Jeff Reimschussel

Project 4 Report

**Code**

def unrestricted\_algorithm(sequence\_a, sequence\_b, align\_length):

# Allow room for empty character in sequences in alignment table

longest\_sequence = max(len(sequence\_a), len(sequence\_b))

table\_width = min(align\_length+1, longest\_sequence+1)

# Allow for sequence length differences

if len(sequence\_b) <= table\_width:

fill = "-" \* (table\_width - len(sequence\_b))

sequence\_b += fill

if len(sequence\_a) <= table\_width:

fill = "-" \* (table\_width - len(sequence\_a))

sequence\_a += fill

alignment\_table = [[(0, "", "")]]

for i in range(1, table\_width):

score = i \* INDEL

fill = '-' \* i

alignment\_table[0].append((score, fill, fill))

for i in range(1, table\_width):

score = i \* INDEL

fill = '-' \* i

row = [(0, fill, fill)]

char\_a = sequence\_a[i-1]

for j in range(1, table\_width):

char\_b = sequence\_b[j-1]

# Find best neighboring score

best\_score = row[j-1][0] + INDEL

best\_a = row[j-1][1] + '-'

best\_b = row[j-1][2] + '-'

if alignment\_table[i-1][j][0] < best\_score:

best\_score = alignment\_table[i-1][j][0] + INDEL

best\_a = alignment\_table[i-1][j][1] + '-'

best\_b = alignment\_table[i-1][j][2] + '-'

if (char\_a == char\_b and alignment\_table[i-1][j-1][0] + MATCH < best\_score):

best\_score = alignment\_table[i-1][j-1][0] + MATCH

best\_a = alignment\_table[i-1][j-1][1] + char\_a

best\_b = alignment\_table[i-1][j-1][2] + char\_b

if (char\_a != '-' and char\_b != '-' and

alignment\_table[i-1][j-1][0] + SUB < best\_score):

best\_score = alignment\_table[i-1][j-1][0] + SUB

best\_a = alignment\_table[i-1][j-1][1] + char\_a

best\_b = alignment\_table[i-1][j-1][2] + char\_b

row.append((best\_score, best\_a, best\_b))

alignment\_table.append(row)

return backtrace(alignment\_table, table\_width - 1, table\_width - 1, a, b)

def banded\_algorithm(sequence\_a, sequence\_b, align\_length):

print\_at\_end = (sequence\_a == 'polynomial' or sequence\_b == 'exponential')

d = 3

table\_width = 2\*d + 1

# If difference between a & b is greater than the band width, return inf

if abs(len(sequence\_a) - len(sequence\_b)) > align\_length:

return float('inf')

# Allow room for empty character in sequences in alignment table

longest\_sequence = max(len(sequence\_a), len(sequence\_b))

table\_height = min(align\_length, longest\_sequence)

alignment\_table = [[None for i in range(table\_width)] for j in range (table\_height)]

if len(sequence\_b) < table\_height:

fill = "-" \* (table\_height - len(sequence\_b))

sequence\_b += fill

if len(sequence\_a) < table\_height:

fill = "-" \* (table\_height - len(sequence\_a))

sequence\_a += fill

# Populate the first row

alignment\_table[0][d] = (0, "", "")

for idx in range(d+1, table\_width):

i = idx - d

score = i \* INDEL

substr = '-' \* i

alignment\_table[0][idx] = (score, substr, substr)

for i in range(1, table\_height):

start = max(d - i, 0)

rows\_left = table\_height - i - 1

if rows\_left <= d:

off = d - rows\_left

end = table\_width - off

else:

end = table\_width

char\_a = sequence\_a[i-1]

for j in range(start, end):

offset = j - d

char\_b = sequence\_b[i + offset - 1]

# Find best neighboring score

best\_score = float('inf')

best\_a = ''

best\_b = ''

# Top MATCH

if (char\_a == char\_b and char\_a != '-' and alignment\_table[i-1][j] is not None and

best\_score > alignment\_table[i-1][j][0] + MATCH):

best\_score = alignment\_table[i-1][j][0] + MATCH

best\_a = alignment\_table[i-1][j][1] + char\_a

best\_b = alignment\_table[i-1][j][2] + char\_b

# Top SUB

if (char\_a != '-' and char\_a != '-' and alignment\_table[i-1][j] is not None and

best\_score > alignment\_table[i-1][j][0] + SUB):

best\_score = alignment\_table[i-1][j][0] + SUB

best\_a = alignment\_table[i-1][j][1] + char\_a

best\_b = alignment\_table[i-1][j][2] + char\_b

# Left INDEL

if (j - 1 >= 0 and alignment\_table[i][j-1] is not None and

best\_score > alignment\_table[i][j-1][0] + INDEL):

best\_score = alignment\_table[i][j-1][0] + INDEL

best\_a = alignment\_table[i][j-1][1] + '-'

best\_b = alignment\_table[i][j-1][2] + '-'

# Top right INDEL

if (j + 1 < end and alignment\_table[i-1][j+1] is not None and

best\_score > alignment\_table[i-1][j+1][0] + INDEL):

best\_score = alignment\_table[i-1][j+1][0] + INDEL

best\_a = alignment\_table[i-1][j+1][1] + '-'

best\_b = alignment\_table[i-1][j+1][2] + '-'

alignment\_table[i][j] = (best\_score, best\_a, best\_b)

return backtrace(alignment\_table, table\_width - 1, d, a, b)

def backtrace(alignment\_table, i, j, a, b):

score = alignment\_table[i][j][0]

a\_align = ''

b\_align = ''

a\_idx = len(a) - 1

b\_idx = len(b) - 1

while i > 0 and j > 0:

if alignment\_table[i][j][3] == MATCH or alignment\_table[i][j][3] == SUB:

a\_align = a[a\_idx] + a\_align

b\_align = b[b\_idx] + b\_align

a\_idx -= 1

b\_idx -= 1

if alignment\_table[i][j][4] == TOP\_BANDED:

i -= 1

else:

i -= 1

j -= 1

elif alignment\_table[i][j][4] == TOP:

a\_align = '-' + a\_align

i -= 1

b\_idx -= 1

elif alignment\_table[i][j][4] == TOP\_RIGHT\_BANDED:

a\_align = '-' + a\_align

i -= 1

j += 1

b\_idx -= 1

elif alignment\_table[i][j][4] == LEFT:

b\_align = '-' + b\_align

j -= 1

a\_idx -= 1

return (score, a\_align, b\_align)

**Analysis**

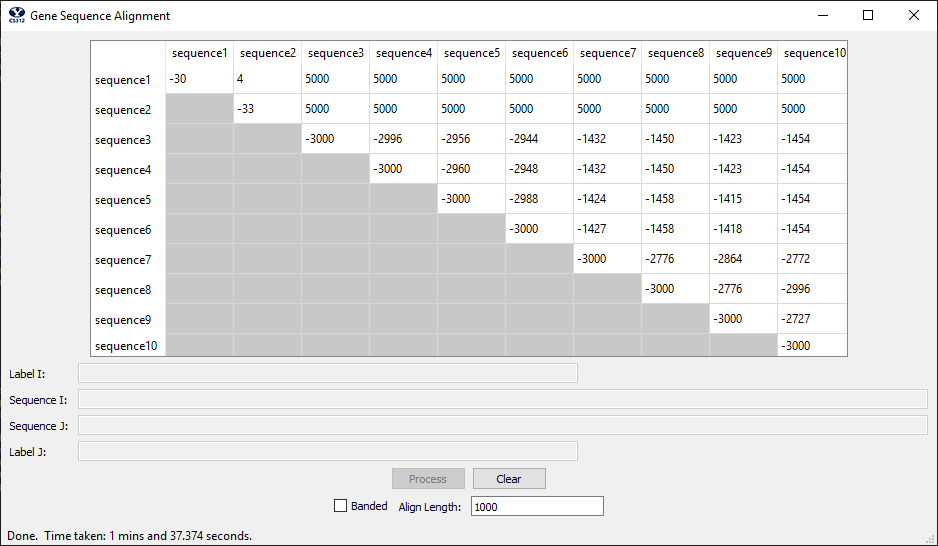
The unrestricted algorithm is called once. The biggest time complexity comes from the looping through every j columns, in every i rows. The number of rows is at worst the align length, and at best the length of the longest of the two sequences. Every iteration only does indexing and assignment; there’s no need to check every other element. Ultimately the time complexity for iterating through to find the best score is at worst O(nm).

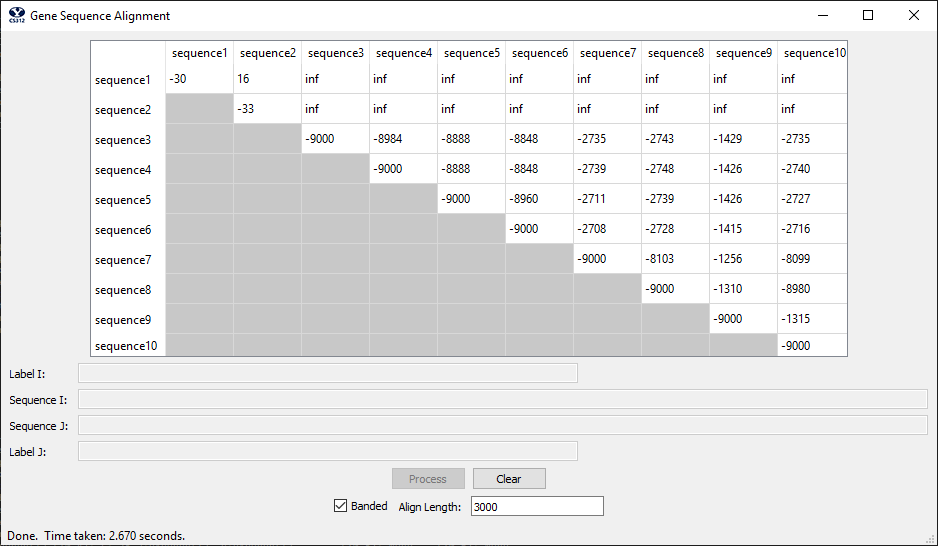
Creating the alignment table requires entries for every n columns in every m rows. Each iteration adds one entry to the table. Apart from adding an entry into the table in every iteration, no new space is required for this algorithm. So the total space complexity is also O(nm).

The banded alignment requires less space. Because the algorithm only considers entries within d range, the width is restricted down to k columns, where k is the size of 2d + 1. Apart from this change in space, the basic structure of the algorithm is still the same; two loops iterating through a table, and simple indexing and assignment in every iteration. So substituting k for m (because of the restricted size), the time and space complexities become O(kn).

The backtrace function tries to find the extracted alignment found in the graph. In order to do so, both the banded and unrestricted algorithms store in each entry what action was taken to get there, and what entry it came from. Then when backtracing, the algorithm checks which action; if it was a match or a substitution, the algorithm adds the corresponding characters for each sequence and moves character indices to the place in the string for the previous entry. If it was an indel, only the index that needs to change is changed (i.e. in the unrestricted algorithm, if the previous entry was the top, then the character index for the row is changed. If the previous entry was the left, then the character index for the column is changed).

**Results**





Alignment (3, 10) unrestricted