CSCI311-F22: DNA Project Team 3

Nate Ahearn (nga004) Sam Baldwin (sdb035)
Nishant Shrestha (ns037) Warren Wang (wmw015)

November 4, 2022

1 Introduction

Running the code will present the user with a request for a file that contains the sequences that the query sequence will be compared to. If the file that is being selected is within the same directory as the .py file, only the filename is necessary, otherwise the full path to the file is necessary. The user will then be prompted with a request for the query file, containing the sequence that is to be compared. The same logic as above for providing the file name applies. If there is an issue in selecting either file, the program will continually request the names of files until valid files are given.

Once the files have been provided, the program asks for input from the user, in the form of an integer from 0-3, to determine which of the algorithms should be run on the two provided files. The algorithm will then print the comparison for each of the sequences within the sequences file against the query file, and the corresponding return value from the algorithm.

After the algorithm that is chosen is finished running, the user is prompted if they want to run another algorithm or exit the program.

2 Longest Common Subsequence

The result of our algorithm suggest that the longest common subsequence when comparing the query sequence to the provided sequences was: NC_045512.2:21563-25384 SARS-Cov-2 - surface spike protein
The process of finding the longest common subsequence is by comparing characters on the extremes of strings, and adding or not omitting them based on if they match. Recursively calling this comparison on smaller and smaller segments of the string, and filling a matrix dimensioned by the lengths of the two strings with their comparisons, either a match incrementing the previously longest substring, which is held in the cell one up and one left regardless of distance from the most recent match, or keeping that value and not incrementing, allows the tracking of the length of the longest common subsequence.

2.1 Runtime

As mentioned above, the algorithm populates a n x m matrix where n and m are the lengths of the two strings provided. Each of the cells takes some constant time to fill, resulting in the total runtime of the algorithm being $O(n * m)$.

3 Longest Common Substring

The result of our algorithm suggest that the longest common substring when comparing the query sequence to the provided sequences was: NC_000011.10:c2161209-2159779 Homo sapiens chromosome 11, GRCh38.p13 Insulin
The process of finding the longest common substring is incredibly similar to longest common substring, but instead of checking single characters, and building from the problem subspace, referring to previously constructed sets regardless of distance, the algorithm checks single characters, and builds series of matching characters, without looking across non-matching gaps. It populates a n x m matrix, with a 0 in any place where characters do not match, and the length of a matching substring if there is a match.

3.1 Runtime

The runtime of this algorithm is incredibly similar to the runtime of the longest common subsequence, given it populates an n by m matrix, where n and m are the lengths of the two strings provided. Each of the cells takes some constant time to fill, resulting in the total runtime of the algorithm being $O(n * m)$.

4 Edit Distance

The result of our algorithm suggest that the shortest edit distances necessary to compare the query sequence to the provided sequences was: V01243.1 Rat gene for insulin 2. This result is shared by Needleman-Wunsch and Smith-Waterman
The specific edit distance algorithm we are using is called the Levenshtein distance. It describes the number of string operations needed to change one string into another. We only allow 3 string manipulation operations: deletion, insertions, and substitutions. The idea is that the lower the number of required string manipulation operations or edit distance, the more similar two strings are.

4.1 Runtime

For time complexity, we are bounded by the dimensions of our matrix that we are building in our iterative solution, so we have a asymptotic runtime of $O(m*n)$ where m, n are the lengths of our two strings that are being compared.

5 Needleman-Wunsch

The overall idea of the Needleman-Wunsch algorithm is to first create a matrix for the two strings we are trying to compare. To utilize this matrix, a "score" must be implemented for matches, alignments, and mismatches. The values our group chose were 1, -5, -1 respectively. We now go back to the big matrix, in this matrix we fill in the top row and left most column with 0 through $-n$ for n being the size of the strings. Next we initialize the maximum point value and then compare point values between indices and the maximum point value to fill out this table. We must do this $O(m * n)$ time with m and n being the size of the input strings. We then return the m, n value in the table which represents the similarity score for that comparison.

5.1 Runtime

The largest computation in this algorithm is the creating and subsequent filling out of the score matrix. We must have $O(nm)$ computations with the m and n value representing the size of each string. Due to all of the other computations taking a pre-determined set amount of computations we can conclude the runtime is $O(nm)$.

6 Smith-Waterman

Our algorithm assists us to identify the most similar local region between the two sequences we compare. Similar to the Smith-Waterman algorithm, our algorithm adopts a scoring system 2, -2, -1 for matches, mismatches, and alignments, respectively. Our scoring matrix is initialize as a $(m + 1, n + 1)$ matrix where the first row and column are all zeros. Unlike Needleman-Wunsch, everytime we come across an overall negative score, we substitute that with a zero. Doing so allows us to restart our algorithm at any point between the two sequences avoiding highly unsimilar regions between our two sequences and ultimately pick out the most simlar local sequence. Furthermore, we fill out our scoring matrix one row at a time. Our algorithm has the following recurrence, given the scoring matrix is S:

$$1 + (-1)^n = max \begin{cases} S[i-1, j-1] + c(a_i, b_j), & if a_i = b_j \\ S[i-1, j] - K, \\ S[i, j-1] - K, \\ 0 \end{cases}$$

Here, is K the gap-penalty and $c(x, y)$ is the match score of matching x element of the first sequence and y element in the second sequence, where the position of the element matters.

6.1 Runtime

Again, since we are filling out the scoring matrix, this algorithm has an asymptotic run-time of $O(m \times n)$, where m and n represent the length of the two sequences we have.