

School of Biological Sciences



Bioinformatics Algorithms

ICA Report

Author: B236494

Tuesday 2nd April, 2024

Introduction

The Smith-Waterman algorithm is used to determine regions of sequence similarity when comparing protein sequences or nucleic acid sequences by performing a local alignment. A version of this algorithm has been coded into a Python script and provided. Instructions were given to add functionality to this code, keeping the EMBOSS suite's water program as the "gold-standard".

This report will be focused on the changes made in the SmithWaterman.py (Version 1.9 SRT) file that was provided to improve functionality and robustness. The code has been modified with the implementation of 4 key elements:

1. Allows the parsing of certain parameters that will be used to generate the alignment.
2. Checking input file is it is suitable for alignment.
3. Inclusion of print statements at multiple points in the code.
4. A test at the end which compares the output of the script to a "gold-standard".

The changes that were made to achieve these implementations will be described along with how it has aided in assessing if the code is being executed appropriately.

Please Note:

- The user is able to choose the pause (in seconds) between important steps to read what is being executed or input 0 and skip all pauses.
- The modified script file (B236494_SmithWaterman_Final_Script.py) has been provided in the Supplementary Material.
- A pdf has been appended with this report (Modifications_Made_To_SmithWaterman.py_V1.9.pdf) which shows the changes made in the program and a clear comparison can be seen with the base code that was provided. All line numbers mentioned moving forward will be quoted from the same.
- The -datafile parameter used (EDNAFULL_srt) in the EMBOSS water program execution was provided with the base script file.

Parameter Parsing for Alignment Generation

This was done using the argparse module. It enables the user to provide parameters for the match/mis-match/gap scores on the CLI (command line interface) to be used during the alignment. Default values have also been assigned so that the user does not have to provide the parameters if they do not wish to do so. These values will also be passed to EMBOSS water at the end so that meaningful comparison can be done between the two

This functionality has been implemented in lines 335 - 349 of the modified script file.

Verifying Input File Suitability for Alignment

Input files were vetted to ensure that only A/T/G/C characters were present in the files chosen for selection. Note that the user can provide the filename, provided it is in the current working directory, and the programme reads the filenames, stores the sequences and then checks for non-ATGC characters. Execution proceeds as normal if there are none or the script is terminated and the user is told to provide the right sequence files.

This functionality has been implemented in lines 361 - 422 of the modified script file.

Integrating Print Statements Throughout the Code

The script was developed using many print statements to gauge the output at various stages of execution. Most of these have been commented out as there is no benefit to the user in knowing these results. The informative print statements have been displayed with an optional pause functionality present before each, if the user wishes to use it.

Some examples of this functionality can be seen in lines 62, 70, 215 and 282.

Final Test Comparing Script Output to Gold Standard

It is good practice when testing a variation of a particular algorithm/function to have a “gold-standard” for comparison. As mentioned earlier, EMBOSS’s water is being used for that purpose here. The output of the script has been inspired from that of water and the output of water is used to validate that the script is working properly. After the script has given us the alignment that it has calculated, water is run using `os.system()` step and displayed to the user. The output of both is quite similar and this tells us that the script is working properly.

This functionality has been implemented in lines 445 - 460 of the modified script file.

Conclusion

All these implementations ensure that the code is being executed in the expected manner and also give the user room for flexibility in both altering the parameters and in reusing the code conveniently to include other functionalities or grasp the flow of execution. This script can be improved by additional test functions for comparison, not printing out certain outputs or including additional print statements and further error-trapping the inputs that it takes. The changes that have been made in the code can be visualised clearly in the next few pages.

Please view the supplementary Material for a full picture of the modifications made. In particular, do note the comparison view mode (after the code) as mentioned earlier.

Supplementary Material

The modified script file adapted from the base script file: B236494_SmithWaterman_Final_Script.py

```
#!/usr/bin/python3

### Bioinformatics Algorithms ICA Script by B236494 ###

### Adapted from BA4 class code written by Simon Tomlinson ###

### print statements were used throughout execution to test the outputs of the program ###

### A test has also been implemented at the end to compare the output witht the EMBOSS water Gold-standard ###

### B236494_SmithWaterman_Final_Script.py script adapted from SmithWaterman.py Version 1.9 SRT ###

# Simon Tomlinson Bioinformatics Algorithms
# Perform Smith Waterman Alignment in Python (from first principles)
# contains rows lists each of length cols initially set to 0
# index as my_matrix[1][2] my_matrix[R][C]
#
# Version 1.9 SRT

import time
import os
import sys
from time import sleep
from enum import Enum
import pandas as pd
import re
import argparse

# Global enumeration used for tracement
TypeB = Enum('TypeB', ['INSERT', 'DELETE', 'MISMATCH', 'MATCH', 'END'])

def create_matrix(rows, cols):
    my_matrix = [[0 for col in range(cols + 1)] for row in range(rows + 1)]
    return my_matrix

# x is row index, y is column index
# follows[r][c]
def calc_score(matrix, x, y):
    print("seq1:", sequence1[y - 1], " seq2: "+sequence2[x - 1], "x:", x, " y:", y)
    sc = seqmatch if sequence1[y - 1] == sequence2[x - 1] else seqmismatch
    base_score = matrix[x - 1][y - 1] + sc
    insert_score = matrix[x - 1][y] + seqgap
    delete_score = matrix[x][y - 1] + seqgap
    v = max(0, base_score, insert_score, delete_score)
    return v

# makes a single traceback step
def traceback(mymatrix, maxv):
    x = maxv[0]
    y = maxv[-1]
    val = mymatrix[x][y]

    # todo add some outputs for checking errors
    sc = seqmatch if sequence2[x - 1] == sequence1[y - 1] else seqmismatch
    # print(sc)

    base_score = mymatrix[x - 1][y - 1] + sc
    # print(base_score)
    if base_score == val:
        if sc == seqmatch:
            return [x - 1, TypeB.MATCH, y - 1]
        else:
            return [x - 1, TypeB.MISMATCH, y - 1]

    insert_score = mymatrix[x - 1][y] + seqgap
    # print(input_score)
    if insert_score == val:
        return [x - 1, TypeB.INSERT, y]
    else:
        return [x, TypeB.DELETE, y - 1]
```

```

# builds the initial scoring matrix used for traceback
def build_matrix(mymatrix):
    rows = len(mymatrix)
    cols = len(mymatrix[0])
    row_number=0

    for i in range(1, rows):
        row_number = row_number + 1
        print("\nRow Number:", row_number)
        sleep(wait)
        for j in range(1, cols):
            mymatrix[i][j] = calc_score(mymatrix, i, j)

    return mymatrix

# gets the max value from the built matrix
# Max is the end of the traceback for SW
def get_max(mymatrix):
    max = mymatrix[0][0]
    mrow = 0
    mcol = 0

    rows = len(mymatrix)
    cols = len(mymatrix[0])

    for i in range(1, rows):
        for j in range(1, cols):
            if mymatrix[i][j] > max:
                max = mymatrix[i][j]
                mrow = i
                mcol = j
    print("The Maximum Score was: ", max, "\n")
    return [mrow, TypeB.END, mcol]

# print out the best scoring path from the SW matrix
def print_matrix(mymatrix):
    rows = len(mymatrix)
    cols = len(mymatrix[0])
    s1 = " " + sequence1
    s2 = " " + sequence2

    sleep(wait)

    print("\nDimensions of The SmithWaterman Matrix: Rows= %2d , Columns= %2d\n" % (rows, cols))

    sleep(wait)

    for a in s1:
        print(a, end="")
        print(" \t", end="")
    print("\n", end="")

    for i in range(0, rows):
        print(s2[i], end="")
        print(" \t", end="")
        for j in range(0, cols):
            print("%02d\t" % (mymatrix[i][j]), end="")
        print("\n", end="")

# print out the traceback of the best scoring alignment
def print_traceback(mymatrix):
    # this will print as expected with internal gaps

    sleep(wait)

    print("\n### We Will Now Build The Traceback... ###\n")
    maxv = get_max(mymatrix)
    print(maxv)

# stash the max score for later

```

```

max_score = mymatrix[maxv[0]][maxv[-1]]

# traverse the matrix to find the traceback elements
# if more than one path just pick one
topstring = ""
midstring = ""
bottomstring = ""

# pad the sequences so indexes into the sequences match the matrix indexes
asequence1 = "#" + sequence1
asequence2 = "#" + sequence2

# this vector is used to store the traceback results

traversal_results = []

# add the rest of the elements
search = True
lastelement = False

# Stores the position so it can track if it is an insertion or deletion
# Check if it is a gap or not
if max_score < 1:
    print ("There is no suitable alignment...Check your inputs please!")
    exit;
old_maxv = maxv

while (search):

    # print(" position: %d, %d " % (maxv[0], maxv[-1]))

    # print("Testing execution, type is", current_type)

    # store the results
    traversal_results.append(maxv)

    # type_traversal = type(traversal_results)

    maxv = traceback(mymatrix, maxv)

    # catch the trivial case that we are at the end of one of the sequences
    if (maxv[-1] < 0 or maxv[0] < 0):
        traversal_results.append(maxv)
        search= False
        continue

    if (mymatrix[maxv[0]][maxv[-1]] == 0 and lastelement == False):
        lastelement = True
        continue

    if(lastelement==True) :
        search= False
        traversal_results.append(maxv)
        continue

for i in range(0, len(traversal_results)-2):

    # print("Testing execution")

    # The TypeB of the next element gives how the current element was reached
    # in the dynamic programming table
    # The current element gives the index of the two matching bases to be aligned

    curr_el=traversal_results[i]
    next_el=traversal_results[i+1]

```

```

# Match
if(next_el[1]==TypeB.MATCH):
    bottomstring += asequence2[curr_el[0]]
    topstring += asequence1[curr_el[-1]]
    midstring += "|"
    # print(" position: ", i,i )
    # print("MATCHHHHHH")

#Mismatch
elif(next_el[1]==TypeB.MISMATCH):
    bottomstring += asequence2[curr_el[0]]
    topstring += asequence1[curr_el[-1]]
    midstring += "."
    # print(" position: ", i,i )
    # print("MISMATCH/START?")

elif(next_el[1]==TypeB.INSERT):
    bottomstring += asequence2[curr_el[0]]
    topstring += "-"
    midstring += " "
    # print(" position: ", i,i )
    # print("Insertion")

elif(next_el[1]==TypeB.DELETE):
    bottomstring += "-"
    topstring += asequence1[curr_el[-1]]
    midstring += " "
    # print(" position: ", i,i )
    # print("DELETED")

print("")
for element in traversal_results:
    print(element,"\n")

sleep(wait)

print("\nFinal Alignment, Score: %d\n" % max_score)

sleep(wait)

### Printing the alignment with an effect ###

# print(topstring[::-1])
animated_print(topstring[::-1])

# print(midstring[::-1])
animated_print(midstring[::-1])

# print(bottomstring[::-1])
animated_print(bottomstring[::-1])

sleep(wait)

# print("Testing execution")

def time_to_pause():
    print("How long do you want the pauses in between important steps to be in seconds?")
    print("Please input a positive integer value <= 3 OR input 0 to execute without pauses")
    while True:
        wait_input = input("What is you choice? \n")
        if not wait_input.isdigit():
            print("TypeError: Please input a numerical value")
            continue;
        wait = int(wait_input)
        if wait < 0 or wait > 3:
            print("Invalid choice. Please choose a positive integer value <= 3")
        else :
            print("You have chosen a value of ", wait," seconds\n")
            return wait
        break;

```

```

def animated_print(s):
    for c in s:
        sys.stdout.write(c)
        sys.stdout.flush()
        time.sleep(0.25)
    print("")

# build the SW alignment...
def perform_smith_waterman():
    # values for weights
    global seqmatch
    global seqmismatch
    global seqgap
    global sequence1
    global sequence2
    global wait

    print("\n\nWelcome to B236494's version of the SmithWaterman.py Script \n")
    print("\nThis is an adapted version of SmithWaterman.py V 1.9 by Simon Tomlinson\n")
    print("\nThis script takes command line inputs for the match/mismatch/gap scores and user-input during the
        ↪ start of execution for the FASTA files to be
        ↪ aligned\n")
    print("\nEnsure that the files to be aligned are present in the current directory where the programme is
        ↪ being executed \n")
    print("\nExample CLI input could be ---> python3 SmithWaterman.py --seqmatch 1 --seqmismatch -1 --seqgap -1
        ↪ \n")

    wait = time_to_pause()

    ### Taking command line input for the match, mismatch and gap penalties ###

    parser = argparse.ArgumentParser(description='Please provide the parameters to Perform Smith-Waterman
        ↪ Alignment.')
    sleep(wait)

    # Add arguments for sequence match, mismatch, and gap penalties
    # note these defaults are not the exact weights used in the original SW paper

    parser.add_argument('--seqmatch', type=int, default=1, help='Input the score for sequence matches. Default
        ↪ is 1.')
    parser.add_argument('--seqmismatch', type=int, default=-1, help='Input the Penalty for sequence mismatches.
        ↪ Default is -1.')
    parser.add_argument('--seqgap', type=int, default=-1, help='Input the penalty for a gap. Default is -1.')

    args = parser.parse_args()

    seqmatch = args.seqmatch
    seqmismatch = args.seqmismatch
    seqgap = args.seqgap

    sleep(wait)

    print(f"Using match score: {seqmatch}")
    print(f"Using mismatch penalty: {seqmismatch}")
    print(f"Using gap penalty: {seqgap}\n")

    sleep(wait)

    ### Function to check if there are any non ATGC characters ###

    def check_if_any_non_atgc_chars(sequence):
        pattern = re.compile(r'^[ATGC]*')
        match = pattern.search(sequence.upper())
        return not bool(match)

```



```

### Function to store sequences that need to be aligned ###

def read_fasta_filename(filename):
    seq = ""
    with open(filename, 'r') as filehandle:
        for line in filehandle:
            # Using regular expression search to ignore lines starting with ">"
            if re.search("^>", line.strip()):
                continue
            seq += line.strip()
    return seq

# Taking input for file name

seq1_file = input("Enter the filename of the first sequence file: ")
seq2_file = input("Enter the filename to the second sequence file: ")

# Saving sequences to variables to be aligned

sequence1 = read_fasta_filename(seq1_file)
sequence2 = read_fasta_filename(seq2_file)

### Alternate options to test sequences ###

# sequence1="AGTGATAAACTAGTAATTTTT"
# sequence2="TTGGGGGTAAACAGGGG"

# sequence1 ="AGTCGGTTAGTAAA"
# sequence2 ="TTTTGGGTTTAGCGC"

# sequence1 = "GIGTATTTTTTT"
# sequence2 = "AAAAGTGTATT"

# sequence1 = "TCGTTCTAG"
# sequence2 = "TCGTTTTTG"

# sequence1 = "SimonTomkinson"
# sequence2 = "SimonTomlinsonBioinformaticsAlgorithms"

### Checking sequence1 & sequence2 if they contain any characters other than A, T, G, C ###

check_seq1 = check_if_any_non_atgc_chars(sequence1)
check_seq2 = check_if_any_non_atgc_chars(sequence2)

sleep(wait)

if not check_seq1 or not check_seq2:
    print("Error in One or both input sequences. There are characters other than A, T, G, and C.")
    print("Please make sure that your input files contain the correct sequences and try again.")
    print("Exiting Program...Bye")
    sleep(wait)
    sys.exit()

### Printing the sequences ###

print("The input sequences are\n")
sleep(wait)

print("Sequence1: " + sequence1)
print("Sequence2: " + sequence2)
print("\n") ### Empty line

mymatrix = create_matrix(len(sequence2), len(sequence1))
mymatrix = build_matrix(mymatrix)
print_matrix(mymatrix)

print_traceback(mymatrix)

```

```

### Taking the absolute value as EMBOSS water does not permit negative values for parameters ###

# seqmismatch = abs(seqmismatch)
seqgap = abs(seqgap)

print("\n\nComparing output with EMBOSS water")

sleep(wait)

water_command = "water -asequence {} -bsequence {} -gapopen {} -gapextend {} -outfile water_{}_{}.water -
                ↪ datafile EDNAFULL_srt".format(seq1_file,
                ↪ seq2_file, seqgap, seqgap, seq1_file, seq2_file)

os.system(water_command)

print("\n\nThis is the output from EMBOSS water using the same files which is our Gold-Standard for
                ↪ alignment")

sleep(wait)

display_water_file = "cat water_{}_{}.water".format(seq1_file, seq2_file)
os.system(display_water_file)

sleep(wait)

print("\n\nThanks for using this script.\n")

##this calls the SW algorithm when the script loads
perform_smith_waterman()

```

@@ -1,13 +1,32 @@

```

1  # Simon Tomlinson Bioinformatics Algorithms
2  # Perform Smith Waterman Alignment in Python (from first principles)
3  # contains rows lists each of length cols initially set to 0
4  # index as my_matrix[1][2] my_matrix[R][C]
5  - # Teaching code not production code!!
6  - #Version 1.9 SRT
7
8  from enum import Enum
9
10 - #Global enumeration used for tracement
11  TypeB = Enum('TypeB', ['INSERT', 'DELETE', 'MISMATCH', 'MATCH', 'END'])
12
13
14  def create_matrix(rows, cols):
15      my_matrix = [[0 for col in range(cols + 1)] for row in range(rows + 1)]
16      return my_matrix
17
18  # x is row index, y is column index
19  # follows[r][c]
20
21  def calc_score(matrix, x, y):
22      - # print("seq1:",sequence1[y- 1], " seq2: "+sequence2[x - 1], "x:",x,"
23      y:",y)
24      -
25      sc = seqmatch if sequence1[y - 1] == sequence2[x - 1] else seqmismatch
26      -
27      base_score = matrix[x - 1][y - 1] + sc
28      insert_score = matrix[x - 1][y] + seqgap
29      delete_score = matrix[x][y - 1] + seqgap
30      v = max(0, base_score, insert_score, delete_score)
31      return v
32
33  # makes a single traceback step
34  def traceback(mymatrix, maxv):
35      x = maxv[0]
36      y = maxv[-1]
37      val = mymatrix[x][y]
38
39  # todo add some outputs for checking errors
40  sc = seqmatch if sequence2[x - 1] == sequence1[y - 1] else seqmismatch
41
42  base_score = mymatrix[x - 1][y - 1] + sc
43
44  if base_score == val:
45      if sc==seqmatch:
46          return [x - 1,TypeB.MATCH, y - 1]
47      else:
48          return [x - 1,TypeB.MISMATCH, y - 1]
49
50  insert_score = mymatrix[x - 1][y] + seqgap
51
52  if insert_score == val:
53      return [x - 1, TypeB.INSERT, y]
54  else:
55      return [x, TypeB.DELETE, y - 1]
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70 + # print(input_score)
71
72  if insert_score == val:
73      return [x - 1, TypeB.INSERT, y]
74  else:
75      return [x, TypeB.DELETE, y - 1]
76
77 +
78 +
79 +

```

```

56 # builds the initial scoring matrix used for traceback
57 def build_matrix(mymatrix):
58     rows = len(mymatrix)
59     cols = len(mymatrix[0])
60 -
61     for i in range(1, rows):
62
63         for j in range(1, cols):
64             mymatrix[i][j] = calc_score(mymatrix, i, j)
65     return mymatrix
66
67 # gets the max value from the built matrix
68 # Max is the end of the traceback for SW
69 def get_max(mymatrix):
70     max = mymatrix[0][0]
71     mrow = 0
72     mcol = 0
73
74     rows = len(mymatrix)
75     cols = len(mymatrix[0])
76
77     for i in range(1, rows):
78         for j in range(1, cols):
79             if mymatrix[i][j] > max:
80                 max = mymatrix[i][j]
81                 mrow = i
82                 mcol = j
84 - print("max score: ", max)
85     return [mrow, TypeB.END, mcol]
86
87 # print out the best scoring path from the SW matrix
88 def print_matrix(mymatrix):
89     rows = len(mymatrix)
90     cols = len(mymatrix[0])
91     s1 = " " + sequence1
92     s2 = " " + sequence2
93
94
95 - print("Dimensions: r= %2d , c= %2d" % (rows, cols))
96
97     for a in s1:
98         print(a, end="")
99         print(" \t", end="")
100     print("\n", end="")
101
102     for i in range(0, rows):
103         print(s2[i], end="")
104         print(" \t", end="")
105         for j in range(0, cols):
106             print("%02d\t" % (mymatrix[i][j]), end="")
107         print("\n", end="")
108
109 # print out the traceback of the best scoring alignment
110 def print_traceback(mymatrix):
111     # this will print as expected with internal gaps
113 - print("Building traceback...")
114
115     maxv = get_max(mymatrix)
116 - #stash the max score for later
117     max_score = mymatrix[maxv[0]][maxv[-1]]
118
119     # traverse the matrix to find the traceback elements
120     # if more than one path just pick one
121     topstring = ""
122     midstring = ""
123     bottomstring = ""
124
125     # pad the sequences so indexes into the sequences match the matrix
126     indexes
127     asequence1 = "#" + sequence1
128     asequence2 = "#" + sequence2

```

```

80 # builds the initial scoring matrix used for traceback
81 def build_matrix(mymatrix):
82     rows = len(mymatrix)
83     cols = len(mymatrix[0])
84 + row_number=0
85 +
86     for i in range(1, rows):
87         row_number = row_number + 1
88         print("\nRow Number:", row_number)
89         sleep(wait)
90
91         for j in range(1, cols):
92             mymatrix[i][j] = calc_score(mymatrix, i, j)
93     return mymatrix
94
95 # gets the max value from the built matrix
96 # Max is the end of the traceback for SW
97 def get_max(mymatrix):
98     max = mymatrix[0][0]
99     mrow = 0
100     mcol = 0
101
102     rows = len(mymatrix)
103     cols = len(mymatrix[0])
104
105     for i in range(1, rows):
106         for j in range(1, cols):
107             if mymatrix[i][j] > max:
108                 max = mymatrix[i][j]
109                 mrow = i
110                 mcol = j
112 + print("The Maximum Score was: ", max, "\n")
113     return [mrow, TypeB.END, mcol]
114
115 # print out the best scoring path from the SW matrix
116 def print_matrix(mymatrix):
117     rows = len(mymatrix)
118     cols = len(mymatrix[0])
119     s1 = " " + sequence1
120     s2 = " " + sequence2
121
122 +
123 + sleep(wait)
124
125 + print("\nDimensions of The SmithWaterman Matrix: Rows= %2d , Columns=
126 + %2d\n" % (rows, cols))
127 + sleep(wait)
128
129     for a in s1:
130         print(a, end="")
131         print(" \t", end="")
132     print("\n", end="")
133
134     for i in range(0, rows):
135         print(s2[i], end="")
136         print(" \t", end="")
137         for j in range(0, cols):
138             print("%02d\t" % (mymatrix[i][j]), end="")
139         print("\n", end="")
140
141 # print out the traceback of the best scoring alignment
142 def print_traceback(mymatrix):
143     # this will print as expected with internal gaps
144
145 +
146 + sleep(wait)
147 +
148 + print("\n### We Will Now Build The Traceback... ###\n")
149     maxv = get_max(mymatrix)
150 + print(maxv)
151
152 + #stash the max score for later
153     max_score = mymatrix[maxv[0]][maxv[-1]]
154
155 +
156     # traverse the matrix to find the traceback elements
157     # if more than one path just pick one
158     topstring = ""
159     midstring = ""
160     bottomstring = ""
161
162     # pad the sequences so indexes into the sequences match the matrix
163     indexes
164     asequence1 = "#" + sequence1
165     asequence2 = "#" + sequence2

```

128		165	
129	- #this vector is used to store the traceback results	166	+ #_this vector is used to store the traceback results
130		167	
131	traversal_results = []	168	traversal_results = []
132		169	
133	# add the rest of the elements	170	# add the rest of the elements
134	search = True	171	search = True
135	lastelement = False	172	lastelement = False
136		173	
		174	+ # Stores the position so it can track if it is an insertion or deletion
		175	+ # Check if it is a gap or not
		176	+ if max_score < 1:
		177	+ print ("There is no suitable alignment...Check your inputs please!")
		178	+ exit;
		179	+ old_maxv = maxv
		180	+
		181	+
137	while (search):	182	while (search):
138		183	
139	- #debug print("position: %d, %s, %d" % (maxv[0], maxv[1], maxv[-1]))	184	+ # print(" position: %d, %d " % (maxv[0], maxv[-1]))
140	- #store the results	185	+
		186	+ # print("Testing execution, type is", current_type)
		187	+
		188	+ # store the results
141	traversal_results.append(maxv)	189	traversal_results.append(maxv)
		190	+
		191	+ # type_traversal = type(traversal_results)
142		192	
143	maxv = traceback(mymatrix, maxv)	193	maxv = traceback(mymatrix, maxv)
144		194	
145		195	
146	- #catch the trivial case that we are at the end of one of the sequences	196	+ #_catch the trivial case that we are at the end of one of the sequences
147	if (maxv[-1] < 0 or maxv[0] < 0):	197	if (maxv[-1] < 0 or maxv[0] < 0):
148	traversal_results.append(maxv)	198	traversal_results.append(maxv)
149	search= False	199	search= False
150	continue	200	continue
151		201	
152		202	
153	if (mymatrix[maxv[0]][maxv[-1]] == 0 and lastelement == False):	203	if (mymatrix[maxv[0]][maxv[-1]] == 0 and lastelement == False):
154	lastelement = True	204	lastelement = True
155	continue	205	continue
156		206	
157	if(lastelement==True) :	207	if(lastelement==True) :
158	search= False	208	search= False
159	traversal_results.append(maxv)	209	traversal_results.append(maxv)
160	continue	210	continue
161		211	
162		212	
163	for i in range(0, len(traversal_results)-2):	213	for i in range(0, len(traversal_results)-2):
164		214	
165	- #The TypeB of the next element gives how the current element was reached	215	+ # print("Testing execution")
166	- #in the dynamic programming table	216	+
167	- #The current element gives the index of the two matching bases to be aligned	217	+ # The TypeB of the next element gives how the current element was reached
		218	+ # in the dynamic programming table
		219	+ # The current element gives the index of the two matching bases to be aligned
		220	+
168		221	
169	curr_el=traversal_results[i]	222	curr_el=traversal_results[i]
170	next_el=traversal_results[i+1]	223	next_el=traversal_results[i+1]
171		224	
172	- #Match	225	+ #_Match
173	if(next_el[1]==TypeB.MATCH):	226	if(next_el[1]==TypeB.MATCH):
174	bottomstring += asequence2[curr_el[0]]	227	bottomstring += asequence2[curr_el[0]]
175	topstring += asequence1[curr_el[-1]]	228	topstring += asequence1[curr_el[-1]]
176	midstring += " "	229	midstring += " "
		230	+ # print(" position: ", i,i)
		231	+ # print("MATCH###")
		232	+
177		233	
178	#Mismatch	234	#Mismatch
179	elif(next_el[1]==TypeB.MISMATCH):	235	elif(next_el[1]==TypeB.MISMATCH):
180	bottomstring += asequence2[curr_el[0]]	236	bottomstring += asequence2[curr_el[0]]
181	topstring += asequence1[curr_el[-1]]	237	topstring += asequence1[curr_el[-1]]
182	midstring += "."	238	midstring += "."
		239	+ # print(" position: ", i,i)
		240	+ # print("MISMATCH/START?")
183		241	
184	elif(next_el[1]==TypeB.INSERT):	242	elif(next_el[1]==TypeB.INSERT):
185	bottomstring += asequence2[curr_el[0]]	243	bottomstring += asequence2[curr_el[0]]
186	topstring += " "	244	topstring += " "
187	midstring += " "	245	midstring += " "
		246	+ # print(" position: ", i,i)
		247	+ # print("Insertion")
188		248	

```

189         elif(next_el[1]==TypeB.DELETE):
190             bottomstring += "-"
191             topstring += asequence1[curr_el[-1]]
192             midstring += " "

```

```

193
194 -     print("\nFinal Alignment, Score: %d" % max_score)
195
196 -     print(topstring[::-1])
197 -     print(midstring[::-1])
198 -     print(bottomstring[::-1])

```

```

199
200 # build the SW alignment...
201 def perform_smith_waterman():
202     # values for weights
203     global seqmatch
204     global seqmismatch
205     global seqgap
206     global sequence1

```

```

249         elif(next_el[1]==TypeB.DELETE):
250             bottomstring += "-"
251             topstring += asequence1[curr_el[-1]]
252             midstring += " "
253 +         # print(" position: ", i,i )
254 +         # print("DELETED")
255 +
256 +
257 +         print("")
258 +         for element in traversal_results:
259 +             print(element,"\n")
260 +
261 +         sleep(wait)
262 +
263 +         print("\nFinal Alignment, Score: %d\n" % max_score)
264 +
265 +         sleep(wait)

```

```

266
267
268 +     ### Printing the alignment with an effect ###
269 +
270 +     # print(topstring[::-1])
271 +     animated_print(topstring[::-1])
272 +
273 +     # print(midstring[::-1])
274 +     animated_print(midstring[::-1])
275 +
276 +     # print(bottomstring[::-1])
277 +     animated_print(bottomstring[::-1])
278 +
279 +     sleep(wait)
280 +
281 +
282 +     # print("Testing execution")
283 +
284 +
285 +
286 +
287 +     def time_to_pause():
288 +         print("How long do you want the pauses in between important steps to be
in seconds?")
289 +         print("Please input a positive integer value <= 3 OR input 0 to execute
without pauses")
290 +         while True:
291 +             wait_input = input("What is you choice? \n")
292 +             if not wait_input.isdigit():
293 +                 print("TypeError: Please input a numerical value")
294 +                 continue;
295 +             wait = int(wait_input)
296 +             if wait < 0 or wait > 3:
297 +                 print("Invalid choice. Please choose a positive integer value <=
3")
298 +             else :
299 +                 print("You have chosen a value of ", wait," seconds\n")
300 +                 return wait
301 +                 break;
302 +
303 +
304 +     def animated_print(s):
305 +         for c in s:
306 +             sys.stdout.write(c)
307 +             sys.stdout.flush()
308 +             time.sleep(0.25)
309 +         print("")
310 +

```

```

311 # build the SW alignment...
312
313 def perform_smith_waterman():
314     # values for weights
315     global seqmatch
316     global seqmismatch
317     global seqgap
318     global sequence1

```

```
207 global sequence2
```

```
209 - # note these are not the exact weights used in the original SW paper
210 - seqmatch = 1

211 - seqmismatch = -1
212 - seqgap = -1

213

214 - # input sequences- other examples
215 - sequence1="AGTGATAAACTAGTAATTTTT"
216 - sequence2="TTGGGGGTAACAGGGG"

217

218 - # sequence1 ="AGTCGGTTAGTAAA"
219 - # sequence2 ="TTTTGGGTTTAGGCGC"
220

221 - # sequence1 = "GTGTAATTTTTT"
222 - # sequence2 = "AAAAGTGTATT"

223

224 - # sequence1 = "TCGTTCTAG"
225 - # sequence2 = "TCGTTTTTG"
226

227 - # sequence1 = "SimonTomkinson"
228 - # sequence2 = "SimonTomlinsonBioinformaticsAlgorithms"
```

```
319 global sequence2
```

```
320 + global wait
321 +
322 + print("\n\nWelcome to B236494's version of the SmithWaterman.py Script
\n")
323 + print("\nThis is an adapted version of SmithWaterman.py V 1.9 by Simon
Tomlinson\n")
324 + print("\nThis script takes command line inputs for the match/mismatch/gap
scores and user-input during the start of execution for the FASTA files to
be aligned\n")
325 + print("\nEnsure that the files to be aligned are present in the current
directory where the programme is being executed \n")
326 + print("\nExample CLI input could be ---> python3 SmithWaterman.py --
seqmatch 1 --seqmismatch -1 --seqgap -1 \n")
327 +
328 +
329 +
330 + wait = time_to_pause()
331 +
332 +
333 + ### Taking command line input for the match, mismatch and gap penalties
###
334 +
335 + parser = argparse.ArgumentParser(description='Please provide the
parameters to Perform Smith-Waterman Alignment.')
336 + sleep(wait)

337

338 + # Add arguments for sequence match, mismatch, and gap penalties
339 + # note these defaults are not the exact weights used in the original SW
paper

340

341 + parser.add_argument('--seqmatch', type=int, default=1, help='Input the
score for sequence matches. Default is 1.')
342 + parser.add_argument('--seqmismatch', type=int, default=-1, help='Input
the Penalty for sequence mismatches. Default is -1.')
343 + parser.add_argument('--seqgap', type=int, default=-1, help='Input the
penalty for a gap. Default is -1.')

344

345 + args = parser.parse_args()

346

347 + seqmatch = args.seqmatch
348 + seqmismatch = args.seqmismatch
349 + seqgap = args.seqgap

350

351 + sleep(wait)

352

353 + print(f"Using match score: {seqmatch}")
354 + print(f"Using mismatch penalty: {seqmismatch}")
355 + print(f"Using gap penalty: {seqgap}\n")
356 +
357 + sleep(wait)
358 +
359 + ### Function to check if there are any non ATGC characters ###
360 +
361 + def check_if_any_non_atgc_chars(sequence):
362 +     pattern = re.compile(r'[^ATGC]')
363 +     match = pattern.search(sequence.upper())
364 +     return not bool(match)
365 +
366 + ### Function to store sequences that need to be aligned ###
367 +
368 + def read_fasta_filename(filename):
369 +     seq = ""
370 +     with open(filename, 'r') as filehandle:
371 +         for line in filehandle:
372 +             # Using regular expression search to ignore lines starting
with ">"
373 +             if re.search("^>", line.strip()):
374 +                 continue
375 +             seq += line.strip()
376 +         return seq
377 +
378 +
379 + # Taking input for file name
380 +
381 + seq1_file = input("Enter the filename of the first sequence file: ")
382 + seq2_file = input("Enter the filename to the second sequence file: ")
383 +
384 + # Saving sequences to variables to be aligned
385 +
386 + sequence1 = read_fasta_filename(seq1_file)
387 + sequence2 = read_fasta_filename(seq2_file)
388 +
389 +
390 +
```

```

391 +     ### Alternate options to test sequences ###
392 +
393 + #     sequence1="AGTGATAAACTAGTAATTTTT"
394 + #     sequence2="TTGGGGGTAACAGGGG"
395 +
396 + #     sequence1 ="AGTCGGTTAGTAAA"
397 + #     sequence2 ="TTTGGGTTAGGCCG"
398 +
399 + #     sequence1 = "GTGTATTTTTT"
400 + #     sequence2 = "AAAAGTGTATT"
401 +
402 + #     sequence1 = "TCGTTCTAG"
403 + #     sequence2 = "TCGTTTTTG"
404 +
405 + #     sequence1 = "SimonTomkinson"
406 + #     sequence2 = "SimonTomlinsonBioinformaticsAlgorithms"
407 +
408 +
409 +
410 +     ### Checking sequence1 & sequence2 if they contain any characters other
        than A, T, G, C ###
411 +
412 +     check_seq1 = check_if_any_non_atgc_chars(sequence1)
413 +     check_seq2 = check_if_any_non_atgc_chars(sequence2)
414 +
415 +     sleep(wait)
416 +
417 +     if not check_seq1 or not check_seq2:
418 +         print("Error in One or both input sequences. There are characters
        other than A, T, G, and C.")
419 +         print("Please make sure that your input files contain the correct
        sequences and try again.")
420 +         print("Exiting Program...Bye")
421 +         sleep(wait)
422 +         sys.exit()
423 +
424 +
425 +     ### Printing the sequences ###
426 +
427 +     print("The input sequences are\n")
428 +     sleep(wait)
229
230     print("Sequence1: " + sequence1)
231     print("Sequence2: " + sequence2)
232
233     mymatrix = create_matrix(len(sequence2), len(sequence1))
234     mymatrix = build_matrix(mymatrix)
235     print_matrix(mymatrix)
236
237     print_traceback(mymatrix)
238
429
430     print("Sequence1: " + sequence1)
431     print("Sequence2: " + sequence2)
432 +     print("\n") ### Empty line
433
434     mymatrix = create_matrix(len(sequence2), len(sequence1))
435     mymatrix = build_matrix(mymatrix)
436     print_matrix(mymatrix)
437
438     print_traceback(mymatrix)
439
440 +
441 +
442 +     ### Taking the absolute value as EMBOSS water does not permit negative
        values for parameters ###
443 +
444 +     # seqmismatch = abs(seqmismatch)
445 +     seqgap = abs(seqgap)
446 +
447 +
448 +     print("\n\nComparing output with EMBOSS water")
449 +
450 +     sleep(wait)
451 +
452 +     water_command = "water -asequence {} -bsequence {} -gapopen {} -gapextend
        {} -outfile water_{}_{}.water -datafile EDNAFULL_srt".format(seq1_file,
        seq2_file, seqgap, seqgap, seq1_file, seq2_file)
453 +     os.system(water_command)
454 +
455 +     print("\n\nThis is the output from EMBOSS water using the same files
        which is our Gold-Standard for alignment")
456 +
457 +     sleep(wait)
458 +
459 +     display_water_file = "cat water_{}_{}.water".format(seq1_file, seq2_file)
460 +     os.system(display_water_file)
461 +
462 +     sleep(wait)
463 +
464 +     print("\n\nThanks for using this script.\n")
465 +
466 +
467 +
239
240     ##this calls the SW algorithm when the script loads
241     perform_smith_waterman()
468
469     ##this calls the SW algorithm when the script loads
470     perform_smith_waterman()

```