Carleton University
Department of Systems and Computer Engineering
SYSC 2004 - Object Oriented Software Development

# DNA ALIGNMENT PROJECT

April 9, 2017

Team Member: Denis Shleifman: 101001778
Team Member: Joseph Hyland: 101007225

Professor: Moayad Aloqaily
TA: Justin Sigouin
TA: Sasha Gunaratne

# Contents

# List of Tables

# 1 Problem Identification and Statement

The problem being addressed by this report is the alignment of two sequences of DNA. This can be useful in comparing the DNA of different species by determining how much they are related. It can also be used to compare the DNA of different humans in determining the different properties that certain parts of DNA control.

This report presents a method of assigning a numerical value to the similarity between strands. As well, it presents the design of a program that accepts user inputs for different strands of DNA and calculates a score for the similarity of the strands.

Another issue to be address is the fact that different situations may require different methods of comparison. Some applications may require different systems of scoring for the calculation of similarity. One application may allow for the possibility of gaps between nucleotides where another may need to indicate that the chance of gaps is very unlikely. The latter case would need to be able to assign a significant penalty to any alignments with gaps in the sequence while the former might want to find alignments that include gaps within the sequence.

The proposed solution to these problems is to write a program capable to accepting user information about strands of DNA along with the desired scoring method and producing an objective scope to reflect the similarity of the strands.

# 2 Gathering of Information and Input/Output Description

This section contains definitions for important words in section 2.1, a description of the method of scoring in section 2.2 and a description of the algorithm that will be used to solve this problem in section 2.3. Section 2.4 contains the a description of the objectives of this laboratory and section 2.5 contains a description of the user interface of the program.

## 2.1 Definitions

This section contains definitions and descriptions of words that will be used throughout the report. Section 2.1.1 contains a definition for DNA and section 2.1.2 contains a definitions for the alignment process.

### 2.1.1 The DNA Molecule

DNA is a molecule that encodes the information required for making the various proteins required for life. Strands of DNA are made of nucleotides that encode the information of the molecule. DNA is composed of four different kinds of nucleotide base pairs A, T, C, and G effectively creating a base 4 number system. The molecule is constructed as a double helix with two opposite strands bonded together. One side of the DNA molecule bonds with the other by A bonding with T and C bonding with G. A piece of DNA can be seen with its reverse strand as:

Forward strand: ACGTGCTAGCTAG

Reverse strand: TGCACGATCGATC

The process of mirroring the strand of DNA allows for another molecule RNA to copy the information encoded in the DNA sequence and pass it along to other entities within the cell that can use the information in the RNA to fabricate amino acids for assembly into proteins.

### 2.1.2 The Alignment Process

The alignment of two strands of DNA refers to the process which is performed to align certain subsequences of a strand of DNA with another strand of DNA.

## 2.2 Method of Scoring

In order for the program to assign a score to a given alignment of DNA it requires a method of scoring. This report uses 3 different scoring parameters that must be assigned to complete the calculation. The parameters that are required are match score, gap penalty, and mismatch penalty. The match score is the amount of points rewarded for two characters of the DNA strands aligning. The gap penalty is the score drop for every character aligned with a gap and the mismatch penalty is the score drop for every charterer aligned with a different character in the other strand of DNA.Based on these penalties an algorithm can be used to find the alignment to give the best score. Depending on the scoring system, the alignment produced by the algorithm should be different.

## 2.3 Needleman-Wunsch Algorithm

In order for the program to be able to generate a score to represent the quality of the alignment of the two strands it requires an algorithm to follow. The algorithm that this report will

implement to accomplish this task is known as the Needleman-Wunsch algorithm which uses a matrix of values to score each section of the sequence alignment. The matrix for the scoring system is setup as shown in Table 1 assuming penalties of 0 for both gap and misalignment and a score of 1 for a match. The zeros down the side and across the top of the matrix are due to the gap penalty. For gap penalties that are non-zero the values would be initialized to $n \times gap_{penalty}$ where $n$ is the number of rows or columns from the top left point. This is due to the fact that moving across the top and left edges of the matrix the value is going to come exclusively from the previous cell plus the gap penalty.

**Table 1:** Setup matrix for calculating the score and alignment of two sequences of DNA

| | | G | A | A | T | T | C | A | G | T | T | A |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| G | 0 | | | | | | | | | | | |
| G | 0 | | | | | | | | | | | |
| A | 0 | | | | | | | | | | | |
| T | 0 | | | | | | | | | | | |
| C | 0 | | | | | | | | | | | |
| G | 0 | | | | | | | | | | | |
| A | 0 | | | | | | | | | | | |

The values in the table are entered based on the set scores for matches, mismatches, and gaps. If the value for matching is made positive the penalties for gaps and mismatches should be made zero or negative so that the maximum score can be found to reflect the best alignment. The match score should not be set below zero otherwise the program will not calculate the correct traceback and find the desired alignment. The values in the matrix are filled in based on the maximum score from the preceding cells. An example of two strands being scored in the matrix is shown in Table 2 where the penalties are zero and the match score is one.

**Table 2:** Matrix for calculating the score and alignment of two sequences of DNA with the score values entered

| | | G | A | A | T | T | C | A | G | T | T | A |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| G | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| G | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 2 | 2 | 2 | 2 |
| A | 0 | 1 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 3 |
| T | 0 | 1 | 2 | 2 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 |
| C | 0 | 1 | 2 | 2 | 3 | 3 | 4 | 4 | 4 | 4 | 4 | 4 |
| G | 0 | 1 | 2 | 2 | 3 | 3 | 4 | 4 | 5 | 5 | 5 | 5 |
| A | 0 | 1 | 2 | 3 | 3 | 3 | 4 | 5 | 5 | 5 | 5 | 6 |

The values seen filled in Table 2 have been obtained from using the algorithm

$$F(i,j) = MAX\{F(i-1,j-1)+S(seq_1 i, seq_2 j), F(i-1,j)+gap_{penalty}, F(i,j-1)+gap_{penalty}\}$$

to fill in the table from the top left to the bottom right. The algorithm fills in the values for the table but at the same time the direction from which the values were obtained must be remembered so that the direction for traceback can be remembered. An additional step that can be taken is to remember all the places that the number could have been generated from so that all possible alignments can be obtained in the traceback process. Table 3 shows the traceback of one possible path shown in red from the maximum value back through the matrix.

**Table 3:** Traceback process for calculating the score and alignment of the two DNA sequences with the traceback path shown in red.

| | | G | A | A | T | T | C | A | G | T | T | A |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| G | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| G | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 2 | 2 | 2 | 2 |
| A | 0 | 1 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 3 |
| T | 0 | 1 | 2 | 2 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 |
| C | 0 | 1 | 2 | 2 | 3 | 3 | 4 | 4 | 4 | 4 | 4 | 4 |
| G | 0 | 1 | 2 | 2 | 3 | 3 | 4 | 4 | 5 | 5 | 5 | 5 |
| A | 0 | 1 | 2 | 3 | 3 | 3 | 4 | 5 | 5 | 5 | 5 | 6 |

Using the traceback in Table 3 the alignment can be found as

$$G\ A\ A\ T\ T\ C\ A\ G\ T\ T\ A$$

$$G\ G\ A\ \_\ T\ C\ \_\ G\ \_\ \_\ A$$

where the _ values represent gaps in the alignment.

## 2.4 Objectives

The objectives of this project are to create a program to implement an algorithm capable of accepting accepting user inputs for penalties and rewards and finding an optimal alignment for two input strands of DNA. The program will be able to accept the penalties and reward as inputs from the user along two strands of DNA and find the best way to align the two strands based on the scores assigned.

## 2.5 User Interface

This section contains a description of the program inputs in section 2.5.1 and the outputs in section 2.5.2.

### 2.5.1 Input

The input from the user will be either double or int values for the gap and misalignment penalties as well as the match reward. The DNA values will be input by the user as strings. When the program starts it will request inputs from the user in a form such as:

**Input 2 valid sequences of DNA and the desired scores and penalties:**
**Input Sequence 1 (String of A,T,C and G):**
**Input Sequence 2 (String of A,T,C and G):**
**Input Scoring Parameters under the format:**
**(MatchScore, Mismatch Penalty, Gap Penalty)**
**Example: "(5.03,22,10.3)"**
**Parameters:**

### 2.5.2 Output

The program will then generate a user interactive menu similar to the one shown below.

**Choose an option:**

**1) Print both sequences**
**2) Print initialized Needleman-Wunsch matrix**
**3) Print results of the Needleman-Wunsch matrix (entire matrix)**
**4) Print one optimal alignment**
**5) Print all optimal alignments**
**6) Enter two new sequences of DNA to score**
**7) Exit program**

The program will be able to loop through an repeat the first step of accepting strings of DNA and calculating their alignment based on the initial scores that were entered as many times as the user desires until the exit option is selected.

# 3 Design

The program is divided into 5 main classes: the DNA class, the NeedleManWunsch class, the Parameters class, the Alignment class and the Main class. These classes as well as their methods are described below. Figure 1 below demonstrates the layout of all the classes.

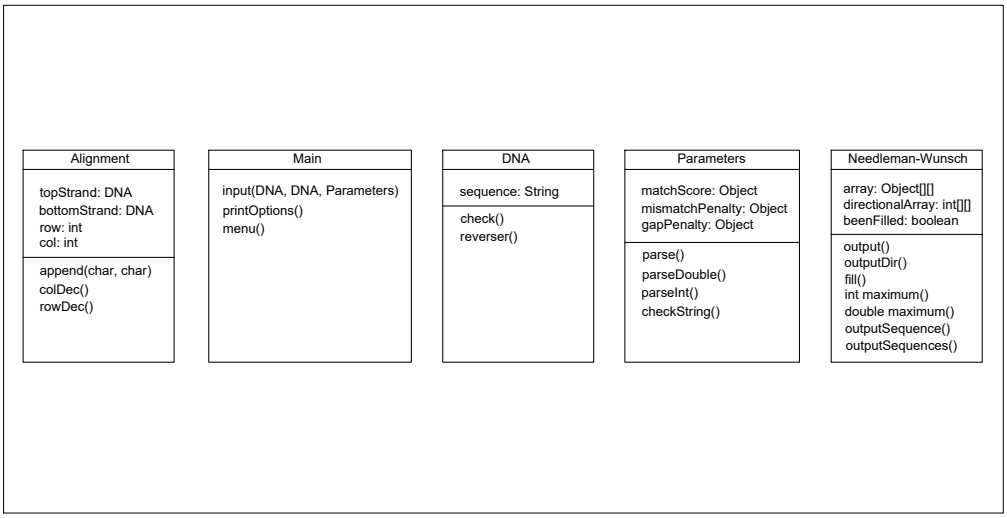


**Figure 1:** Project layout

### 3.0.1 DNA Class

**3.0.1.1 Fields** This class contains one private string field called sequence which stores the DNA sequence.

**3.0.1.2 Constructors** This class contains a parametrised constructor which initializes the sequence field. It also contains a default constructor which initializes the sequence to a null pointer.

**3.0.1.3 Getters and Setters** This class contains getters and setters for the sequence field.

**3.0.1.4 public String toString()** This method returns the string stored in sequence.

**3.0.1.5 public boolean equals(Object obj)** This method does a comparison between two objects by checking that they are both of the class DNA and both contain the same data stored

in their fields. This method returns true if both objects are the same and false if both objects are not.

**3.0.1.6 public boolean check()** This method parses through the string and ensures that there are not illegal characters. Any illegal characters (any character other than A,C,T or G) is discarded and any lowercase letters are changed to their respective uppercase letters. This method returns true if the string contains at least one legal character (A, C, T or G) and false if the string did not contain any.

**3.0.1.7 public String reverser()** This method returns a string where the characters in the sequence have been reversed. Meaning that the characters start from the last character in the sequence, and go to the first.

### 3.0.2 NeedleManWunsch Class

The needleManWunsch class contains all the information and methods required to compute the optimal alignments between the strands of DNA based on the Parameters set by the user.

**3.0.2.1 Fields** The three fields required by the needleManWunsch class include an "Object[][] array", an "int[][] directionalArray" and a "boolean beenFilled". The roll of the array is to store the scores from using the Needleman-Wunsch algorithm to fill the matrix. The array is left as an Object so that it can hold either integers or doubles depending on what the user inputs for the parameter scores and penalties. The directionalMatrix is designed to remember where each value in the array matrix came from. It can be used as an integer matrix because the program can use only integers to indicate the direction from which the maximum values were generated and not have to worry about doubles. The boolean beenFilled is need in the class to remember if the fill method has been called so that it does not get called more than once.

**3.0.2.2 Constructors** The needleManWunsch class contains constructors to accept dimensions for the two matrices as well as a default constructor that just sets the matrices to null. For both of the constructors the boolean value is set to false.

**3.0.2.3 Getters and Setters** The needleManWunsch class has getters and setters for both of the matrices returning a handle to the matrix in each instance.

**3.0.2.4 Methods** The methods for the Needleman-Wunsch class include an output method, an outputDir method, a fill method, an overloaded maximum method, an outputSequence method and an outputSequences method.

$$output(DNA\ A, DNA\ B)$$

**Parameters:**

A and B are the DNA sequences input by the user passed to the function so that they can be printed across the top and side of the matrix

**Return type:**

the function is a void return type but it outputs the initialized Needleman-Wunsch matrix

$$outputDir(DNA\ A, DNA\ B)$$

**Parameters:**

A and B are the DNA sequences input by the user passed to the function so that they can be printed across the top and side of the matrix

**Return type:**

the function is a void return type but it outputs the initialized directional matrix filled to keep track of where the values in the Needleman-Wunsch matrix were generated from

$$fill(DNA\ A, DNA\ B, Parametersscore)$$

**Parameters:**

A and B are the DNA sequences input by the user passed to the function so that they can be printed across the top and side of the matrix and so that their intersections can be compared for matches

**Return type:**

the function is a void return type but it generates the initialized Needleman-Wunsch matrix

$$maximum(int\ a, int\ b, int\ c)$$

**Parameters:**

a, b, and c are the input values for the function to find the maximum of

**Return type:**

the function return the maximum value of the three inputs as an integer

$$maximum(double\ a, double\ b, double\ c)$$

**Parameters:**

a, b, and c are the input values for the function to find the maximum of

**Return type:**

the function return the maximum value of the three inputs as a double

$$outputSequence(DNA\ A, DNA\ B)$$

**Parameters:**

A and B are the DNA sequences input by the user passed to the function so that they can be referenced to determine which letters to append to the end of the alignment throughout the traceback process

**Return type:**

the function is a void return type but it outputs one of the optimal alignments found for the two sequences of DNA

$$outputSequences(DNA\ A, DNA\ B)$$

**Parameters:**

A and B are the DNA sequences input by the user passed to the function so that they can be referenced to determine which letters to append to the end of the alignment throughout the traceback process

**Return type:**

the function is a void return type but it outputs all the optimal alignments found for the two sequences of DNA

**3.0.2.5 equals(Object obj)** This method does a comparison between two objects by checking that they are both of the class NeedleManWunsh and both contain the same data stored in their fields. This method returns true if both objects are the same and false if both objects are not.

### 3.0.3 Parameters Class

**3.0.3.1 Fields** This class contains 3 private Object fields. One for match score, one for mistmatch penalty and one for gap penalty. The reason that these fields are Object fields is so that they can either store integers or doubles.

**3.0.3.2 Constructors** This class contains a parametrised two parametrised constructors. One constructor initializes all the fields with doubles and the other with integers. The class also contains a default constructor which intitializes each field with the integer value 1.

**3.0.3.3 Getters and Setters** This class contains 1 getter for every field that returns an object and 2 setters for every field. One setter allows the field to be set as an integer, the other allows the field to be set as a double.

**3.0.3.4 public String toString()** The toString method returns a string of all the parameters in the format: "(Match Score, Mismatch Penalty, Gap Penalty) = (3.2,5,1.9)"

**3.0.3.5 public boolean equals(Object obj)** This method does a comparison between two objects by checking that they are both of the class Parameters and both contain the same data stored in their fields. This method returns true if both objects are the same and false if both objects are not.

**3.0.3.6 public static boolean checkString(String string)** This method is used to iterate through a string to ensure that it contains the proper control character to be parsed. The control character needed for parsing are: "( 0 , - 0 , - 0 )". Where the "0" character represents the minimum requirement of 1 numeric character. This method will iterate through the string determine that these characters are in the string and are in the right order. If this condition is satisfied the method will return true, otherwise the method returns false.

**3.0.3.7 public static String cleanString(String string)** This method is used to iterate through a string to ensure and remove all superflous characters rendering it to its simplist form. This form is: "(0,-0,-0)" Where the '0' character represents a number that is either a double or integer.

**3.0.3.8 public static boolean checkDouble(String string)** This method is used to check to see if any of the values in the string are doubles. If at least one is (by finding a "." character) is a double, the method returns true. If no values are doubles, the method returns false.

**3.0.3.9 public boolean parse(String string)** This method is used to parse through a string (string) and set every parameter in the class. This method will first check the string to ensure it contains all control characters, then it will clean the string by removing all superflous characters, then the method will check to see if all numbers are integers. If all the values are integers, the method will parse each one as an integer, if not, each value will be parsed as a double.

**3.0.3.10 public static int parseInt(String string)** This method is used to parse through an integer. The method will iterate through the string, add up the values (multiplying by ten after each subsquent numerical character) (discarding any non-numerical characters) and return the integer result.

**3.0.3.11 public static double parseDouble(String string)** This method is used to parse through an double. The method will iterate through the string, add up the values (multiplying by ten after each subsquent numerical character) (discarding any non-numerical characters) until the first "." character. Then the method will treat any subsquent numerical character as a decimal value. If a string is passed into this method containing no "." character, the method will still be able to return the value.

### 3.0.4 Alignment Class

The Alignment class is required by the program to store each of the alignments found to be optimal based on the process of the algorithm and the parameters set by the user.

**3.0.4.1 Fields** The fields of the Alignment class include "DNA topStrand", "DNA bottomStrand", "int col", and "int row". The topStrand is required to store the value of the top strand of the alignment and bottomStrand is required to store the bottom strand of the alignment. Together the two strands for the optimal alignment sequence pair. The col and row values are required to store the values of the row and column that the alignment was at when branching off to find all the possible alignments.

**3.0.4.2 Constructors** The class has three constructors with a default constructor, one to take in two sequence of DNA, and one to take in two sequences of DNA along with two integers for the row and column values.

**3.0.4.3 Getters and Setters** The getters and setters for this class were implemented for each of the fields described above. The getters and setters for the strands in particular are necessary to be able to access the DNA strands of the alignment.

**3.0.4.4 Methods** One of the methods in Alignment is the append function which takes in two characters and appends them to the end of the DNA sequences of the alignment.

$$append(char\ a, char\ b)$$

**Parameters:**

char a is the value to be appended to the top DNA strand of the alignment
char b is the value to be appended to the bottom DNA strand of the alignment

**Return type:**

The append function is a void return type One of the other methods in Alignment is the colDec function which takes in no arguments but decrements the value of col for the alignment.

$$colDec()$$

colDec has no parameters and no return type The lase method in Alignment is the rowDec function which takes in no arguments but decrements the value of row for the alignment.

$$rowDec()$$

rowDec has no parameters and no return type

**3.0.4.5 equals(Object obj)** This method does a comparison between two objects by checking that they are both of the class Alignment and both contain the same data stored in their fields. This method returns true if both objects are the same and false if both objects are not.

### 3.0.5 Main Class

**3.0.5.1 public static int parse(String string)** This method parses through the string containing the option the user chose. The parser searchers for a numerical character between

1 and 7. If such a character is found, the parser returns the integer value, if not the parser returns 0.

**3.0.5.2 public static void input (DNA A, DNA B, Parameters score)** This method uses the scanner class to obtain input strings from the user and manipulates the strings using methods in the DNA class and the Parameters class to set all the fields in A, B and score.

**3.0.5.3 public static void printOptions()** This method outputs to the console the 7 options the user has available to them.

**3.0.5.4 public static int menu()** This method uses the scanner class to obtain a string from the user containing the option they wish to chose. The method then uses the parse method to determine the option the user wishes to chose and returns this value. A returned value of 0 represents an invalid choice.

**3.0.5.5 public static void main(String [] args)** This method runs the program. This method conains the initialization of each variable and every method is called from this method.

## 3.1 Test Cases

### 3.1.1 Test Case 1

Sequence 1: GAATT, sequence 2: GAATT, match score = 1, gap penalty = mismatch penalty = 0

| | | G | A | A | T | T |
|---|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 | 0 |
| G | 0 | 1 | 1 | 1 | 1 | 1 |
| A | 0 | 1 | 2 | 2 | 2 | 2 |
| A | 0 | 1 | 2 | 3 | 3 | 3 |
| T | 0 | 1 | 2 | 3 | 4 | 4 |
| T | 0 | 1 | 2 | 3 | 4 | 5 |

Alignment sequence output from given inputs:

$$G\ A\ A\ T\ T$$

$$G\ A\ A\ T\ T$$

### 3.1.2 Test Case 2

Sequence 1: GAATT, sequence 2: GAAT, match score = 1, gap penalty = mismatch penalty = -1

| | | G | A | A | T | T |
|---|---|---|---|---|---|---|
| | 0 | -1 | -2 | -3 | -4 | -5 |
| G | -1 | 1 | 0 | -1 | -2 | -3 |
| A | -2 | 0 | 2 | 1 | 0 | -1 |
| A | -3 | -1 | 1 | 3 | 2 | 1 |
| T | -4 | -2 | 0 | 2 | 4 | 3 |

Alignment sequence output from given inputs:

$$G\ A\ A\ T\ T$$

$$G\ A\ A\ \_\ T$$

### 3.1.3 Test Case 3

Sequence 1: TTAGCT, sequence 2: TAGG, match score = 1, gap penalty = -1, mismatch penalty = -2

| | | T | T | A | G | C | T |
|---|---|---|---|---|---|---|---|
| | 0 | -1 | -2 | -3 | -4 | -5 | -6 |
| T | -1 | 1 | 0 | -1 | -2 | -3 | -4 |
| A | -2 | 0 | -1 | 1 | 0 | -1 | -2 |
| G | -3 | -1 | -2 | 0 | 2 | 1 | 0 |
| G | -4 | -2 | -3 | -1 | 1 | 0 | -1 |

Alignment sequence output from given inputs:

$$T\ T\ A\ G\ C\ T$$

$$T\ \_\ A\ G\ \_\ T$$

### 3.1.4 Test Case 4

Sequence 1: ATCGAT, sequence 2: TAGCTA, match score = 1, gap penalty = 0, mismatch penalty = 0

| | | T | T | A | G | C | T |
|---|---|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| T | 0 | 0 | 1 | 1 | 1 | 1 | 1 |
| A | 0 | 1 | 1 | 1 | 1 | 2 | 2 |
| G | 0 | 1 | 1 | 1 | 2 | 2 | 2 |
| C | 0 | 1 | 1 | 2 | 2 | 2 | 2 |
| T | 0 | 1 | 2 | 2 | 2 | 2 | 3 |
| A | 0 | 1 | 2 | 2 | 2 | 3 | 3 |

Alignment sequence output from given inputs:

$$T \ \_ \ T \ A \ G \ C \ T$$

$$T \ A \ G \ C \ T \ A \ \_$$

### 3.1.5 Test Case 5

Sequence 1: GATC, sequence 2: TAGCTA, match score = 0, gap penalty = -1, mismatch penalty = -1

| | | G | A | T | C |
|---|---|---|---|---|---|
| | 0 | -1 | -2 | -3 | -4 |
| T | -1 | -1 | -2 | -2 | -3 |
| A | -2 | -2 | -1 | -2 | -3 |
| G | -3 | -2 | -2 | -2 | -3 |
| C | -4 | -3 | -3 | -3 | -2 |
| T | -5 | -4 | -4 | -3 | -3 |
| A | -6 | -5 | -4 | -4 | -4 |

Alignment sequence output from given inputs:

$$G \ A \ \_ \ \_ \ T \ C$$

$$T \ A \ G \ C \ T \ A$$

# 4 Implementation

Please see the JAVA program in the attached ZIP folder.

# 5 Software Testing and Verification

### 5.0.1 Test Case 1

Sequence 1: GAATT, sequence 2: GAATT, match score = 1, gap penalty = mismatch penalty = 0

```
Choose an Option:
        1) Print Both Sequences
        2) Print Initialized Needleman-Wunsch Matrix
        3) Print Results of Needleman-Wunsch Matrix (Entire Matrix)
        4) Print One Optimal Alignment
        5) Print All Optimal Alignments
        6) Enter Two New Sequences of DNA to Score
        7) Exit Program
        Please Choose an Option (1 to 7): 5

Option 5: Print All Optimal Alignments
GAATT
GAATT
```

**Figure 2:** All optimal alignments from the first test case

### 5.0.2 Test Case 2

Sequence 1: GAATT, sequence 2: GAAT, match score = 1, gap penalty = mismatch penalty = -1

```
Choose an Option:
        1) Print Both Sequences
        2) Print Initialized Needleman-Wunsch Matrix
        3) Print Results of Needleman-Wunsch Matrix (Entire Matrix)
        4) Print One Optimal Alignment
        5) Print All Optimal Alignments
        6) Enter Two New Sequences of DNA to Score
        7) Exit Program
        Please Choose an Option (1 to 7): 5

Option 5: Print All Optimal Alignments
GAATT
GAA_T

GAATT
GAAT_
```

**Figure 3:** All optimal alignments from the second test case

### 5.0.3 Test Case 3

Sequence 1: TTAGCT, sequence 2: TAGG, match score = 1, gap penalty = -1, mismatch penalty = -2

```
Choose an Option:
        1) Print Both Sequences
        2) Print Initialized Needleman-Wunsch Matrix
        3) Print Results of Needleman-Wunsch Matrix (Entire Matrix)
        4) Print One Optimal Alignment
        5) Print All Optimal Alignments
        6) Enter Two New Sequences of DNA to Score
        7) Exit Program
        Please Choose an Option (1 to 7): 5

Option 5: Print All Optimal Alignments
TAGCT
TAG_G

TTAGCT
T_AG_G

TAGCT_
TAG__G

TTAGCT_
T_AG__G

TAGCT
TAGG_

TTAGCT
T_AGG_

TAGC_T
TAG_G_

TTAGC_T
T_AG_G_

TA_GCT
TAGG__

TTA_GCT
T_AGG__

TAG_CT
TAGG__

TTAG_CT
T_AGG__
```

**Figure 4:** All optimal alignments from the third test case

### 5.0.4 Test Case 4

Sequence 1: ATCGAT, sequence 2: TAGCTA, match score = 1, gap penalty = 0, mismatch penalty = 0

```
Choose an Option:
        1) Print Both Sequences
        2) Print Initialized Needleman-Wunsch Matrix
        3) Print Results of Needleman-Wunsch Matrix (Entire Matrix)
        4) Print One Optimal Alignment
        5) Print All Optimal Alignments
        6) Enter Two New Sequences of DNA to Score
        7) Exit Program
        Please Choose an Option (1 to 7): 5

Option 5: Print All Optimal Alignments
ATCGAT
AGCTA_

AT_CGAT
A_GCTA_

T__CGAT
TAGCTA_

A_TCGAT
AG_CTA_

ATCG_AT
AGC_TA_

AT_CG_AT
A_GC_TA_

T__CG_AT
TAGC_TA_

A_TCG_AT
AG_C_TA_

TCG__AT
TAGCTA_

TC_G__AT
T_AGCTA_
```

**Figure 5:** A sample of the optimal alignments from the fourth test case. Run the program to see all alignments.

#### 5.0.5 Test Case 5

Sequence 1: GATC, sequence 2: TAGCTA, match score = 0, gap penalty = -1, mismatch penalty = -1

```
Choose an Option:
        1) Print Both Sequences
        2) Print Initialized Needleman-Wunsch Matrix
        3) Print Results of Needleman-Wunsch Matrix (Entire Matrix)
        4) Print One Optimal Alignment
        5) Print All Optimal Alignments
        6) Enter Two New Sequences of DNA to Score
        7) Exit Program
        Please Choose an Option (1 to 7): 5

Option 5: Print All Optimal Alignments
GATC
GCTA

GA__TC
TAGCTA

GATC__
TAGCTA
```

**Figure 6:** All optimal alignments from the fifth test case

## 5.1 Observations

From the observed test cases it can be seen that the number of optimal alignments increases as the length of the DNA sequences increases. The number also increases at times when the gap penalty is zero making the disincentive for adding gaps to the alignment disappear.

Setting the penalties to zero also seemed to decrease the quality of our alignments as it allowed many more gaps to be introduced into the alignment. Increasing the penalties keeps the alignments shorter and does not create as many optimal alignments.

## 5.2 All possible alignments and program complexity

To find all optimal alignments we used a list to store the alignments and appended to list when we got to points in the traceback where our Needleman-Wunsch values could have came from multiple places. After completing an alignment we took the next alignment off the end of the list and completed it; repeating the process until the list was empty.

We stored the direction from which each value came by using a directional matrix to store a unique value for each of the directions. When filling the Needleman-Wunsch matrix we added 1 to the directional matrix if the value came from the top, two if it came from the

side, and four if it came from the diagonal. Then in traceback we took each of the values to correspond to a scenario as:

value = 1: From the top
value = 2: From the side
value = 3: From the top and side
value = 4: From the diagonal
value = 5: From the diagonal and top
value = 6: From the diagonal and side
value = 7: From all directions

These cases were then handles appropriately with the correct values being added to the alignments and the row and column values being adjusted appropriately.

## 5.3 Program bugs

One of the bugs that our program has is that the penalty parameters must be added with a negative sign as -0. This is not a major bug and could likely be fixed given time to do so.

## 5.4 Team member roles

Both team members shared the programming and report writing roles throughout the project. Denis was more focused on the user interface part of the program while Joseph was in charge of getting the traceback process to work properly.