CS5390 - Algorithmic Foundations of Computational Biology

Final Project Report

Jose G. Perez and Sebastian Nunez

Table of Contents:

[**Background and Introduction**](#_koxlwzjed64r) **2**

[Local Alignment: Smith-Waterman](#_qjg91mv43pew) 2

[Global Alignment: Needleman-Wunsch](#_a8udluqnxzyu) 2

[Affine Gap Alignment: Gotoh’s Algorithm](#_vmjynutiihip) 3

[Multiple Sequence Alignment: Center Star](#_8uvb6q93q4iq) 3

[Neighbor Joining](#_tyx98zy4t18m) 3

[BLAST Nearest Neighbors](#_tyx98zy4t18m) 3

[Burrows-Wheeler Transform](#_d5lga6ugar6y) 4

[**Method Implementation**](#_w2zvqte4y8ku) **5**

[Alignment: Smith-Waterman and Needleman-Wunsch](#_y03nvmw6xoyw) 5

[Affine Gap Alignment: Gotoh’s Algorithm](#_tb3sfcoympoc) 5

[Multiple Sequence Alignment: Center Star](#_99ntt06asdnz) 5

[Neighbor Joining](#_b4f3ag62w58q) 5

[BLAST Nearest Neighbors](#_xtaahetoem7) 5

[Burrows-Wheeler Transform](#_m2xsv577anyw) 6

[**Results**](#_8m0kx5717n2r) **7**

[Local Alignment: Smith-Waterman](#_kk4rgittp174) 7

[Global Alignment: Needleman-Wunsch](#_eukhoyv7t89s) 8

[Affine Gap Alignment: Gotoh’s Algorithm](#_jzom1gdfxz5) 9

[Multiple Sequence Alignment: Center Star](#_u8gx4snddw7v) 10

[Neighbor Joining](#_nno0hetnzx7r) 11

[Basic Local Alignment Search Tool: BLAST](#_tmeoz23vdjb) 12

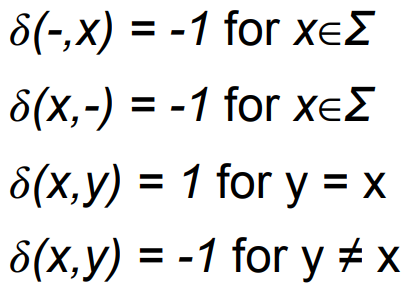
[**Conclusion**](#_jbz3anjf0bfq) **13**

[**Individual Contributions**](#_kzhu2kbx25h7) **14**

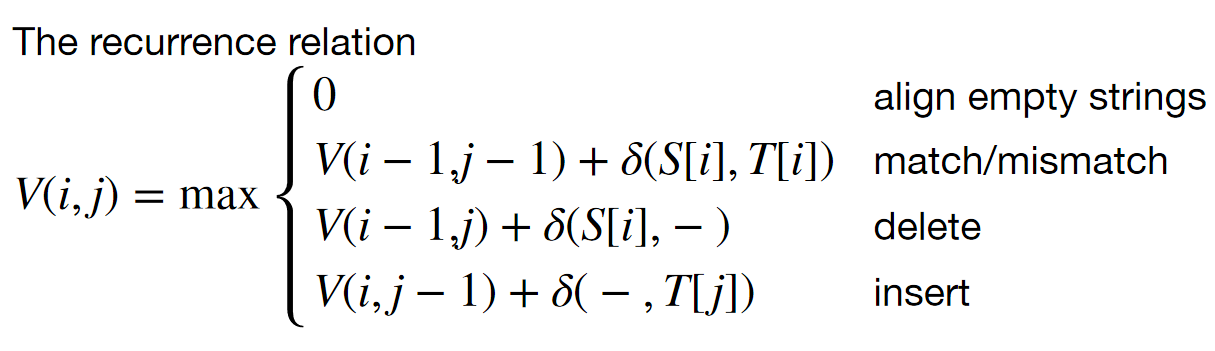
# Background and Introduction

## Local Alignment: Smith-Waterman

Smith-Waterman is an inexact string matching algorithm that takes in two strings as an input and outputs the best substring alignment with the highest alignment score which takes possible insertions, deletions, matches, and mismatches into account. The algorithm uses 4 parameters which are called costs or penalties, the cost of inserting a character, the cost of deleting a character, the cost of matching a character, and the cost of a character being a mismatch. In the figure below, the notation for such penalties is demonstrated. The first 2 formulas are the penalties for insertion and deletion respectively, the 3rd formula is for character matches, and the final formula is for character mismatches. By changing what each formula is equal to one can change the penalties to fit their needs.



Once these parameters have been set, the algorithm uses the following recursion formula to fill a dynamic programming table.

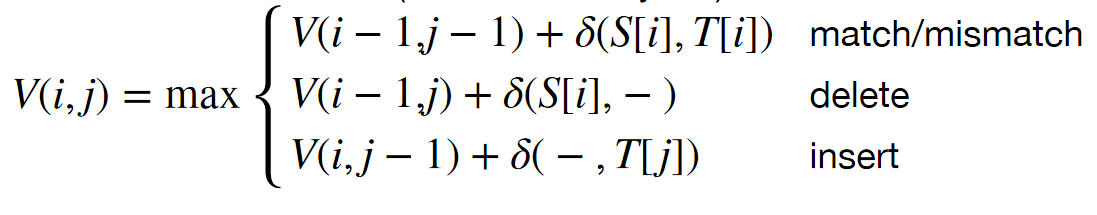


To find the optimal alignment from this dynamic programming table, the whole array must be traversed to find the maximum value as this will mark the place where the optimal substring ends. The traceback will then follow the path taken when filling the dynamic programming table to find the start of the substring, which will be where the first zero appears in the path.

The time the algorithm takes to run is O(nm). N being the size of one input string, and M being the size of the other input string. This is because you need to compare all the characters once with each other to completely fill the dynamic programming table.

## Global Alignment: Needleman-Wunsch

Needleman-Wunsch is an inexact string matching algorithm that takes in two strings as input and outputs the the best alignment of both strings while weighing gaps, indels, matches, and mismatches into account for both strings as a whole regardless of length differences. It utilizes the same type of cost penalty parameters used in Smith-Waterman. The difference is in how the dynamic programming table is filled, the recurrence relation is:



Once these formulas have been calculated Global Alignment will choose the max value based on the characters that are being compared at that moment to choose where once should be deleted, inserted, mismatched or matched. Once the current characters have compared either x or y shall be compared to the rest of the characters in the opposing string before advancing to the next character in the current string. This is to ensure that each character is compared to each other character for the optimal solution stored in the last index of the matrix.

The time the algorithm takes to run is O(nm). N being the size of one string M being the size of the other. The time it takes is nm because each character must be compared to each other character.

## Affine Gap Alignment: Gotoh’s Algorithm

Gotoh’s algorithm is also known as affine gap alignment because it is very similar to global alignment but it penalizes gaps with gap penalties. The algorithm is focused on taking these gaps into account and is run across 3 matrices to find the optimal solution. The solution will again be stored at the bottom right but the traceback must be all along all 3 matrices. One of the matrices is used to score insertions, one for deletions, and the final is matches.

The time the algorithm takes to run is O(nm) and the space is also O(nm) because it is constant time operations but the matrix needs to be traversed completely.

## Multiple Sequence Alignment: Center Star

Center Star is used for multiple sequence alignment because by using this method we can come up with a base string. This base string can then be used to help align strings according to the original alignment. Any indels and gaps can also be tracked easily know and the differences become clear between the other strings. The base string is just used to help keep track of those comparisons.

The time it takes is O(k2n2) because each string takes n2 to be compared and k2 comparisons must be made. This runtime being for computing all pairwise scores.

## Neighbor Joining

Neighbor Joining is a multiple sequence alignment method that allows strings to be grouped together in clusters and integrated together according to either an average, minimum, or maximum weights according to what the user sets. They can then begin integrated clusters together based off the distance from the clusters to the others. From there the weights will be adjusted as 2 clusters combined to make keep track of the new weight and changes from one cluster to another. This is done until one giant cluster is created.

The algorithm takes O(n3) time due to the matrix slowly decreasing in size but starting at nxn size.

## BLAST Nearest Neighbors

The basic local alignment search tool is used to find what can be replaced for kmers inside of DNA, RNA, or amino acids. It provides a list of all the possible replacements according to the weights it is provided. The weights of a replacement are then combine to see If the replacement is above a certain threshold. If it is above the threshold it will be recommended as a possible replacement.

The runtime will be O(k2n) because it has to compare each character replacement in the kmer to each other possible character replacement in the kmer.

## Burrows-Wheeler Transform

A BWT is a suffix array that makes use of rank and the C array (array for storing locations of each unique char) and only stores the letter prior to the substring in the original suffix array. From there the BWT is used to count occurences using C and the rank function.

The runtime will be constant once the BWT is constructed because all that is done are calculations.

# 

# Method Implementation

## Alignment: Smith-Waterman and Needleman-Wunsch

For my implementation of these alignment algorithms I used Python as the base programming language. The user is prompted to input the two strings which will be aligned, and all 4 penalties (insertion, deletion, match, mismatch) to use when running the algorithm. The user may opt to run the algorithm some default penalties that I have chosen, which set the penalties to insertion & deletion = -0.5, match = 5, and mismatch =-1. I made this design choice because it gives users a quick way of performing alignment without needing to worry too much about the specifics of the algorithm. After inputting the required information, the algorithm is run, and when it finishes the user is presented the results. The results shown are: the time it took the algorithm to run, the dynamic programming table, the alignment score, the alignment, and a table that contains the values of the dynamic programming table as well as arrows that tell you the origin of the value. This allows you to traceback the alignment manually if you wish to.

## 

## Affine Gap Alignment: Gotoh’s Algorithm

For Gotoh’s algorithm the user interface is almost exactly the same as in the previous two alignment algorithms. The only difference being that the parameters which the user can input are different, this time it’s the alpha, beta, match penalty, and mismatch penalty. Once again, the user may opt to run the algorithm with some default values I have chosen for the parameters (alpha=2, beta=0.5, match=5, mismatch=-1). After inputting the required information, the algorithm is run, and when it finishes the user is presented the results. The results shown are: the time it took the algorithm to run, the dynamic programming tables for F, G, and E, the alignment score, the alignment, and 3 tables that contains the values of the dynamic programming tables as well as arrows that tell you the origin of the value. This allows you to traceback the alignment manually if you wish to. A double arrow is used as an indicator to let you know when to jump between tables, and the tables are positioned in such a way that it is very natural to tell which table you are meant to jump to next.t

## Multiple Sequence Alignment: Center Star

For the Center Star algorithm the user is asked to provide the 4 parameters of Needleman-Wunsch, as that’s the alignment algorithm used by my implementation for alignment. The user may opt to use the default values that I set, just like in the Needleman-Wunsch program. The user is then prompted about how many sequences they will be aligning, and after providing a number the user is prompted to input each one. After inputting the sequences, the user is asked if they would like to manually select the center star or let the program automatically calculate it. I added this option to allow the user to experiment with different center stars. After providing the necessary inputs, the program aligns all the sequences through a method called combine\_sequences. After the sequences are combined, redundant columns (places in the sequences where all the sequences have gaps) are removed. Then the user is given the result of the algorithm, the aligned sequences and an indicator of which one is the center star.

## Neighbor Joining

## BLAST Nearest Neighbors

Blast Nearest Neighbors takes the kmer the user wants to replace, the threshold, and the name of a text file containing the weights. The first row and column of the text file are used to store the symbols that are being taken into account in this case it would be the DNA/RNA/Amino acids they are checking. Once the characters are identified all of the possibilities inside of the rows and columns pertaining to the characters are stored. The algorithm then recursively traverses the possibilities by going to the next character possibilities until the last character possibilities is reached then backtracks through each character possibility. The character combinations that meet the threshold are then stored in an array and printed once the recursive method is complete.

## Burrows-Wheeler Transform

Burrows-Wheeler Transform Counting occurrences takes the string the user wants to convert into a BWT then the substring they are searching for occurrences of. The formulas of updating sp using C[c] + rank[sp-1]+1 is then used and ep is updated using C[c] +rank[ep] is used to find if the current string occurrences according to pointers. The difference in pointers is then printed if it is negative there were no occurrences if it is positive that is the amount of occurences.

# 

# Results

## Local Alignment: Smith-Waterman

Experiment:

S1=TTACTGTGTTT and S2=CACCCCTGTG (same as the homework question)

Insertion Cost = -0.5

Deletion Cost = -0.5

Match Cost = 5

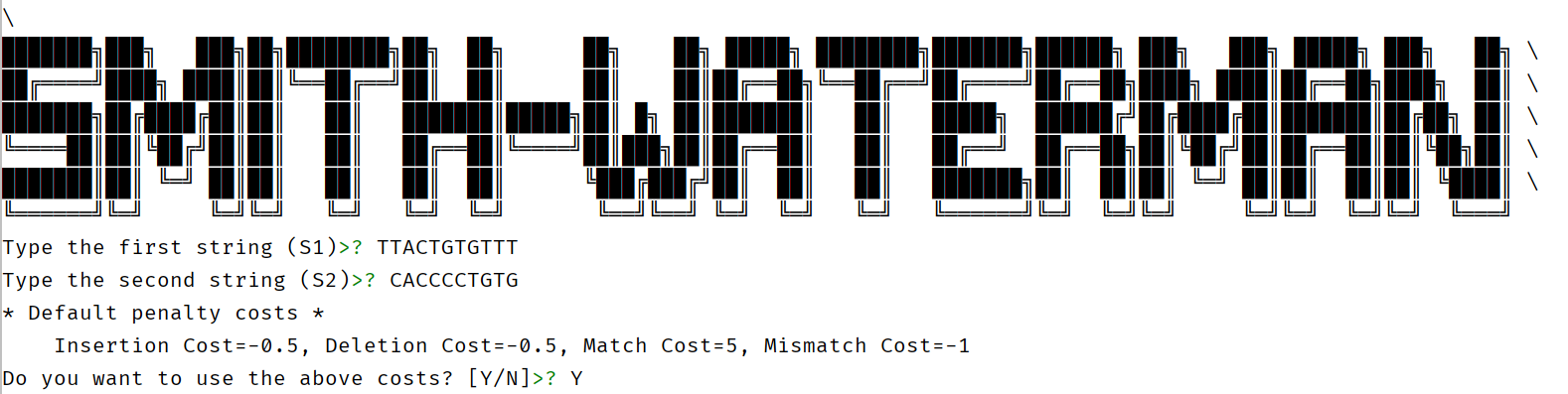
Mismatch Cost = -1

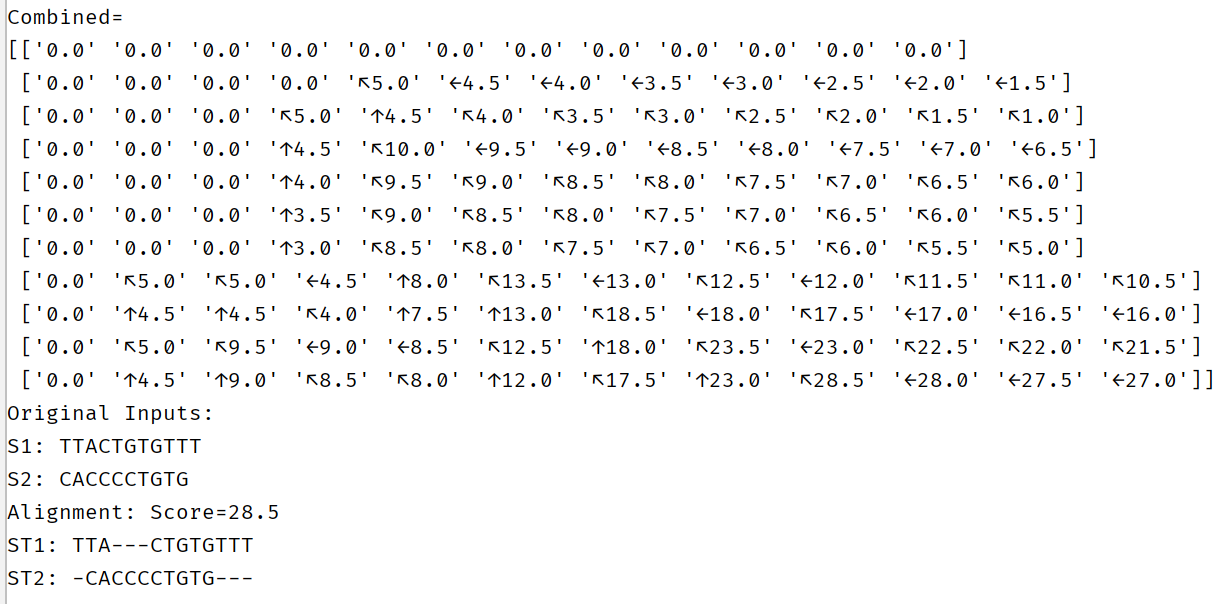
Running Time: 0.0013 seconds

Alignment Score=28.5

ST1: TTA---CTGTGTTT

ST2: -CACCCCTGTG---





## 

## Global Alignment: Needleman-Wunsch

Experiment:

S1=TTACTGTGTTT and S2=CACCCCTGTG (same as the homework question)

Insertion Cost = -0.5

Deletion Cost = -0.5

Match Cost = 5

Mismatch Cost = -1

Running Time: 0.0014 seconds

Alignment Score=25.5

ST1: TTA---CTGTGTTT

ST2: -CACCCCTGTG---

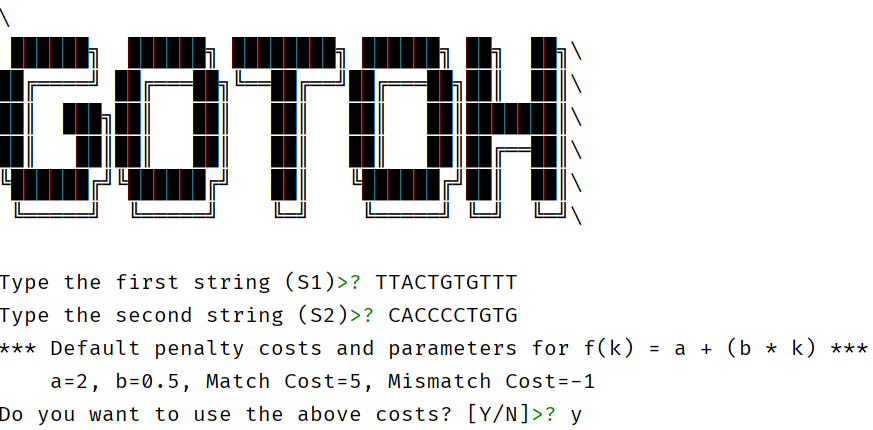
## 

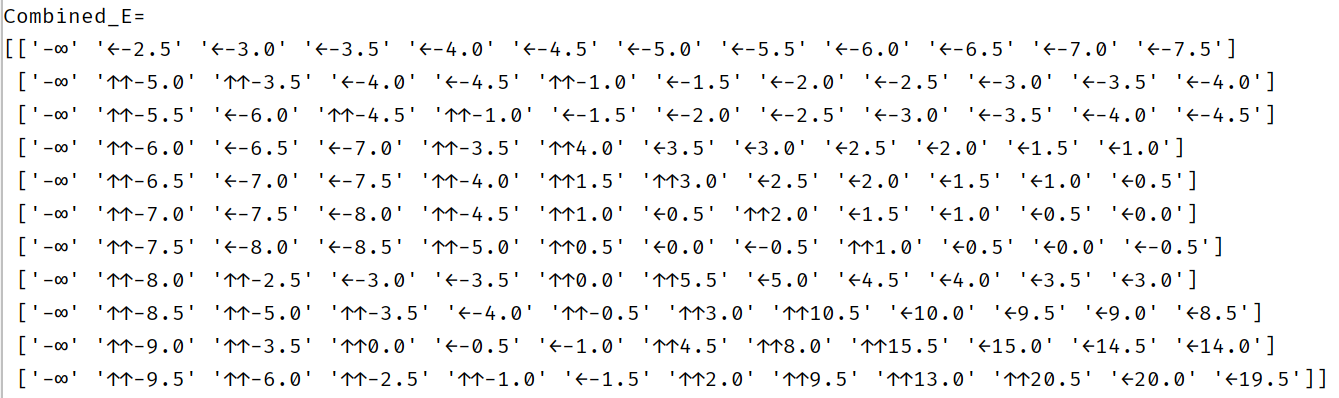
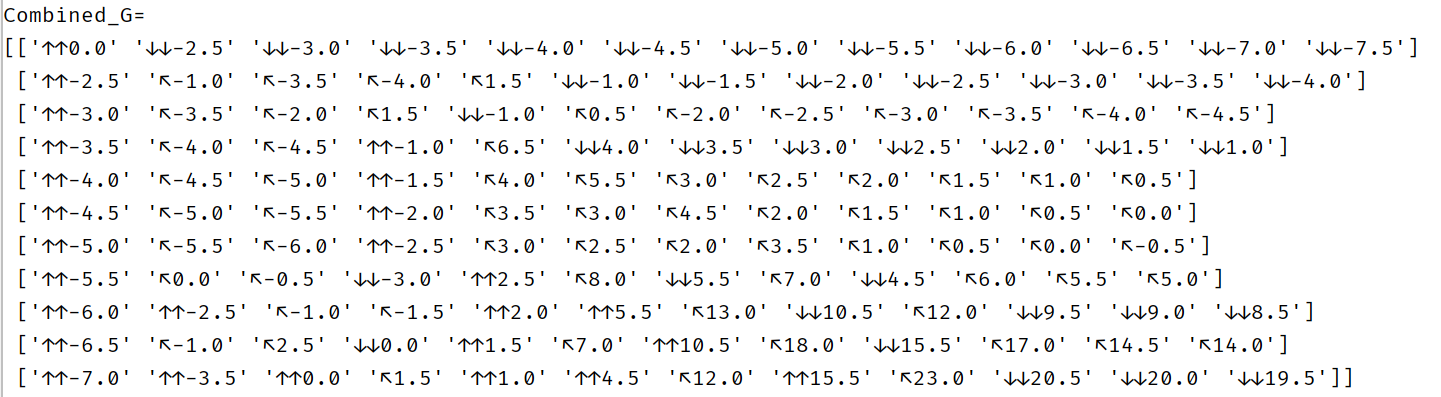
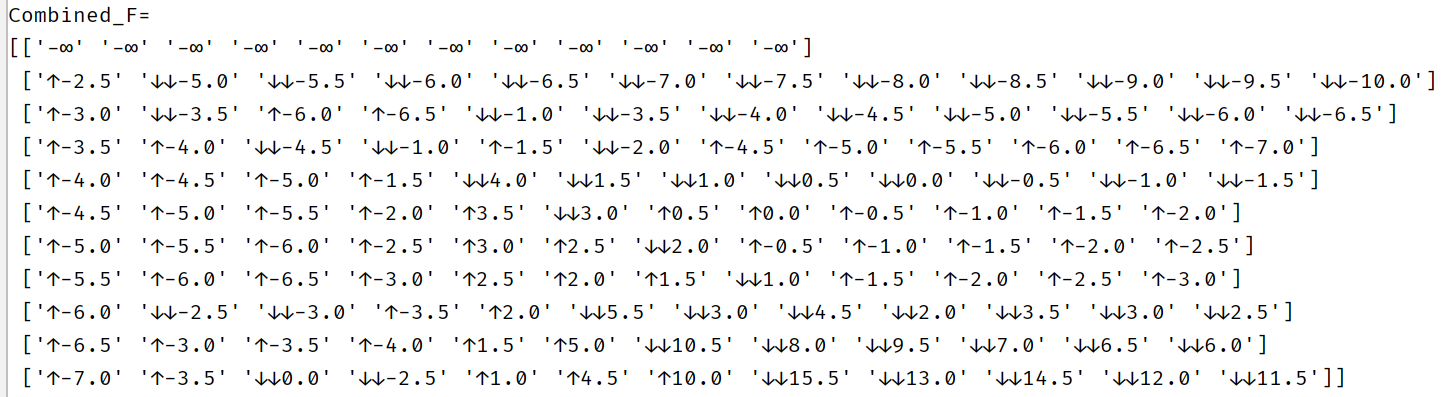
## Affine Gap Alignment: Gotoh’s Algorithm

Experiment: S1=TTACTGTGTTT and S2=CACCCCTGTG (same as the homework question)

Alpha = 2, Beta = 0.5, Match Cost = 5, Mismatch Cost = -1, Running Time: 0.0032 seconds

Alignment Score=19.5, ST1: TA---CTGTGTTT, ST2: CACCCCTGTG---





## Multiple Sequence Alignment: Center Star

S1 = CCTGCTGCAG

S2 = GATGTGCCG

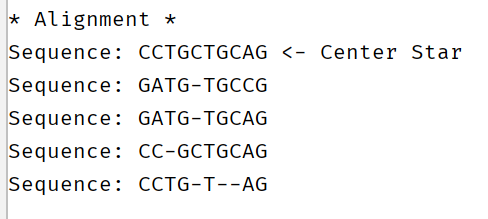
S3 = GATGTGCAG

S4 = CCGCTAGCAG

S5 = CCTGTAGG

(same as in the center star explanation in the MultipleSequenceAlignment PDF)





## Neighbor Joining

## 

## Basic Local Alignment Search Tool: BLAST

Sample Matrix provided in attachment for all runs.

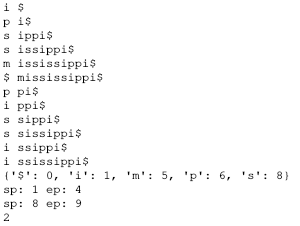
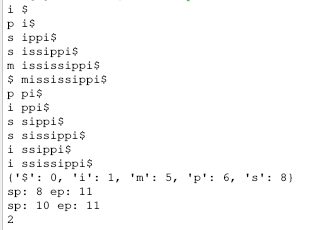
Input: AA threshold: 7 Input: BB threshold: 7



|  |  |  |
| --- | --- | --- |
| Size of Matrix (nxn) | Size of kmer | Run time (seconds) |
| 6 | 4 | 7.4620 |
| 6 | 5 | 8.7707 |
| 6 | 6 | 12.5106 |
| 6 | 7 | 48.9699 |
| 6 | 8 | 508.0497 |

* Burrows-Wheeler Transform Counting Occurences

Test Case “Mississippi” find “ss” find “si”



|  |  |
| --- | --- |
| String | Run time (seconds) |
| Mississippi | 0.0009 |
| MississippiMississippi | 0.0014 |
| MississippiMississippiMississippi | 0.0014 |

# Conclusion

In this project we presented some foundational algorithms of computational biology covering the topics of local alignment, global alignment, affine gap alignment, multiple sequence alignment, the Basic Alignment Search Tool, and the Burrows-Wheeler transformation. We demonstrate we implemented these algorithms in Python through sample input sets and experiments in the result section, as well as explain some of our design choices in the methodology section. We also provide the code used to generate all the results as an attachment to this report.

The major challenge I encountered while implementing the alignment algorithms was the fact that it’s very easy to confuse the order of indices for rows and columns. There were many instances where the algorithm output was incorrect due to the fact that the order of the indices was reversed in the dynamic programming recurrence relation code. When implementing center star this issue was even bigger, as there were indices for rows and columns, for individual characters, and indices for elements of lists, all mixed together. Overall, I did not feel like it was super challenging, and I learned some neat features of Python while working on this project such as f-strings. I did my best to structure my code into methods to avoid code duplication, and also attempted to follow PEP8 standards when writing my code. There is one thing I do not believe I implemented correctly, which is the “D” score used in Center Star to automatically select the best center star. However, apart from that, the methods which I implemented seem to work as intended as I tested them on examples we had in class from the homework and midterm exam (for which the solutions are provided). Regrettably, it seems we were unable to finish the neighbor joining algorithm implementation on time to put it on the report as well.

BLAST Nearest Neighbors was difficult to implement because the size of the string and the amount of characters you had to check was unknown which means you don’t know how many times you have to traverse the matrix. To overcome this problem I made a matrix of coordinates then stored all the possibilities then proceeded to traverse all the possible replacements recursively. The debugging was also difficult because of the amount of time it takes to run and how complex the recursive method I created is. To overcome the debugging i did unit testing on each method before integrating them together to make sure it was working. In the end it is completed and runs with O(k2n) which can be seen in the results section.

Burrows-Wheeler Transform Counting Occurrences was not as difficult to implement as BLAST Nearest Neighbors the most difficult part was implementing an algorithm to create the suffix array. Afterwards making the BWT and the counting occurrences algorithm seen in class became easier. The run time is constant and that can be seen in the results section.

# Individual Contributions

Sebastian: BLAST Nearest Neighbors, BWT Counting Occurences, Neighbor Joining

Jose: Needleman-Wunsch, Smith-Waterman, Gotoh, Center Star

For the report, both of us put in the same amount of work for the results and methodology section. Sebastian put more work than Jose in the introduction, and Jose put more work in the conclusion.