is set to 'd' indicating that this cell depends on the one diagonal to it.

Case 2: The characters do not match and the cell above our current cell is less than the one to the left. Then, cell [i][j] in c is set to the value of the cell to the left and cell [i][j] in b is set to 'l' indicating that this cell depends on the one to the left of it.

Case 3: The characters do not match and the cell above our current cell is greater than the one to the left. Then, cell [i][j] in c is set to the value of the cell above it and cell [i][j] in b is set to 'u' indicating that this cell depends on the one above it.

After our for loops complete, the element in the bottom right corner of our array, c[x len][y len] will hold the length of the longest common subsequence. Now, we can go back through b to extract the longest common subsequence itself.

Finally, adding the runtimes together we get $O(1) + O(mn) * O(1) + O(min(m, n))$ giving us a runtime of $O(mn)$.

3 Edit Distance

The EditDistance algorithm differs from the other algorithms in that it returns the "edit distance" between the two input strings, which measures the cost or number of "edits" for deletion, insertion, substitution operations to transform one string into the other and returns the minimum of these. If and only if both strings are equal, the edit distance is 0.

The cost for each operation required to transform one character to another is stored in a $(n+1)*(m+1)$ matrix in a dynamic programming fashion, where m, n are the lengths of the strings. The end cost is stored in the bottom-right cell (the cell with each string's last character). The implementation is based on the Wagner-Fisher algorithm.

Time complexity: $\Theta(mn)$ - the length of each string multiplied together, as it compares each character in the first string with each character in the other using two for-loops.
Space complexity: $\Theta(mn)$ - this is the size of the dynamic programming matrix

4 Needleman-Wunsch Algorithm

The Needleman-wunsch algorithm has two input string arguments to be compared. We define the length of the first string as m, and the length of the second string n. Once the two strings are passes in, the algorithm initializes a grid of $(m+2)\times(n+2)$. Then the algorithm would initialize the cells of the first row and column to be the sequence starting from -1 to $-m/-n$. Then the algorithm can start to compare each character in the two strings, starting in different positions. If the characters match, a point is added, otherwise a point is taken off. The cell in [m+1][n+1] contains the final result of the similarity score, which is our return.

The algorithm uses a grid of size $(m+2)\times(n+2)$, and each cell is reached by a order, so the runtime to get to each cell is O(1), the total runtime should be O(mn).

The algorithm would have two cases, the first case is a match, that a point is added; the second case is indel, which means insertion or deletion, then a point is subtracted.

Post-Demonstration Debugging and Changes

After the demonstration, we made a couple of changes to debug our program.

1. Our Longest Common Subsequence and Longest Common Substring algorithms were returning the actual subsequence/substring as opposed to a numeric score. We adjusted the way we handle the return of those functions so that the score is equal to the length of the LCS.

2.We added error-handling functionality to deal with invalid inputs.

3.Modifications to switch statements that handles the logic for choosing the algorithm.