CS312 Gene Sequencing

Cannon Farr

1. Source Code (See Appendix)
2. Time and Space Complexity:

Unrestricted Algorithm

For the unrestricted algorithm, the time complexity is *O(nm)*. This is shown by the double for loop in the ranges of both *n* and *m*. For initializing both arrays, we also must use a double for loop but this simply adds to the co-efficient of *nm*.

```
        # Solve
        # Loop through the rows
        for i in range(1, n + 1):
            # Loop through the columns
            for j in range(1, m + 1):

                # Determine the minimum value of the left, up, and upLeft values
                arr[i][j], val = self.min_with_index(self.left(arr, i, j) +
INDEL, self.upLeft(arr, i, j) + self.cost(seq1[i - 1], seq2[j - 1]), self.up(arr,
i, j) + INDEL)

                # Determine the operation and add it to the backtrack list
                if val == 0:
                    backtrack[i][j] = ('INS',i,j)
                elif val == 1:
                    backtrack[i][j] = ('SUB',i,j)
                else:
                    backtrack[i][j] = ('DEL',i,j)
```

The space complexity of the unrestricted algorithm is similarly O(*nm)*. The values that we need to store are the costs of each path and the pointers to trace the path. These are represented in two two-dimensional arrays both of size *nm*. The cost array simply stores the integer cost and the pointer array carries a string representing the type of operation made to get that cost.

Banded Algorithm

For the banded algorithm, we limit one dimension of the matrix to a band size *k*. This follows mostly the same process as the previous example but only calculates and stores the surrounding results of the edit distance in a *kn* array.

	_	A	G	C	A	T	G	C
_	*	*	*	*				
A	*	*	*	*	*			
C	*	*	*	*	*	*		
A	*	*	*	*	*	*	*	
A		*	*	*	*	*	*	*
T			*	*	*	*	*	*
C				*	*	*	*	*
C					*	*	*	*

Doing this results in a time complexity of O(*kn)*. As shown in the code below, we iterate through *n* rows, similar how we did the unrestricted calculation. The change comes with the second for loop which is defined by a range of $range(i - band, i + band + 1)$. This for loop will always be *k* times where *k* is the size of the band. The cost comparisions are only done for the elements in the band which gives a complexity of O(*kn)*.

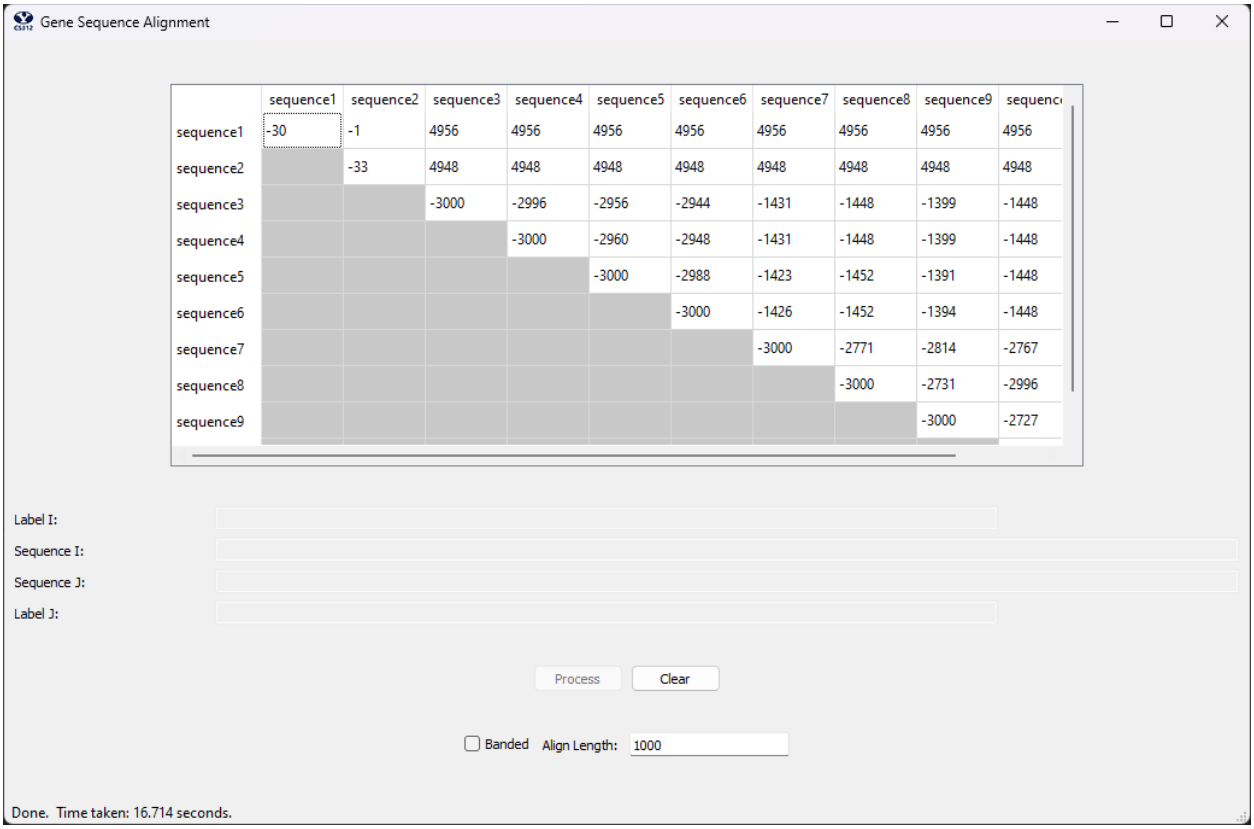
```
def bandedAlgorithm(self, seq1: str, seq2: str, n: int, m: int, band: int):
        #Initialize variables
        k = 2 * band + 1
        backtrack = []
        for i in range(n + 1):
            backtrack.append([('')] * (k)) # k + 1
        for i in range(band + 1): # k + 1
            backtrack[i][0] = ('DEL',i,0)
            backtrack[0][i] = ('INS',0,i)
        backtrack[0][0] = ('SUB',0,0)


        # Create a 2D array to store the cost of the alignment
        arr = self.initArray(n, m, band, k)

        # Solve
```

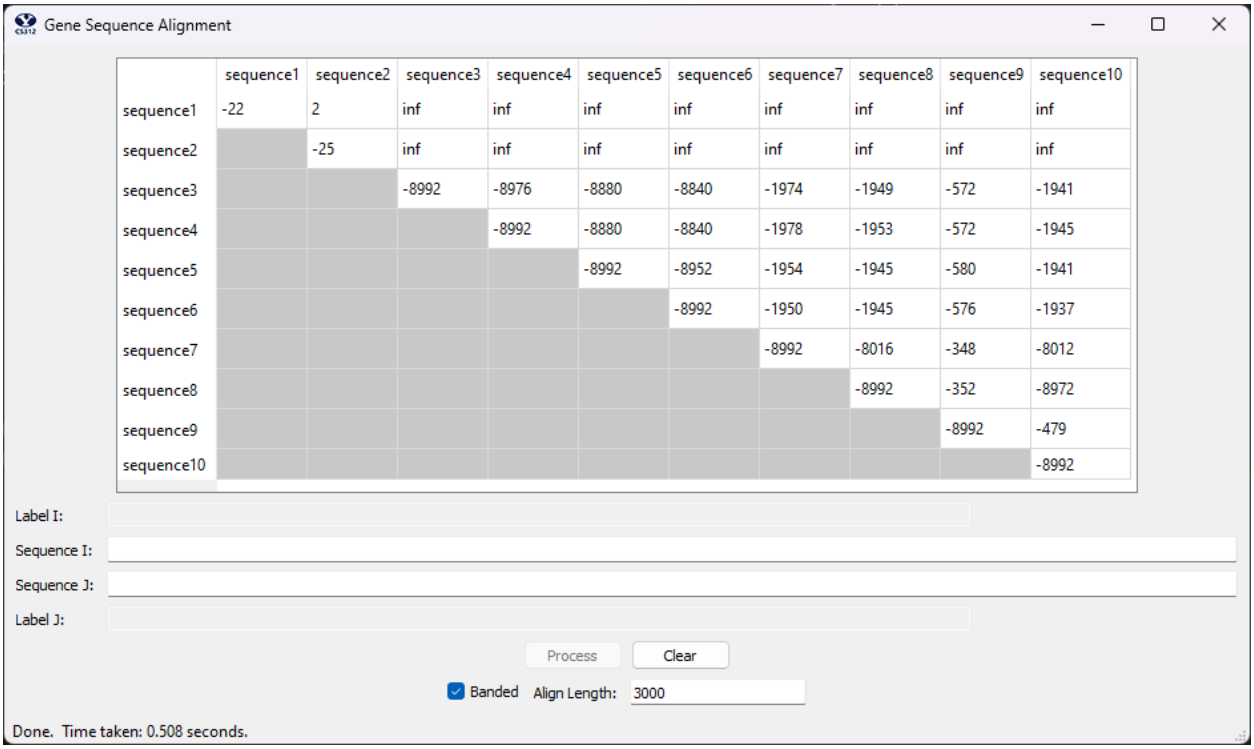
```
        # If we are reaching the end of the sequence, we need to skip some
columns
        skip = 0

        # Loop through the rows
        for i in range(1, n + 1):

            # Col is the index of the column in the n * k array wheras j is the
index of the column in the original sequence
            col = 0

            # Determine how many columns to skip based upon the current row
            if i > m - band:
                skip += 1

            # Loop through the columns
            for j in range(i - band, i + band + 1):

                # The current column we are working on is the col value + the
number of columns we have skipped
                index = col + skip

                # If we are at the beginning, then we skip the column indexes
that are less than 1
                if j < 0:
                    continue

                # If we are at the end, then we skip the column indexes that are
greater than k
                if index >= k:
                    break

                # If we are in the rows that don't have the full band, we check
the values to the left, up, and upLeft
                if i <= band or i > n - band:
                    arr[i][index], val = self.min_with_index(self.left(arr, i,
index) + INDEL, (self.upLeft(arr,i,index) + self.cost(seq1[i - 1], seq2[j - 1])),
self.up(arr, i, index) + INDEL)

                # If we are in the rows that have the full band, we check the
values to the left, up, and upRight
                else:
                    arr[i][index], val = self.min_with_index(self.left(arr, i,
index) + INDEL, (self.up(arr, i, index) + self.cost(seq1[i - 1], seq2[j - 1])),
(float('inf') if j >= k - 1 else self.upRight(arr, i, j) + INDEL))
```

```
            # Determine the operation and add it to the backtrack list
            if val == 0:
                backtrack[i][index] = ('INS',i,j)
            elif val == 1:
                backtrack[i][index] = ('SUB',i,j)
            else:
                backtrack[i][index] = ('DEL',i,j)

            # Increment the column index
            col += 1

    # Return the score and the backtrack list
    return arr[n - 1][k - 1], backtrack
```

For space complexity, we get an identical complexity of O(*kn)*. The space needed is for the size of the *kn* cost array, as well as the space for the *kn* array of back pointers. The back pointers in this array have added information tying the operation to the actual index of the sequence string. We need this because with the banded format of the *kn* array, the *k* index does not line up with the *m* index of the actual string. I have represented this information in a tuple: (‘OP’,i, j).

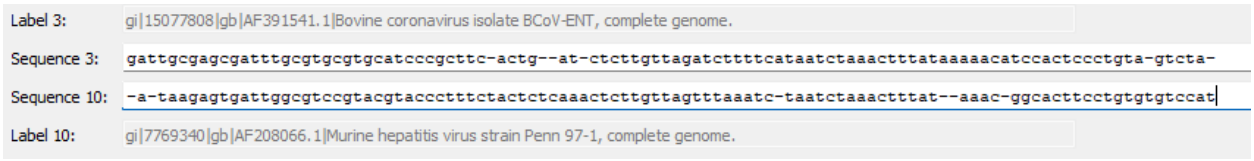
3. Screenshots:



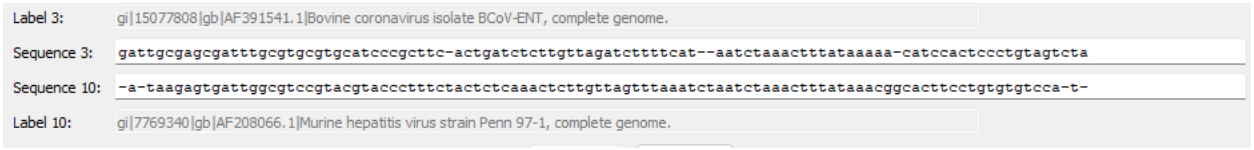


4. Characters:

Unrestricted:



Banded



Appendix:

Source Code:

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


        if len(seq1) > len(seq2):
            seq1, seq2 = seq2, seq1
        n = self.MaxCharactersToAlign if len(seq1) > self.MaxCharactersToAlign
else len(seq1)
        m = self.MaxCharactersToAlign if len(seq2) > self.MaxCharactersToAlign
else len(seq2)


        if self.banded:
            if m - n > MAXINDELS * 2 + 1:
                return {'align_cost':float('inf'), 'seqi_first100':'No Alignment
Possible.', 'seqj_first100':'No Alignment Possible.'}
            score, backtrack = self.bandedAlgorithm(seq1[:n], seq2[:m], n, m,
MAXINDELS)
            alignment1, alignment2 = self.constructAlignmentBanded(seq1[:n],
seq2[:m], n, m, MAXINDELS, backtrack)
        else:
            score, backtrack = self.unrestrictedAlgorithm(seq1[:n], seq2[:m], n,
m)
            alignment1, alignment2 =
self.constructAlignmentUnrestricted(seq1[:n], seq2[:m], n, m, backtrack)
```

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

    def unrestrictedAlgorithm(self, seq1: str, seq2: str, n: int, m: int):

        # Initialize variables
        backtrack = []
        for i in range(n + 1):
            backtrack.append([('')] * (m + 1))
            backtrack[i][0] = ('DEL',i,0)
        for i in range(m + 1):
            backtrack[0][i] = ('INS',0,i)
        backtrack[0][0] = ('SUB',0,0)

        # Create a 2D array to store the cost of the alignment
        arr = self.initArray(n, m, -1)


        # Solve
        # Loop through the rows
        for i in range(1, n + 1):
            # Loop through the columns
            for j in range(1, m + 1):

                # Determine the minimum value of the left, up, and upLeft values
                arr[i][j], val = self.min_with_index(self.left(arr, i, j) +
INDEL, self.upLeft(arr, i, j) + self.cost(seq1[i - 1], seq2[j - 1]), self.up(arr,
i, j) + INDEL)

                # Determine the operation and add it to the backtrack list
                if val == 0:
                    backtrack[i][j] = ('INS',i,j)
                elif val == 1:
                    backtrack[i][j] = ('SUB',i,j)
                else:
                    backtrack[i][j] = ('DEL',i,j)

        # Return the score and the backtrack list
        return arr[n][m], backtrack




    def bandedAlgorithm(self, seq1: str, seq2: str, n: int, m: int, band: int):
        #Initialize variables
```

```
        k = 2 * band + 1
        backtrack = []
        for i in range(n + 1):
            backtrack.append([('')] * (k)) # k + 1
        for i in range(band + 1): # k + 1
            backtrack[i][0] = ('DEL',i,0)
            backtrack[0][i] = ('INS',0,i)
        backtrack[0][0] = ('SUB',0,0)


        # Create a 2D array to store the cost of the alignment
        arr = self.initArray(n, m, band, k)

        # Solve

        # If we are reaching the end of the sequence, we need to skip some
columns
        skip = 0

        # Loop through the rows
        for i in range(1, n + 1):

            # Col is the index of the column in the n * k array wheras j is the
index of the column in the original sequence
            col = 0

            # Determine how many columns to skip based upon the current row
            if i > m - band:
                skip += 1

            # Loop through the columns
            for j in range(i - band, i + band + 1):

                # The current column we are working on is the col value + the
number of columns we have skipped
                index = col + skip

                # If we are at the beginning, then we skip the column indexes
that are less than 1
                if j < 0:
                    continue

                # If we are at the end, then we skip the column indexes that are
greater than k
                if index >= k:
```

```
                    break

                # If we are in the rows that don't have the full band, we check
the values to the left, up, and upLeft
                if i <= band or i > n - band:
                    arr[i][index], val = self.min_with_index(self.left(arr, i,
index) + INDEL, (self.upLeft(arr,i,index) + self.cost(seq1[i - 1], seq2[j - 1])),
self.up(arr, i, index) + INDEL)

                # If we are in the rows that have the full band, we check the
values to the left, up, and upRight
                else:
                    arr[i][index], val = self.min_with_index(self.left(arr, i,
index) + INDEL, (self.up(arr, i, index) + self.cost(seq1[i - 1], seq2[j - 1])),
(float('inf') if j >= k - 1 else self.upRight(arr, i, j) + INDEL))

                # Determine the operation and add it to the backtrack list
                if val == 0:
                    backtrack[i][index] = ('INS',i,j)
                elif val == 1:
                    backtrack[i][index] = ('SUB',i,j)
                else:
                    backtrack[i][index] = ('DEL',i,j)

                # Increment the column index
                col += 1

        # Return the score and the backtrack list
        return arr[n - 1][k - 1], backtrack









    def initArray(self, n: int, m: int, band: int, k: int = 0):
        arr = []
        if band != -1:

            #initialize the first row and column for banded
            for i in range(n + 1):
                arr.append([float('inf')] * (k))
            arr[0][0] = 0
```

```
            for i in range(band + 1):
            #   arr[i][0] = i * INDEL
                arr[0][i] = i * INDEL

        else:
            #initialize the first row and column for unbanded
            for i in range(n + 1):
                arr.append([0] * (m + 1))
                arr[i][0] = i * INDEL

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

        return arr


    def constructAlignmentBanded(self, seq1: str, seq2: str, n: int, m: int,
band: int, backtrack: list[tuple[str, int, int]]):
        i = n
        k = (2 * band + 1) - 1
        cur = backtrack[i][k]
        alignment1 = ''
        alignment2 = ''
        while i > 0 or k > 0:
            if cur[0] == 'INS':
                alignment1 = '-' + alignment1
                alignment2 = seq2[cur[2] - 1] + alignment2
                k -= 1
            elif cur[0] == 'DEL':
                alignment1 = seq1[cur[1] - 1] + alignment1
                alignment2 = '-' + alignment2
                if i <= band or i > n - band:
                    i -= 1
                else:
                    i -= 1
                    k += 1
            else:
                alignment1 = seq1[cur[1] - 1] + alignment1
                alignment2 = seq2[cur[2] - 1] + alignment2
                if i <= band or i > n - band:
                    i -= 1
                    k -= 1
                else:
                    i -= 1
            cur = backtrack[i][k]
```

```
        return alignment1, alignment2


    def constructAlignmentUnrestricted(self, seq1: str, seq2: str, n: int, m:
int, backtrack: list[tuple[str, int, int]]):
        i = n
        j = m
        cur = backtrack[i][j][0]
        alignment1 = ''
        alignment2 = ''
        while i > 0 or j > 0:
            if cur == 'INS':
                alignment1 = '-' + alignment1
                alignment2 = seq2[j - 1] + alignment2
                j -= 1
            elif cur == 'DEL':
                alignment1 = seq1[i - 1] + alignment1
                alignment2 = '-' + alignment2
                i -= 1
            else:
                alignment1 = seq1[i - 1] + alignment1
                alignment2 = seq2[j - 1] + alignment2
                i -= 1
                j -= 1
            cur = backtrack[i][j][0]
        return alignment1, alignment2




    def left(self, arr: list, i: int, j: int):
        return arr[i][j - 1]

    def up(self, arr: list, i: int, j: int):
        return arr[i - 1][j]

    def upLeft(self, arr: list, i: int, j: int):
        return arr[i - 1][j - 1]

    def upRight(self, arr: list, i: int, j: int):
        return arr[i - 1][j + 1]

    def cost(self, a: str, b: str):
        if a == b:
            return MATCH
```

```
        else:
            return SUB

    def min_with_index(self, *args):
        # Check if there are any arguments provided
        if not args:
            raise ValueError("min_with_index() arg is an empty sequence")

        # Find the minimum value among the arguments
        min_value = min(args)

        # Find the index of the minimum value
        # Note: The first occurrence of the minimum value is returned in case of
duplicates
        min_index = args.index(min_value)

        # Return both the minimum value and its index (position among the
arguments)
        return min_value, min_index
```