Thomas Neyman
CS 4412 - Advanced Algorithms
March 17 2022
Project 4

Gene Sequencing

Algorithm Analysis

Unrestricted Algorithm:

```
        n = len(seq1)
        m = len(seq2)
        score = self.init_matrix(m + 1, n + 1)
```

Lines 1 and 2 are all simple assignments, so they are constant time. The third line however, initiates the matrix of size *n* x *m* with 0's.

```
    def init_matrix(self, rows, cols):
        matrix = []
        for x in range(rows):
            matrix.append([])
            for y in range(cols):
                matrix[-1].append(0)
        return matrix
```

Because it will append 0's for every row against every column, this function runs in an *n* x *m* time and now takes up *n* x *m* space. The current time & space complexity are O(*nm*)
Next the function will begin assigning values for every spot in the matrix.

```
        for i in range(0, m + 1):
            score[i][0] = INDEL * i

        for i in range(0, n + 1):
            score[0][i] = INDEL * i
```

First, 2 loops are run to simply fill in all of the outside spots since these spots will always have the same score which is the gap score multiplied by its position. Each loop takes either *n* or *m* time. Because O(*n*) and O(*m*) are not greater than O(*n* x *m*) the time complexity remains the same. The space complexity also remains the same. No new data structures have been created.

```
        for i in range(1, m + 1):
            for j in range(1, n + 1):
                # Calculate the scores
                match = score[i - 1][j - 1] + self.match_score(seq1[j - 1],
seq2[i - 1])
                delete = score[i - 1][j] + INDEL
                insert = score[i][j - 1] + INDEL
                score[i][j] = min(match, delete, insert)
```

We then run another nested loop *n* x *m* times. Inside each iteration, only assignment calls are being made, which are all constant time, so the time complexity doesn't increase past O(*nm*). Once again, the matrix is only being modified and nothing new is being created, so the space complexity remains O(*nm*). Once complete, the matrix will contain every score for alignment.

```
        while i > 0 and j > 0:
            score_current = score[i][j]
            score_diagonal = score[i - 1][j - 1]
            score_up = score[i][j - 1]
            score_left = score[i - 1][j]

            if score_current == score_diagonal + self.match_score(seq1[j - 1],
seq2[i - 1]):
                align1 += seq1[j - 1]
                align2 += seq2[i - 1]
                i -= 1
                j -= 1
            elif score_current == score_up + INDEL:
                align1 += seq1[j - 1]
                align2 += '-'
                j -= 1
            elif score_current == score_left + INDEL:
                align1 += '-'
                align2 += seq2[i - 1]
                i -= 1
```

Now we need to backtrack through the matrix we created in order to generate our sequences. Because the backtracking is just comparing what the optimal path will be going backwards from its current position, it will never run more than O(*nm*) times. Worst case would be if it backtracked all the way up, then all the way left, essentially backtracking n then m times, but we know this is most likely not going to happen. The best case scenario would be a complete diagonal line backtracking. Assuming the matrix has *n* and *m* of the same length, this would be *n* or *m* log *n* or *m* calculations total. O(*n log n*) is still less than O(*nm*) so the time complexity for

this algorithm remains O(*nm*). We are also storing our 2 sequences as strings as we backtrack. When finished, the strings will be of length O(*n log n*) as well, which combined are still less than the current space complexity of O(*nm*).

```
        # If i became 0 before j did, loop until j becomes 0 as well
        while j > 0:
            align1 += seq1[j - 1]
            align2 += '-'
            j -= 1

        # If j became 0 before i did, loop until i becomes 0 as well
        while i > 0:
            align1 += '-'
            align2 += seq2[i - 1]
            i -= 1
```

Finally, in case i and j do not converge to 0 together from the while loop just before, we have to make sure the sequence ends at 0,0. In reality, these loops will run for 2 - 5 iterations and likely not much more, but even if they ran for a total of *n* or *m* times, it would not affect the current time complexity. The space complexity remains the same as well.

```
        # flip the alignment paths
        align1 = align1[::-1]
        align2 = align2[::-1]
```

Finally, since we backtracked, the alignments are going to be backwards, so we need to flip them. Python's slicing functionality likely runs anywhere from constant to O(*n*) time, so it doesn't affect the time complexity and the space complexity remains the same.

After all those steps, the final alignments are returned and we are left with our final time complexity of O(*nm*) and space complexity of O(*nm*). The total amount of time calculations are as follows:
O(nm) + O(n) + O(m) + O(nm) + O(n logn) + *unrealistic* O(n) + *likely* O(n) = O(nm)
Space calculations:
O(nm) + O(n logn) + O(m logm) = O(nm)

Banded Algorithm:

The only difference between the unrestricted algorithm and the banded algorithm is the section of code that loops through the matrix assigning values.

```
        for i in range(0, 4):
            score[i][0] = INDEL * i

        for i in range(0, 4):
            score[0][i] = INDEL * i
```

When assigning the edges, they will never travel more than 3 spaces away from 0,0 so only filling the next 3 spaces that extend from 0,0 is sufficient. This is just a constant time operation O(1).

```
        for i in range(1, m + 1):
            for j in range(1, n + 1):
                if j - i <= MAXINDELS and i - j <= MAXINDELS:
                    match = score[i - 1][j - 1] + self.match_score(seq1[j - 1],
seq2[i - 1])
                    delete = score[i - 1][j] + INDEL
                    insert = score[i][j - 1] + INDEL
                    score[i][j] = min(match, delete, insert)
```

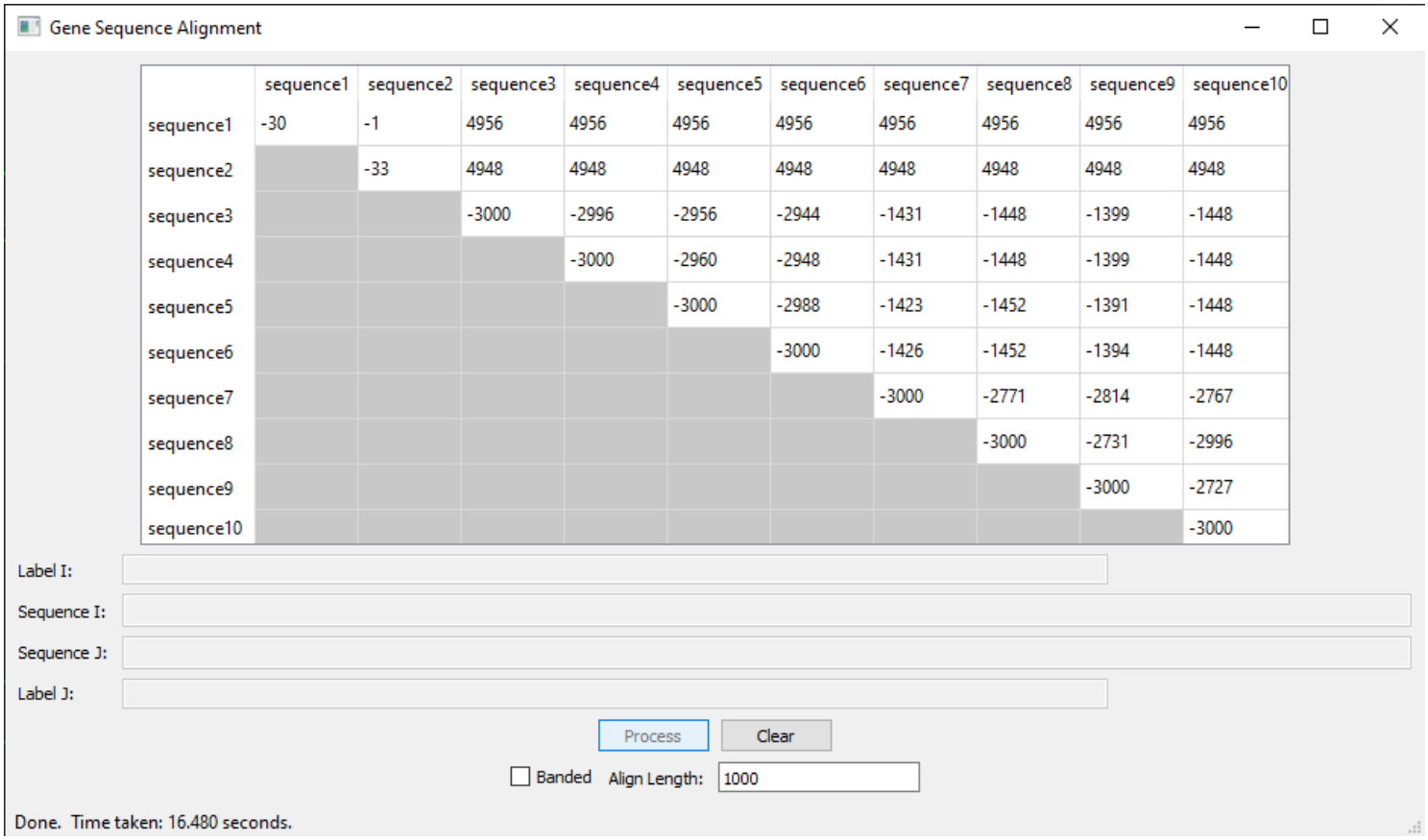
Next when traversing through the matrix, an if statement is added to check if we are too far away from the diagonal. If we are, we just skip doing any computation for that step. This ensures we only do computation for the cells along the band. So the total number of steps will be O(*kn*). The space complexity will still be O(*nm*) however because the matrix is still instantiated with the length of both sequences.

How Algorithm Works:

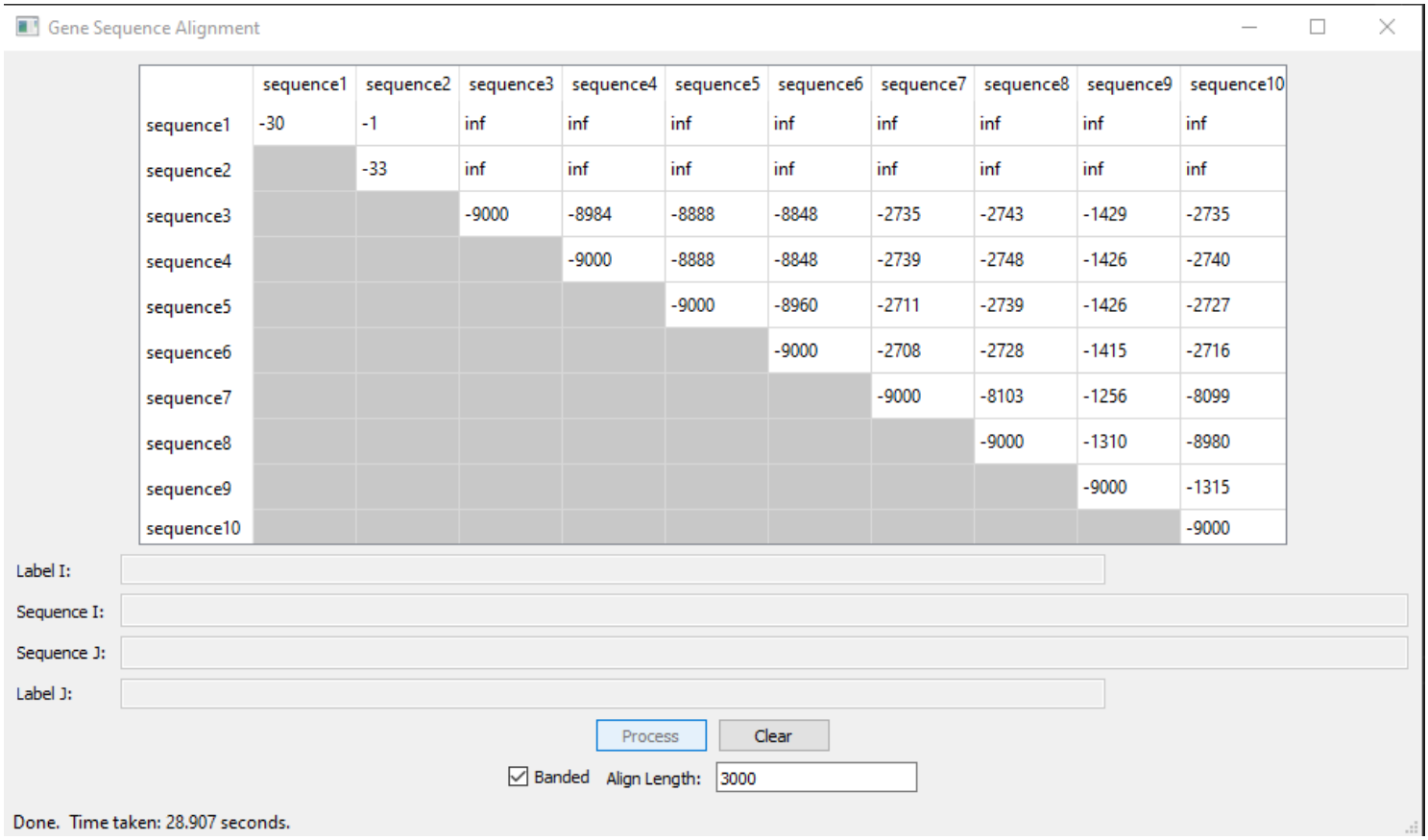
The algorithm works fairly simply. First, a matrix is instantiated with dimensions n x m. This matrix is filled with 0's as place holders. Next, the edges of the matrix are filled with the indel values. These values will always be the indel * the i or j value. Next, the matrix is filled starting from position (1,1) all the way to the bottom right corner. This is done by generating scores and determining which score is the most cost effective. A match, deletion, and insertion are calculated. Whichever score is the smallest becomes the score for the current cell. Once finished, the algorithm then begins at the bottom right corner and starts backtracking to create the alignment sequences. This is done in a while loop where the current score of the current cell is compared to the diagonal, left, and upward cell. It then checks how it arrived at the current cell by calculating how the possible score could have been reached. So for example, if the current cell's score is equal to the cell diagonal from it + the score for getting a match, then the path

must be to go diagonally because the scores align. Once finding which cell is the correct path, the 2 alignment strings are concatenated with their string value, whether that be the letter currently at the cell, or a dash for indels. Finally, the algorithm flips the alignment because we backtracked in order to find the string. Afterwards, the values are returned and the algorithm is complete.

Results:



Time Taken: 16.480 seconds
Row 3, column 4: -2996
Row 9 Column 10: -2727



Time Taken: 28.907 seconds
Row 3 Column 4: -8984
Row 9 Column 10: -1315

First 100 characters for unrestricted algorithm at row 3 column 10
These were taken from i == 2 and j == 9 with k = 1000

```
gattgcgagcgatttgcgtgcgtgcat-ccc--gcttcact-gatctcttgttagatcttttcataatctaaactttataaaaacatccactccctgt-a
-a-taagagtgattggcgtccgtacgtacccttctactctcaaactcttgttagtttaaatc-taatctaaactttat--aaac-ggcacttcctgtgt
```

First 100 characters for the banded algorithm at row 3 column 10
These were taken from i == 2 and j == 9 with k = 3000

```
gattgcgagcgatttgcgtgcgtgcat-ccc--gcttcact-gatctcttgttagatcttttcataatctaaactttataaaaacatccactccctgt-a
-a-taagagtgattggcgtccgtacgtacccttctactctcaaactcttgttagtttaaatc-taatctaaactttat--aaac-ggcacttcctgtgt
```