Gene Sequencing Alignment

March 20, 2020

Kameron Lightheart
CS 312

1 Complexity

1.1 Unrestricted

1.1.1 Temporal

Assuming an input of 2 strings of length n and m respectively, the unrestricted algorithm is O(nm). See commented code in section 4 for details, but basically the algorithm loops through every entry of a nxm matrix E and nxm matrix prev simulatneously and computes the value in constant time and sets the value of both matrices. This is O(nm), since all the other computations or initializations are constant time (since np.empty only allocated memory, it doesn't need to loop through and set any values like np.zeros does) or O(n+m) in the case of backtracking the alignments. The reason for this is because worst case scenario is that none of the characters are matched up, but instead there are all gaps, then the alignment backtrack while loop is run n+m times with constant time operations which gives O(n+m). Thus overall the algorithm is O(nm).

1.1.2 Spacial

Assuming an input of 2 string of size n and m respectively, the unrestricted algorithm has spacial complexity of O(nm). This is because 2 arrays of size nm are used, along with several integers, and the two align lists that are each max size n+m if no matches are chosen. Thus overall the space needed is O(nm).

1.2 Banded

1.2.1 Temporal

Assuming an input of 2 strings of size n and m respectively, the banded algorithm has complexity of O(km) where k is the bandwidth chosen (in our case 7). See commented code for details. This algorithm is almost identical to the previous one except that instead of looping over all values of the nm matrix, we only loop through the diagonal and the 7 values around it (including the diagonal). Since my inner for loop depends on the iterator from the outer for loop, it only spans the 7 values in each row of the matrix. This means that overal the double for loop is entered k*m times and with constant time operations inside that gives a total of O(km) for the algorithm. Similar to before the align backtrack is O(n+m) so it doesn't affect the overall asymptotic complexity.

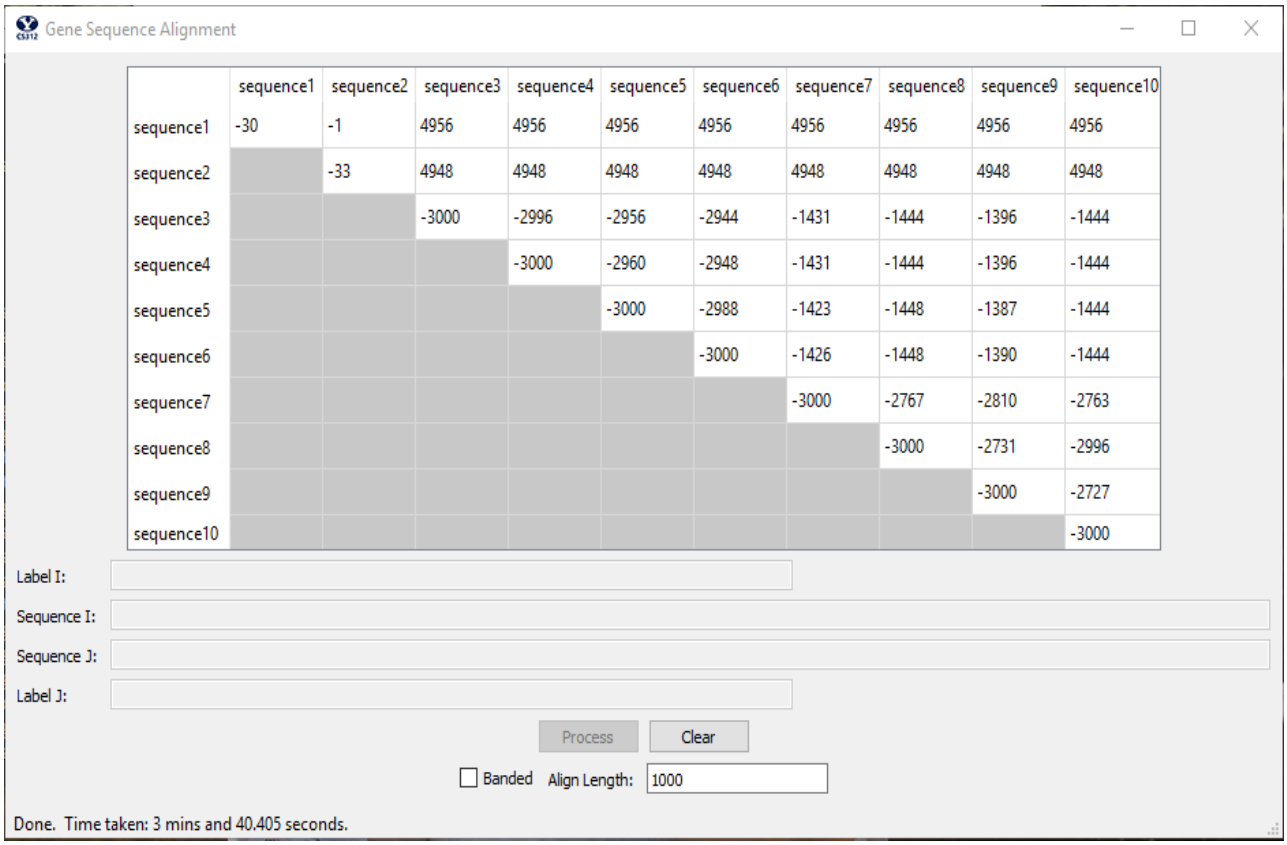
1.2.2 Spacial

Assuming an input of 2 strings of size n and m respectively, the banded algorithm has spacial complexity of O(nm). This is because I used the exact same space as the unrestricted for simplicity. I could have done it in O(km), but I was unable to get this to work, so I ended up leaving the same size matrix and just changing the for loops to only enter the desired entries of the matrices.

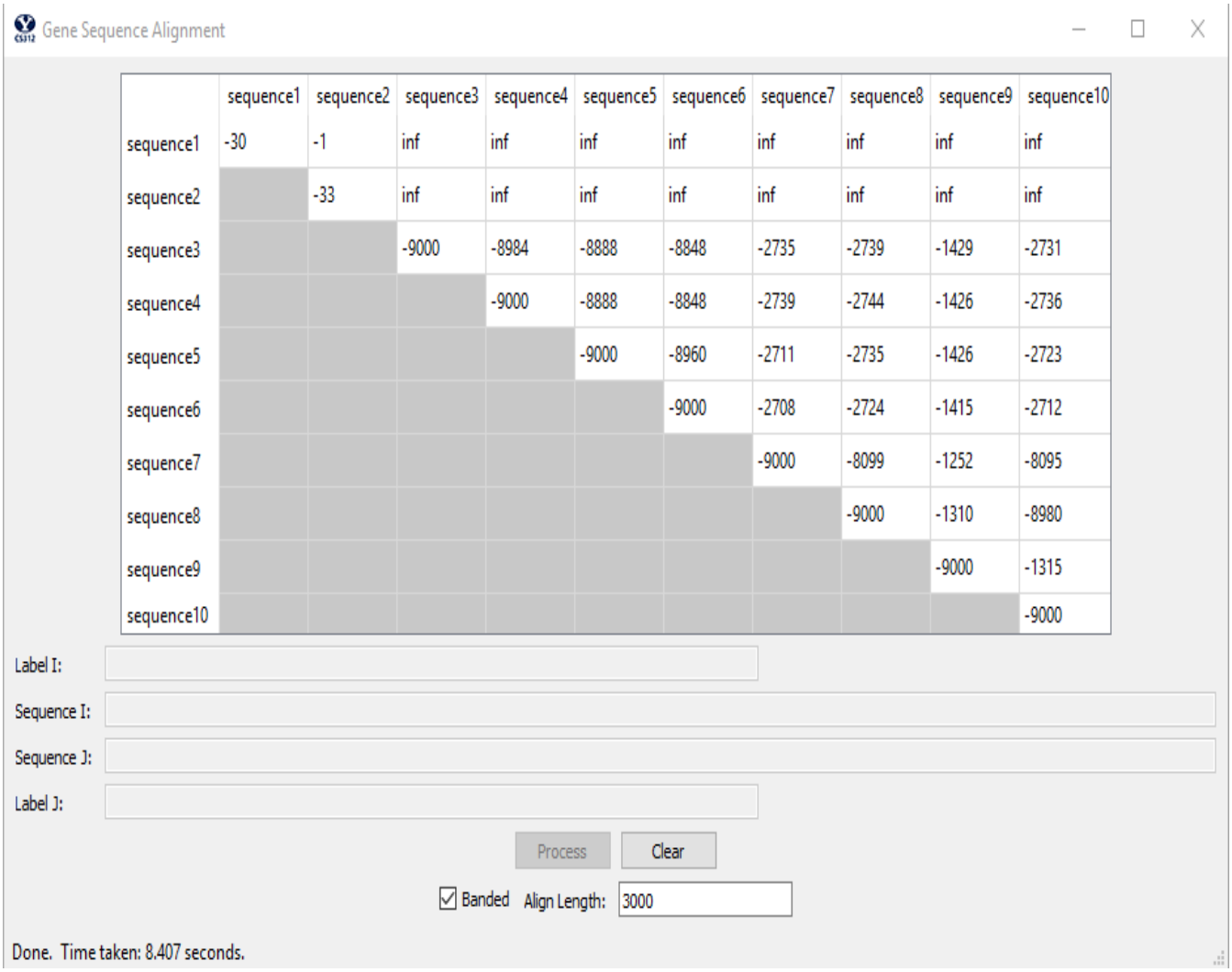
2 Alignment Extraction Algorithm

In order to be able to easily obtain the optimal alignments for each pair of strings, I kept track of the path as I computed the values of the table. I did this by creating another matrix of size n*m which I called "prev". This matrix was filled with values of either 0, 1 or 2. These values represent whether a letter from the first string is added which means the value to the left of the current was chosen (0), a letter from the second string is added which means the value above the current was chosen (1) or both letters were added which means the value diagonal to the current was chosen (2). This allowed me to start from the bottom right entry of the matrix and use these "prev" matrix values to tell me which direction to move in the matrix towards the top left entry. Since each movement corresponds to either adding from the 1st, 2nd or both strings, I built a list of those strings for both alignments. At the end since I worked backwards, I simply reversed the lists and converted them to strings.

3 Results



	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	-1444	-1396	-1444
sequence4				-3000	-2960	-2948	-1431	-1444	-1396	-1444
sequence5					-3000	-2988	-1423	-1448	-1387	-1444
sequence6						-3000	-1426	-1448	-1390	-1444
sequence7							-3000	-2767	-2810	-2763
sequence8								-3000	-2731	-2996
sequence9									-3000	-2727
sequence10										-3000



4 -

4.1 Unrestricted 100 Character Alignment (sequence 3 and 10)

```
--attg-gagcgatttgcgtgcgtgcatcccgcttccactgggattctcttgttagatcttttc-taatctaaactttataaa--catc-actccctgta
---taaagagtgattggcgtccgtacgtacccttttc-actc--aa-ctcttgttagtttaaatcctaatctaaactttataaaaacggccacttcctgtg
```

4.2 Banded 100 Character Alignment (sequence 3 and 10)

```
--attg-gagcgatttgcgtgcgtgcatcccgcttccactgggattctcttgttagatcttttc-taatctaaactttataaa--catc-actccctgta
---taaagagtgattggcgtccgtacgtacccttttc-actc--aa-ctcttgttagtttaaatcctaatctaaactttataaaaacggccacttcctgtg
```

5 -

[3]:
```
#!/usr/bin/python3

#from PyQt5.QtCore import QLineF, QPointF



import math
import time
import numpy as np

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

# Used to implement Needleman-Wunsch scoring
MATCH = -3
INDEL = 5
SUB = 1

class GeneSequencing:

    def __init__( self ):
        pass

# This is the method called by the GUI.  _sequences_ is a list of the ten␣
 ↪sequences, _table_ is a
# handle to the GUI so it can be updated as you find results, _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, sequences, table, banded, align_length ):
        self.banded = banded
        self.MaxCharactersToAlign = align_length
        results = []

        for i in range(len(sequences)):
            jresults = []
            for j in range(len(sequences)):
                if j < i:
                    s = {}
                else:
###############################################################################################
# your code should replace these three statements and populate the three␣
 ↪variables: score, alignment1 and alignment2
                    str1, str2 = '-' + sequences[i], '-' + sequences[j]
```

```
                    if len(sequences[i]) > len(sequences[j]):
                        temp = str1
                        str1 = str2
                        str2 = temp
                    if banded:
                        score, alignment1, alignment2 = self.
↪banded_path_dist('-' + sequences[i], '-' + sequences[j])
                    else:
                        score, alignment1, alignment2 = self.calc_path_dist('-'␣
↪+ sequences[i], '-' + sequences[j])

                    if i == 2 and j == 9:
                        print(alignment1)
                        print(alignment2)

                    # score = i + j
                    # alignment1 = 'abc-easy  DEBUG:(seq{}, {}␣
↪chars,align_len={}{})'.format(i+1,
                    #     len(sequences[i]), align_length, ',BANDED' if banded␣
↪else '')
                    # alignment2 = 'as-123--  DEBUG:(seq{}, {}␣
↪chars,align_len={}{})'.format(j+1,
                    #     len(sequences[j]), align_length, ',BANDED' if banded␣
↪else '')
##############################################################################################
↪
                    s = {'align_cost':score, 'seqi_first100':alignment1,␣
↪'seqj_first100':alignment2}
                    table.item(i,j).setText('{}'.format(int(score) if score !=␣
↪math.inf else score))
                    table.repaint()
                jresults.append(s)
            results.append(jresults)
        return results

    def calc_path_dist(self, str1, str2):
        """Calculate the optimal edit distance and optimal alignments
        for the given 2 strings.
        """
        # 6 operations
        m,n = len(str1), len(str2)
        if m-1 > self.MaxCharactersToAlign:
            m = self.MaxCharactersToAlign + 1
        if n-1 > self.MaxCharactersToAlign:
            n = self.MaxCharactersToAlign + 1
        # 5 operations (np.empty only allocates memory)
```

```
    E = np.empty((m,n))
    prev = np.empty((m,n))
    start = 1
    if str1[1] == str2[1]:
        start = -3

    # n + m operations
    E[:,0] = 0
    E[0,:] = 0

    # 2*(m-1) operations
    E[1:,1] = np.arange(start, 5*(m-1)+start, 5)
    prev[1:,1] = np.ones(m-1)

    # 2*(n-1)+1 operations
    E[1,1:] = np.arange(start, 5*(n-1)+start, 5)
    prev[1,1:] = np.zeros(n-1)
    prev[1,1] = 2

    def diff(i,j):
        if str1[i] == str2[j]:
            return -3
        else:
            return 1

    # 4*(m-2)*(n-2) operations (4 from the inside operations)
    for i in range(2, m):
        for j in range(2, n):
            # 4 operations including the == check in diff
            temp = [E[i-1,j] + 5, E[i,j-1]+5, E[i-1,j-1]+diff(i,j)]
            index = np.argmin(temp)
            E[i,j] = temp[index]
            # index of 0,1,2 represents left, up and diagonal respectively
            prev[i,j] = index

    align1, align2 = [], []
    # 2 operations
    i,j = m-1,n-1

    # While loop is entered at most n+m times if none of the letters are
    # matched up directly, but have gaps everywhere, 5 operations inside
    # total 5(n+m) = O(n+m)
    while True:
        prev_val = prev[i,j]
        if i == 0:
            align1.append(str1[i])
            align2.append("-")
```

```
                j -= 1
            elif j == 0:
                align1.append("-")
                align2.append(str2[j])
                i -= 1
            elif prev_val == 0:
                align1.append("-")
                align2.append(str2[j])
                i -= 1
            elif prev_val == 1:
                align1.append(str1[i])
                align2.append("-")
                j -= 1
            else:
                align1.append(str1[i])
                align2.append(str2[j])
                i -= 1
                j -= 1

            if i == 0 and j == 0:
                break

        # Join is O(n+m) each time
        return E[-1,-1], ''.join(align1[::-1]), ''.join(align2[::-1])

    def banded_path_dist(self, str1, str2):
        # 6 operations
        m,n = len(str1), len(str2)
        if m-1 > self.MaxCharactersToAlign:
            m = self.MaxCharactersToAlign + 1
        if n-1 > self.MaxCharactersToAlign:
            n = self.MaxCharactersToAlign + 1
        # Insignificant even though hypothetically it would be O(nm) since
        # all of the entries are initiallized and added to, but numpy does
        # it very quickly so we can essentially consider it constant time
        # or at least less than O(k*m) (the overall complexity here)
        E = np.zeros((m,n)) + np.inf
        prev = np.zeros((m,n))
        start = 1
        if str1[1] == str2[1]:
            start = -3

        # 2*(m-1) operations
        E[1:,1] = np.arange(start, 5*(m-1)+start, 5)
        prev[1:,1] = np.ones(m-1)
        # 2*(n-1)+1 operations
        E[1,1:] = np.arange(start, 5*(n-1)+start, 5)
```

```
        prev[1,1:] = np.zeros(n-1)
        prev[1,1] = 2

        def diff(i,j):
            if str1[i] == str2[j]:
                return -3
            else:
                return 1

        # 4*(k)*(m-2) operations (4 from the inside operations)
        for i in range(2, m):
            for j in range(i-3, i+4):
                if j < 2 or j >= n:
                    continue
                # 4 operations including == from diff call
                temp = [E[i-1,j] + 5, E[i,j-1]+5, E[i-1,j-1]+diff(i,j)]
                index = np.argmin(temp)
                E[i,j] = temp[index]
                # index of 0,1,2 represents left, up and diagonal respectively
                prev[i,j] = index

        if E[-1,-1] == np.inf:
            return np.inf, "No Alignment Possible", "No Alignment Possible"

        align1, align2 = [], []
        # 2 operations
        i,j = m-1,n-1

        # While loop is entered at most n+m times if none of the letters are
        # matched up directly, but have gaps everywhere, 5 operations inside
        # total 5(n+m) = O(n+m)
        while True:
            prev_val = prev[i,j]
            if i == 0:
                align1.append(str1[i])
                align2.append("-")
                j -= 1
            elif j == 0:
                align1.append("-")
                align2.append(str2[j])
                i -= 1
            elif prev_val == 0:
                align1.append("-")
                align2.append(str2[j])
                i -= 1
            elif prev_val == 1:
                align1.append(str1[i])
```

```
                align2.append("-")
                j -= 1
            else:
                align1.append(str1[i])
                align2.append(str2[j])
                i -= 1
                j -= 1

            if i == 0 and j == 0:
                break

        return E[-1,-1], ''.join(align1[::-1]), ''.join(align2[::-1])
```