CSCI311-S24: DNA Project Team 7

Geoffrey Gaines(gjg013), Hunter Gehman(hig002), Geoffrey Park (ggp003), amci001

March 2024

1 Algorithms

All code implementations of the below described algorithms are available at

$$https://github.com/hig002/CSCI311\_Team7\_DNA\_Sequencing$$

. Visit to run the algorithms for yourself!

1.1 Longest Common Substring

1.1.1 Implementation

The Longest Common Substring algorithm seeks to find the maximum length of a substring that is present in both of the given two strings. Our implementation utilizes dynamic programming to efficiently solve this problem. We construct a 2-dimensional array, *LCStuff*, with dimensions $(m+1) \times (n+1)$, where m and n are the lengths of the two input strings, respectively.

Each cell *LCStuff[i][j]* stores the length of the longest common suffix of the substrings ending at $i-1$ in the first string and $j-1$ in the second string. We initialize the first row and the first column of *LCStuff* with zeros, which represents the case when one of the strings is of length 0. As we fill the table in a bottom-up manner, whenever characters $X[i-1]$ and $Y[j-1]$ match, we set *LCStuff[i][j]* to *LCStuff[i-1][j-1] + 1*. The maximum value found during this process is the length of the longest common substring. To extract the actual substring, we keep track of the cell where the maximum length ends and then backtrack from this cell to extract the substring.

1.1.2 Runtime Analysis

The runtime analysis of the Longest Common Substring algorithm involves examining the nested loop structure of the implementation. For two input strings of lengths m and n, the algorithm iterates through each character of both strings in a nested manner. Since the iteration results in a total of $m \times n$ iterations and each iteration performs a constant amount of work (either setting a cell to 0, setting it to *LCStuff[i-1][j-1] + 1*, or updating the maximum length found so far), the overall time complexity is $O(mn)$.

Moreover, the space complexity of the algorithm is also $O(mn)$ due to the utilization of a 2-dimensional array of size $(m+1) \times (n+1)$. This space is required to store the lengths of longest common suffixes for all substrings considered by the algorithm.

1.2 Longest Common Subsequence

1.2.1 Implementation

The goal of Longest Common Subsequence (LCS) is to identify and count the longest sequence of matching characters between two given inputs of strings. The LCS does not have to retain

continuity, in other words, non-matching characters may exist between characters that do align with the corresponding string. **Optimal Substructure** is exploited and **recursion** plays an important role. In this project, we applied the algorithm to DNA strand sequences, where the characters are limited to $[A, C, G, T]$ but the character count can be in the thousands or more. Similar to Longest Common Substring, **memoization** is used to apply the Dynamic Programming approach. There was also a side quest involving appropriating FASTA file formatting where the DNA characters needed to be separated from their titles. We also needed to truncate what was expected for user to input from absolute path file declaration to a more generalized, simple input such as 'file.txt'.

1.2.2 Runtime Analysis

$O(m \times n)$

Two strings with varying lengths, m, and n are used to construct a 2D array that are filled with subsequences and and saved for later calculations/comparisons. Many if statements returning recursive calls when true as well as the nested for loop inside a for loop to construct rows and columns of the 2D arrays calls for this runtime and is much more efficient than other implementations involving $O(2^{m+n})$.

1.3 Edit Distance - Levenshtein Algorithm

The Edit Distance family contains a few different types of algorithms. Some well-known algorithms include hamming distance, which has applications to data science, and longest common subsequence (LCS), mentioned elsewhere in this paper. However, this implementation focuses on the Levenshtein Distance Algorithm, which allows for the insertion, deletion, and substitution of characters. The implementation details and runtime are discussed below.

1.3.1 Implementation

A dynamic approach made the most sense to make this algorithm most efficient, because otherwise, the algorithm needs to check three branches for each character in each string (VERY slow runtime). Therefore, a 2x2 matrix containing a column for every character in the first string and a row for every character in the second string was constructed. An additional row and column were added to account for empty strings being passed into the algorithm.

From there, the bases cases are if either string is empty, as the number of operations to resolve this will be the number of characters in the non-empty string. Finally, the algorithm recursively calls itself while remembering previous optimal solutions to find an optimal solution for the current cell. It saves the best result, moves onto the next cell, and iterates until the desired cell at the bottom right of the matrix is complete!

1.3.2 Runtime Analysis

For input strings length m, n, we create $m+1$ rows and $n+1$ columns for the dynamic programming solution matrix. Additionally, for each cell, we make exactly three comparisons to other, previously calculated cells (along with an if statement to check for equality). Therefore, the internal section is $O(1)$. This gives us an overall runtime of $T(n) = (m + 1) * (n + 1) * O(1) = O(m * n)$.

1.4 Needleman-Wunsch Algorithm

The Needleman-Wunsch Algorithm is a dynamic programming algorithm as it breaks up the full sequence into smaller subproblems. It assigns a score to every possible alignment between two sequences and finds the combination that produces the higher score.

1.4.1 Implementation

The algorithm uses a matrix of size m x n; m and n represents the size of the DNA sequences being compared. One sequence is represented at the top of the matrix and the other at the left side. Within the scoring system, there are three possibilities: the match score for when the letters of the same index match, the mismatch score for when the letters of the same index don't match and the gap score for when the best alignment is with a letter at a different index (requires a gap). There are different ways to value each score, but the match score is always positive and the mismatch and gap scores will be negative.

The algorithm proceeds through row by row starting at the top. It conducts three score calculations: (1) The diagonal score is calculated by checking if the characters of the current index match; if they do, add +1 to the value of the top left diagonal cell (0 if the start) (2) The top score is calculated by adding -1 to the score at the top cell. (3) The side score is calculated by adding -1 to the score at the left cell. Whichever score is highest is placed at the current table cell. This is continued until the matrix is filled out, at which the score of alignment of the two sequences is located at the bottom rightmost cell of the table.

1.4.2 Runtime Analysis

The runtime of the algorithm is O(m x n). This is because the matrix is of size m x n and at each matrix cell a constant time of 3 calculations are conducted.