

Project 4 Report

Unrestricted Algorithm Analysis

The unrestricted algorithm simply runs in $O(n * m)$ time due to its nature of dealing with a matrix and performing multiple operations based on the size of the sequences. I'm also taking $O(n*m)$ space in order to keep track of the scoring_matrix and $O(n*m)$ space in order to use it for backtracking and aligning the sequences correctly. After initializing some rows and columns with some initial values, the only other operation that is costly in the algorithm itself is the alignment of the sequences itself, which simply run in $O(n + m)$ time and take $O(n)$ and $O(m)$ space in order to store the resulting strings.

Banded Algorithm Analysis

The banded algorithm runs in $O(k*n)$ time, where in this case, we're using $k = 7$. I've also created two matrices that also keep track of the score and the path in order to backtrack and align the strings in the end. Because I'm using two matrices, they each occupy $O(k*n)$ space. One of the hardest parts of this algorithm was populating each cell correctly and finding once we hit specific indexes. Now, for the alignment of the sequences, we're taking $O(k+n)$ time and $O(n)$ space in which I'm creating two lists in order to generate the strings for the alignment. The alignment itself runs in $O(k*n)$ time, in which n is the number of times we're appending results of the sequences to their respective arrays.

Alignment Algorithm

Creating the alignment string for the unrestricted algorithm was a lot easier to do than the banded algorithm. For the unrestricted algorithm, I start tracing back from the last cell, and I start checking for the store path values (DIAGONAL, LEFT or TOP) in order to know where to go next. I then decrease the respective indices depending on the stored value and then append the correct values and dashes to the final strings. Once I finally hit the beginning indices, I simply break the loop and return the strings built from it.

As for the alignment for the banded algorithm, I am not sure if I was able to fully get it to match the expectations and tried my best to get it work, but due to time I think this might be enough for it. My goal was to do something similar as I did for the unrestricted algorithm - however, the goal was to simply check the stored paths and see which directions it would backtrack me to - I then stored the results in a string and updated the indices accordingly. The hardest part for me was making

sure to keep in mind the bandwidth so I didn't accidentally compute values that shouldn't be computed.

Results

Unrestricted Algorithm:

Gene Sequence Alignment

	sequence1	sequence2	sequence3	sequence4	sequence5	sequence6	sequence7	sequence8	sequence9	sequence10
sequence1	-30	-1	4956	4956	4956	4956	4956	4956	4956	4956
sequence2		-33	4948	4948	4948	4948	4948	4948	4948	4948
sequence3			-3000	-2996	-2956	-2944	-1431	-1448	-1399	-1448
sequence4				-3000	-2960	-2948	-1431	-1448	-1399	-1448
sequence5					-3000	-2988	-1423	-1452	-1391	-1448
sequence6						-3000	-1426	-1452	-1394	-1448
sequence7							-3000	-2771	-2814	-2767
sequence8								-3000	-2731	-2996
sequence9									-3000	-2727
sequence10										-3000

Label I:

Sequence I:

Sequence J:

Label J:

Process

Clear

☐ Banded Align Length: 1000

Done. Time taken: 26.771 seconds.

Banded Algorithm:

Gene Sequence Alignment

	sequence1	sequence2	sequence3	sequence4	sequence5	sequence6	sequence7	sequence8	sequence9	sequence10
sequence1	-30	-1	inf	inf	inf	inf	inf	inf	inf	inf
sequence2		-33	inf	inf	inf	inf	inf	inf	inf	inf
sequence3			-9000	-8984	-8888	-8848	-2735	-2743	-1429	-2735
sequence4				-9000	-8888	-8848	-2739	-2748	-1426	-2740
sequence5					-9000	-8960	-2711	-2739	-1426	-2727
sequence6						-9000	-2708	-2728	-1415	-2716
sequence7							-9000	-8103	-1256	-8099
sequence8								-9000	-1310	-8980
sequence9									-9000	-1315
sequence10										-9000

Label I:

Sequence I:

Sequence J:

Label J:

Process

Clear

☒ Banded Align Length: 3000

Done. Time taken: 0.782 seconds.

Results Alignment

Unrestricted Algorithm Alignment n = 1000

```
Sequence 3: gattgcgagcgatttgcgtagcgatcccgcttc-actg--at-ctcttgtagatcttttcataatctaaactttataaaaacatccactccctgta-
Sequence 10: -ataa-gagtgattggcgtagcgtagcgatcccgcttc-actg--at-ctcttgtagatcttttcataatctaaactttataaaa--cggc-acttcctgtgt
```

Banded Algorithm Alignment n = 3000

```
Sequence 3: attgcgagcgatttgcgtagcgatcccgcttc-actg--at-ctcttgtagatcttttcataatctaaactttataaaaacatccactccctgta-g
Sequence 10: taa-gcgagcgatttgcgtagcgatcccgcttc-actg--at-ctcttgtagatcttttcataatctaaactttataaaaacatccactccctgta-
```

Code

```
import math
import time
import random

# Used to compute the bandwidth for banded version
MAXINDELS = 3

# Used to implement Needleman-Wunsch scoring
MATCH = -3
INDEL = 5
SUB = 1
LEFT = "LEFT"
DIAGONAL = "DIAGONAL"
TOP = "TOP"

class GeneSequencing:

    def __init__(self):
        pass
```

```

# This is the method called by the GUI. _seq1_ and _seq2_ are two sequences to be
aligned, _banded_ is a boolean that tells
# you whether you should compute a banded alignment or full alignment, and
_align_length_ tells you
# how many base pairs to use in computing the alignment

    def align(self, seq1, seq2, banded, align_length):
        self.banded = banded
        self.MaxCharactersToAlign = align_length

#####
#####

# your code should replace these three statements and populate the three variables:
score, alignment1 and alignment2

        if banded:
            score, alignment1, alignment2 = self.banded_alignment(seq1, seq2,
align_length)
        else:
            score, alignment1, alignment2 = self.unrestricted_algo(
                seq1, seq2, align_length)

        # Showing only the first 100 characters in the alignment and revealing the
matches, subs and indels
        alignment1 = '{}'.format(alignment1[:100])
        alignment2 = '{}'.format(alignment2[:100])

#####
#####

        return {'align_cost': score, 'seqi_first100': alignment1, 'seqj_first100':
alignment2}

# Unrestricted Alignment Algorithm
# unrestricted in the number of inserts/deletes that can occur consecutively)
of
# the sequences and compute the alignment score in such a way that the actual
# character-by-character alignment can be extracted.

    def unrestricted_algo(self, seq1, seq2, align_length):

        seq1 = " " + seq1[:align_length] # Adding one character in front to see if it
works
        seq2 = " " + seq2[:align_length]

```

```

seq1_length = len(seq1) # This would be my rows in the table
seq2_length = len(seq2) # This would be my columns in the table

# This table will keep track of computing scores
scoring_matrix = [[None for _ in range(
    seq2_length)]for _ in range(seq1_length)]

# This table will keep track of computing the path in order to compute the
optimal alignment by checking for LEFT, TOP and DIAGONAL
path_matrix = [[None for _ in range(seq2_length)]
    for _ in range(seq1_length)]

# Initializing first column and row of both the matrix'
scoring_matrix[0][0] = 0
path_matrix[0][0] = None

# Initializing initial row with costs
for i in range(1, seq2_length):
    scoring_matrix[0][i] = INDEL * i
    path_matrix[0][i] = LEFT

# Initializing initial column with costs
for j in range(1, seq1_length):
    scoring_matrix[j][0] = INDEL * j
    path_matrix[j][0] = TOP

for i in range(1, seq1_length):
    for j in range(1, seq2_length):
        # Left beats all ties
        cost = scoring_matrix[i][j - 1] + INDEL # storing the cost in order
for it to be calculated after tie breaker
        path_matrix[i][j] = LEFT

        # Top is second, and diagonal is next
        if scoring_matrix[i - 1][j] + INDEL < cost:
            cost = scoring_matrix[i - 1][j] + INDEL
            path_matrix[i][j] = TOP

        if scoring_matrix[i - 1][j - 1] + self.match(i, j, seq1, seq2) < cost:
            cost = scoring_matrix[i -
                                                                    1][j - 1] + self.match(i, j,
seq1, seq2)

            path_matrix[i][j] = DIAGONAL

        scoring_matrix[i][j] = cost # only assigning cost after tie break per
iteration

```

```

# print(scoring_matrix)
# print(path_matrix)

i = len(seq1) - 1
j = len(seq2) - 1
seq1_align = seq1
seq2_align = seq2
cost = scoring_matrix[seq1_length-1][seq2_length-1]

# Backtracking through the matrix in order to find the alignments made
while True:
    if i != 0 and j != 0 and path_matrix is None:
        break
    elif i == 0 and j == 0:
        break
    else:
        if path_matrix[i][j] == DIAGONAL:
            i -= 1
            j -= 1
            continue
        elif path_matrix[i][j] == LEFT:
            seq1_align = seq1_align[: i + 1] + "-" + seq1_align[i + 1:]
            j -= 1
            continue
        elif path_matrix[i][j] == TOP:
            seq2_align = seq2_align[: j + 1] + "-" + seq2_align[j + 1:]
            i -= 1

    return cost, seq1_align[1:], seq2_align[1:]

# This is a helper function that will help break the tie on whether to add a SUB
or MATCH value depending whether
# the values at the array are the same or not
# O(1)
def match(self, i, j, seq1, seq2):
    if seq1[i] == seq2[j]:
        return MATCH
    return SUB

# Banded Alignment Algorithm:
# A banded alignment means that we will only consider alignments in
# which the ith character from sequence A and the ith character from
# sequence B are within some small distance d of one another.

```

```
# Restricting ourselves to such alignments means that we will only compute
# scores for a band around the diagonal of the scoring matrix, with bandwidth
2d+1.
```

```
def banded_alignment(self, seq1, seq2, align_length):
    # Deciding whether the length should be the align length or length of
sequences
    if (len(seq1) > align_length):
        seq1_len = align_length + 1
    else:
        seq1_len = len(seq1) + 1

    if (len(seq2) > align_length):
        seq2_len = align_length + 1
    else:
        seq2_len = len(seq2) + 1

    seq1 = seq1[:seq1_len-1]
    seq2 = seq2[:seq2_len-1]

    # If no alignment is needed, just insert infinity
    if abs(seq1_len - seq2_len) > 3:
        return float('inf'), '', ''

    d = 3
    k = 7

    # Initialize the scoring and path matrix, O(k*n)
    scoring_matrix = [[None for i in range(k)] for j in range(seq1_len)]
    path_matrix = [[None for i in range(k)] for j in range(seq1_len)]
    scoring_matrix[0][0] = 0

    # Populating both the scoring and path matrix,
    # O(n*k), n is the length of seq1
    for i in range(seq1_len):
        seq2_offset = '' # keeps track of seq2 j value so I can compare later
whether should SUB or MATCH
        if i < 4:
            seq2_offset = ' ' + seq2[:k] # Insert one extra value to offset
        else:
            seq2_offset = seq2[i-d-1:k+(i-d)] # keep track of offset

        for j in range(k):
            seq1_offset = seq1[i-1] # SUB OR MATCH
```

```

seq2_j = ''
if j >= len(seq2_offset):
    continue
else:
    seq2_j = seq2_offset[j]
left = 0
top = 0
diagonal = 0

if i < 4:
    # Populating some of the more easy values for i = 0 and j = 0
    if i == 0:
        scoring_matrix[i][j] = INDEL * j
        path_matrix[i][j] = LEFT
    if j == 0:
        scoring_matrix[i][j] = INDEL * i
        path_matrix[i][j] = TOP

    left = scoring_matrix[i][j-1]
    top = scoring_matrix[i-1][j]
    diagonal = scoring_matrix[i-1][j-1]
# Need to make sure I'm keeping of the correct positions for j index
elif j == 6:
    left = scoring_matrix[i][j-1]
    top = float('inf')
    diagonal = scoring_matrix[i-1][j]

elif j == 0:
    left = float('inf')
    top = scoring_matrix[i-1][j+1]
    diagonal = scoring_matrix[i-1][j]

else:
    left = scoring_matrix[i][j-1]
    top = scoring_matrix[i-1][j+1]
    diagonal = scoring_matrix[i-1][j]

if seq1_offset != seq2_j:
    diagonal = diagonal + SUB
else:
    diagonal = diagonal + MATCH

# LEFT priority
if left + INDEL <= diagonal and left + INDEL <= top + INDEL:

```



```

        scoring_matrix[i][j] = left + INDEL
        path_matrix[i][j] = LEFT
    # TOP
    elif top + INDEL <= diagonal and top + INDEL <= left + INDEL:
        scoring_matrix[i][j] = top + INDEL
        path_matrix[i][j] = TOP
    # DIAGONAL
    else:
        scoring_matrix[i][j] = diagonal
        path_matrix[i][j] = DIAGONAL

    # Finding the final score
    for scr in scoring_matrix[-1]:
        if scr != None:
            score = scr

    alignment1, alignment2 = self.make_alignment(path_matrix, seq1, seq2,
align_length)

    return score, alignment1[1:], alignment2[1:]
# TODO: Double check the returned values with examples to see if match
def make_alignment(self, path_matrix, seq1, seq2, align_length):

    d = 3
    i = len(seq1) - 1
    j = 0

    seq1_align = seq1[:align_length]
    seq2_align = seq2[:align_length]

    while path_matrix[i][j] != None:
        j+=1

    j -= 1
    path_matrix[0][0] = 'START'
    path = path_matrix[i][j]

    while path != 'START':
        if path == DIAGONAL:
            if i > d:
                i -= 1
                path = path_matrix[i][j]
            else:
                i -= 1

```

```

        j -= 1
        path = path_matrix[i][j]

    elif path == LEFT:
        j-=1
        path = path_matrix[i][j]
        seq1_align = seq1_align[:i] + "-" + seq1_align[i:]
    elif path == TOP:
        if i > d:
            i -= 1
            j += 1
            path = path_matrix[i][j]

        else:
            i -= 1
            path = path_matrix[i][j]

        if i > 4:
            seq2_align = seq2_align[:i - d + j] + "-" + seq1_align[i - d + j:]
        else:
            seq2_align = seq2_align[:i - 1] + "-" + seq1_align[i - 1:]
    else:
        path = "START"

    return seq1_align, seq2_align

```