Exploration of the Dynamic Programming Algorithm

Through Gene Sequencing

Katherine Wilsdon

Idaho State University

Universal Classes

Node

The node class was created to hold information such as the edit distance, previous ith term, and the previous jth term, shown in Figure 1.

```
class Node:

    def __init__(self):
        self.edit_dist = float("inf")
        self.prev_i = float("inf")
        self.prev_j = float("inf")
```

Figure 1: Node Class Implementation.

Unrestricted Algorithm

Edit Distance

Figure 4 shows that a 2D array is initialized with dimensions m+1 by n+1. The algorithm contains two for loops of range [0 to m] and [0 to n]. When i = 0 or j = 0, all the characters from the other sequence are inserted because the current sequence is empty / contains a gap and the previous pointer is set to the left / top node. If the character of both sequences match, the match cost is added to the diagonal node's edit distance and the previous pointer is set to the diagonal node. If the current character of both sequences is different, the minimum is found between the left, top, or diagonal node's edit distance plus either the insertion, deletion, or substitution cost and the previous pointer is set to that associated node.

```
class Unrestricted:

    def __init__(self):
        self.indel = 5  # Insertion / Deletion cost
        self.sub = 1  # Substitution cost
        self.match = -3  # Match cost
```

Figure 2: Unrestricted Class Implementation.

```
function unrestricted_edit_distance(seq1, seq2, m, n)
    create an array of Nodes, m+1 by n+1
    for i from 0 to m
        for j from 0 to n
            if i == 0
                array[i,j].edit_dist = j + indel cost
                set previous pointer
            else if j == 0
                array[i,j].edit_dist = i + indel cost
                set previous pointer
            else if if seq1[i] == seq2[j]
                array[i,j].edit_dist = diagonal node edit dist + match cost
                set previous pointer
            else
                array[i,j].edit_dist = min (left node edit dist + indel cost,
                                            top node edit dist + indel cost,
                                            diagonal node edit dist + sub cost)
                set previous pointer
```

Figure 3: Edit Distance Method for the Unrestricted Algorithm Pseudocode.

```python
def edit_distance(self, seq1, seq2, m, n):
    # Initialize 2D array m + 1 by n + 1
    self.memo = [[Node() for i in range(n + 1)] for j in range(m + 1)]

    for i in range(m + 1):
        for j in range(n + 1):
            # i = 0 indicates a gap/an empty first sequence. All characters from the second sequence are inserted
            if i == 0 and j == 0:
                self.memo[i][j].edit_dist = j * self.indel
            elif i == 0:
                self.memo[i][j].edit_dist = j * self.indel
                # Assign the previous pointer
                self.memo[i][j].prev_i = i                    # Insertion
                self.memo[i][j].prev_j = j - 1
            # j = 0 indicates a gap/an empty second sequence. All characters from the first sequence are inserted
            elif j == 0:
                self.memo[i][j].edit_dist = i * self.indel
                # Assign the previous pointer
                self.memo[i][j].prev_i = i - 1                # Deletion
                self.memo[i][j].prev_j = j
            # If the current character in each sequence match
            elif seq1[i - 1] == seq2[j - 1]:
                self.memo[i][j].edit_dist = self.memo[i - 1][j - 1].edit_dist + self.match
                # Assign the previous pointer
                self.memo[i][j].prev_i = i - 1                # Match
                self.memo[i][j].prev_j = j - 1
            # If the current character in each sequence are different, find the minimum
            else:
                self.memo[i][j].edit_dist = min(self.memo[i][j - 1].edit_dist + self.indel,     # Insertion
                                                self.memo[i - 1][j].edit_dist + self.indel,     # Deletion
                                                self.memo[i - 1][j - 1].edit_dist + self.sub)   # Substitution

                # Assign the previous pointer
                if self.memo[i][j].edit_dist == self.memo[i][j - 1].edit_dist + self.indel:     # Insertion
                    self.memo[i][j].prev_i = i
                    self.memo[i][j].prev_j = j - 1
                elif self.memo[i][j].edit_dist == self.memo[i - 1][j].edit_dist + self.indel:   # Deletion
                    self.memo[i][j].prev_i = i - 1
                    self.memo[i][j].prev_j = j
                elif self.memo[i][j].edit_dist == self.memo[i - 1][j - 1].edit_dist + self.sub:  # Substitution
                    self.memo[i][j].prev_i = i - 1
                    self.memo[i][j].prev_j = j - 1
```

Figure 4: Edit Distance Method for the Unrestricted Algorithm.

Edit Distance – Alignment Extraction

Figure 6 shows the back-trace for the alignment extraction for the unrestricted algorithm. i and j are initialized to the size of m (length of sequence 1) and the size n (length of sequence 2). The algorithm contains a while loop when i or j does not equal infinity. Next the previous node is checked for being the diagonal, left, or top node. If the previous is the diagonal node, insert both characters into alignment 1 and alignment 2. If the previous node is the left node, a gap is inserted into alignment 1 while the character of sequence 2 is inserted into alignment 2. If the previous node is the top node, a gap is inserted into alignment 2 while the character of sequence 1 is inserted into alignment 1. i and j are set the previous pointer, and the while loop continues. The edit distance, alignment 1, and alignment 2 are returned.

```
function unrestricted_alignment_extraction(array, seq1, seq2, m, n)
    create array of characters, alignment1
    create array of characters, alignment2
    i = m
    j = n
    while i != inf or j != inf
        if previous node is the diagonal node
            alignment1.insert(seq1[i])
            alignment2.insert(seq1[j])
        else if previous node is the left node
            alignment1.insert('-')
            alignment2.insert(seq1[j])
        else if previous node is the top node
            alignment1.insert(seq1[i])
            alignment2.insert('-')
        set i and j to previous pointer
    return array[m,n].edit_dist, alignment1, alignment2
```

Figure 5: Edit Distance Method - Alignment Extraction for the Unrestricted Algorithm Pseudocode.

```
alignment1 = []
alignment2 = []
i = m
j = n
while i != float("inf") or j != float("inf"):
    # If the previous node is the diagonal node (match or substitution) insert the character into each alignment
    if self.memo[i][j].prev_i == i - 1 and self.memo[i][j].prev_j == j - 1:
        alignment1.insert(0, seq1[i-1])
        alignment2.insert(0, seq2[j-1])
    # If the previous node is the left node (insertion) insert a gap into alignment 1
    elif self.memo[i][j].prev_i == i and self.memo[i][j].prev_j == j - 1:
        alignment1.insert(0, '-')
        alignment2.insert(0, seq2[j-1])
    # If the previous node is the top node (deletion) insert a gap into alignment 2
    elif self.memo[i][j].prev_i == i - 1 and self.memo[i][j].prev_j == j:
        alignment1.insert(0, seq1[i-1])
        alignment2.insert(0, '-')
    i, j = self.memo[i][j].prev_i, self.memo[i][j].prev_j
return self.memo[m][n].edit_dist, "".join(alignment1), "".join(alignment2)
```

Figure 6: Edit Distance Method - Alignment Extraction for the Unrestricted Algorithm.

Banded Algorithm

Edit Distance

In Figure 9, a 2D array of n+1 by k (bandwidth) is initialized with n being the smaller sequence between the two sequences. The following code is divided into 3 sections: Front, Middle, Back. The front section contains two for loops of range [0, d+1) where d+1 is half of the bandwidth and [0, stop_j) where stop_j is the stopping point for the jth term that incremented before i is incremented until reaching the total length of the bandwidth. The calculate_edit_ distance( ) function is called which is the same code as the unrestricted algorithm. The variable, start_bigger_seq, keeps track of the bigger sequence's current character since the array is restricted to the bandwidth size. The middle section has two for loops of range [d+1, len(small_seq) – end_i + 1) where d+1 is the next index after reaching the full bandwidth, end_i is the position of i when the last position of the bigger sequence has been reached, and +1 to make end_i inclusive and range [0,bandwidth). The calculate_mid_edit_distance( ) function is

called which is a modified version of edit distance because of restricting the jth term to the bandwidth size. Lastly, the back section has two for loops of range[len(smaller_seq) – end_i + 1, len(smaller_seq) + 1) and [start_j, bandwidth) where start_j is the starting point for the jth term that is incremented before i is incremented. The calculate_edit_distance( ) function is called which is the same code as the unrestricted algorithm.

```
class Banded:

    def __init__(self):
        self.indel = 5          # Insertion / Deletion cost
        self.sub = 1            # Substitution cost
        self.match = -3         # Match cost
        self.memo = []          # 2D array
```

Figure 7: Banded Algorithm Implementation.

```
function banded_edit_distance(seq1, seq2, m, n, d)
    k = 2d+1 # bandwidth
    create array of Nodes, n+1 by k where n is the length of the smaller sequence
    # Front Section
    stop = d+1
    for i from 0 to d+1
        for j from 0 to stop
            calcuate_edit_distance(seq1, seq2, i, j, j)
        stop += 1
    # Middle Section
    start_bigger_seq = 1
    for i from d+1 to len(array) - k/2
        for j from 0 to k-1
            calculate_mid_edit_distance(seq1, seq2, i, j, start_bigger_seq+j)
        start_bigger_seq += 1
    # Back Section
    start = 1
    for i from len(array) - k/2 to len(array)
        for j from start to k-1
            calcuate_edit_distance(seq1, seq2, i, j, j)
        start += 1
```

Figure 8: Edit Distance Part 1 of the Banded Algorithm Pseudocode.

```
def edit_distance(self, seq1, seq2, m, n, d):
    self.bandwidth = 2 * d + 1
    # Lengths of the smaller and bigger sequences
    smaller_seq = 0
    bigger_seq = 0
    # The position of i when the last position of bigger sequence has been reached
    end_i = 0
    # Initialize 2D array n + 1 by k (bandwidth), smaller sequence by bigger sequence
    # When the length of sequence 1 is smaller than sequence 2, create an m + 1 by k array
    if m < n:
        self.memo = [[Node() for i in range(self.bandwidth)] for j in range(m+1)]
        smaller_seq = m
        bigger_seq = n
        end_i = d - 1
    # When the length of sequence 2 is smaller than sequence 1, create an n + 1 by k array
    elif n < m:
        self.memo = [[Node() for i in range(self.bandwidth)] for j in range(n+1)]
        smaller_seq = n
        bigger_seq = m
    # When the length of sequence 1 equals than sequence 2, create an m + 1 by k array
    elif m == n:
        self.memo = [[Node() for i in range(self.bandwidth)] for j in range(m+1)]
        smaller_seq = m
        bigger_seq = n
        end_i = d

    # Stopping point for jth term that is incremented before i is incremented
    stop_j = d + 1
    # Front Section: Calculates the edit distance similarly to unrestricted
    for i in range(0, d + 1):
        for j in range(0, stop_j):
            self.calculate_edit_distance(seq1, seq2, i, j, j)
        stop_j += 1

    # Variable that keeps track of the bigger sequence's current character
    start_bigger_seq = 1
    # Middle Section: Calculates a version of edit distance because of restricting jth term to the bandwidth size
    for i in range(d + 1, smaller_seq - end_i + 1):
        for j in range(self.bandwidth):
            self.calculate_mid_edit_distance(seq1, seq2, i, j, start_bigger_seq+j)
        start_bigger_seq += 1

    # Starting point for jth term that is incremented before i is incremented
    start_j = 1
    # Decrement the variable because the jth term account for the incremental change
    start_bigger_seq -= 1
    # Back Section: Calculates the edit distance similarly to unrestricted
    for i in range(smaller_seq - end_i + 1, smaller_seq + 1):
        for j in range(start_j, self.bandwidth):
            self.calculate_edit_distance(seq1, seq2, i, j, start_bigger_seq + j)
        start_j += 1
```

Figure 9: Edit Distance Part 1 of the Banded Algorithm.

Edit Distance – Alignment Extraction

Figure 11 shows the back-trace for the alignment extraction for the banded algorithm. i and j are initialized to the size of m (length of sequence 1) and the bandwidth size. The variable, bigger_seq_j, keeps track of the bigger sequence's current character. The algorithm contains a while loop when i or j does not equal infinity. The while loop has two sections, one for the front and back sections and the other for the middle section. This is required because the indices for the substitution/match, insertion, and deletion differ between the front/back section and middle section. Foreach of the two sections, the previous node is checked for being the diagonal, left, or top node. If the previous is the diagonal node, insert both characters into alignment 1 and alignment 2. If the previous node is the left node, a gap is inserted into alignment 1 while the character of sequence 2 is inserted into alignment 2. If the previous node is the top node, a gap is inserted into alignment 2 while the character of sequence 1 is inserted into alignment 1. i and j are set the previous pointer, and the while loop continues. The edit distance, alignment 1, and alignment 2 are returned.

```
function banded_alignment_extraction(array, seq1, seq2, m, n)
    create array of characters, alignment1
    create array of characters, alignment2
    i = m
    j = n
    while i != inf or j != inf
        if i < d+1 or i > len(array) - k/2
            if previous node is the diagonal node
                alignment1.insert(seq1[i])
                alignment2.insert(seq1[j])
            else if previous node is the left node
                alignment1.insert('-')
                alignment2.insert(seq1[j])
            else if previous node is the top node
                alignment1.insert(seq1[i])
                alignment2.insert('-')
            set i and j to previous pointer
        else
            if previous node is the diagonal node
                alignment1.insert(seq1[i])
                alignment2.insert(seq1[j])
            else if previous node is the left node
                alignment1.insert('-')
                alignment2.insert(seq1[j])
            else if previous node is the top node
                alignment1.insert(seq1[i])
                alignment2.insert('-')
            set i and j to previous pointer
    return array[m,n].edit_dist, alignment1, alignment2
```

Figure 10: Edit Distance Part 2 – Alignment Extraction of the Banded Algorithm Pseudocode.

```
alignment1 = []
alignment2 = []
# The nth term in the n + 1 by k array
i = smaller_seq
# Variable that keeps track of the bigger sequence's current character
bigger_seq_j = bigger_seq
# The kth term in the n + 1 by k array
j = self.bandwidth - 1
while i != float("inf") or j != float("inf"):
    # If i is in the front or back section
    if i < d + 1 or i > smaller_seq - end_i:
        # If the previous node is the diagonal node (match/substitution) insert the char into each alignment
        if self.memo[i][j].prev_i == i - 1 and self.memo[i][j].prev_j == j - 1:     # Substitution / Match
            alignment1.insert(0, seq1[i - 1])
            alignment2.insert(0, seq2[bigger_seq_j - 1])
            bigger_seq_j -= 1
        # If the previous node is the left node (insertion) insert a gap into alignment 1
        elif self.memo[i][j].prev_i == i and self.memo[i][j].prev_j == j - 1:       # Insertion
            alignment1.insert(0, '-')
            alignment2.insert(0, seq2[bigger_seq_j - 1])
            bigger_seq_j -= 1
        # If the previous node is the top node (deletion) insert a gap into alignment 2
        elif self.memo[i][j].prev_i == i - 1 and self.memo[i][j].prev_j == j:       # Deletion
            alignment1.insert(0, seq1[i - 1])
            alignment2.insert(0, '-')
        i, j = self.memo[i][j].prev_i, self.memo[i][j].prev_j
    # If i is in the middle section
    else:
        # If the previous node is the diagonal node (match/substitution) insert the char into each alignment
        if self.memo[i][j].prev_i == i - 1 and self.memo[i][j].prev_j == j:         # Substitution / Match
            alignment1.insert(0, seq1[i - 1])
            alignment2.insert(0, seq2[bigger_seq_j - 1])
            bigger_seq_j -= 1
        # If the previous node is the left node (insertion) insert a gap into alignment 1
        elif self.memo[i][j].prev_i == i and self.memo[i][j].prev_j == j - 1:       # Insertion
            alignment1.insert(0, '-')
            alignment2.insert(0, seq2[bigger_seq_j - 1])
            bigger_seq_j -= 1
        # If the previous node is the top node (deletion) insert a gap into alignment 2
        elif self.memo[i][j].prev_i == i - 1 and self.memo[i][j].prev_j == j + 1:   # Deletion
            alignment1.insert(0, seq1[i - 1])
            alignment2.insert(0, '-')
        i, j = self.memo[i][j].prev_i, self.memo[i][j].prev_j
return self.memo[smaller_seq][self.bandwidth-1].edit_dist, "".join(alignment1), "".join(alignment2)
```

Figure 11: Edit Distance Part 2 – Alignment Extraction of the Banded Algorithm.

Calculate Front and Back Section's Edit Distance

Figure 12 shows the calculation of the front and back section's edit distance for the banded algorithm. See the Edit Distance section for the Unrestricted Algorithm for further details since the algorithm is similar. The only difference is the parameter, bigger_seq_j, which keeps track of the bigger sequence's current character.

```
# Calculates the edit distance for the front and back section similarly to the unrestricted algorithm
def calculate_edit_distance(self, seq1, seq2, i, j, bigger_seq_j):
    # i = 0 indicates a gap/an empty first sequence. All characters from the second sequence are inserted
    if i == 0 and j == 0:
        self.memo[i][j].edit_dist = j * self.indel
    elif i == 0:
        self.memo[i][j].edit_dist = j * self.indel
        # Assign the previous pointer
        self.memo[i][j].prev_i = i  # Insertion
        self.memo[i][j].prev_j = j - 1
    # j = 0 indicates a gap/an empty second sequence. All characters from the first sequence are inserted
    elif j == 0:
        self.memo[i][j].edit_dist = i * self.indel
        # Assign the previous pointer
        self.memo[i][j].prev_i = i - 1  # Deletion
        self.memo[i][j].prev_j = j
    # If the current character in each sequence match
    elif seq1[i - 1] == seq2[bigger_seq_j - 1]:
        self.memo[i][j].edit_dist = self.memo[i - 1][j - 1].edit_dist + self.match
        # Assign the previous pointer
        self.memo[i][j].prev_i = i - 1  # Match
        self.memo[i][j].prev_j = j - 1
    # If the current character in each sequence are different, find the minimum
    else:
        self.memo[i][j].edit_dist = min(self.memo[i][j - 1].edit_dist + self.indel,  # Insertion
                                        self.memo[i - 1][j].edit_dist + self.indel,  # Deletion
                                        self.memo[i - 1][j - 1].edit_dist + self.sub)  # Substitution

        # Assign the previous pointer
        if self.memo[i][j].edit_dist == self.memo[i][j - 1].edit_dist + self.indel:  # Insertion
            self.memo[i][j].prev_i = i
            self.memo[i][j].prev_j = j - 1
        elif self.memo[i][j].edit_dist == self.memo[i - 1][j].edit_dist + self.indel:  # Deletion
            self.memo[i][j].prev_i = i - 1
            self.memo[i][j].prev_j = j
        elif self.memo[i][j].edit_dist == self.memo[i - 1][j - 1].edit_dist + self.sub:  # Substitution
            self.memo[i][j].prev_i = i - 1
            self.memo[i][j].prev_j = j - 1
```

Figure 12: Calculate Edit Distance for the Front and Back Sections of the Banded Algorithm.

Calculate Middle Section's Edit Distance

Figure 14 shows the calculation of the middle section's edit distance for the banded algorithm. If the character of both sequences match, the match cost is added to the diagonal node's edit distance and the previous pointer is set to the diagonal node. When j = 0, the minimum is found between the top or diagonal node's edit distance plus either the deletion or substitution cost, and the previous pointer is set to that associated node. If j = bandwidth -1, the minimum is found between the left or diagonal node's edit distance plus either the insertion or substitution cost, and the previous pointer is set to that associated node. If not any of these special cases, the minimum is found between the left, top, or diagonal node's edit distance plus either the insertion, deletion, or substitution cost, and the previous pointer is set to that associated node. Note: The indices for the substitution/match, insertion, and deletion differ between the front/back section and middle section due to a fixed bandwidth size.

```
function banded_calculate_mid_edit_distance(array, seq1, seq2, m, n, bigger_seq_j)
    if if seq1[i] == seq2[j]
        array[i,j].edit_dist = diagonal node edit dist + match cost
        set previous pointer
    else if j == 0
        array[i,j].edit_dist = min (top node edit dist + indel cost,
                                    diagonal node edit dist + sub cost)
        set previous pointer
    else if j == len(array[0]):
        array[i,j].edit_dist = min (left node edit dist + indel cost,
                                    diagonal node edit dist + sub cost)
        set previous pointer
    else
        array[i,j].edit_dist = min (left node edit dist + indel cost,
                                    top node edit dist + indel cost,
                                    diagonal node edit dist + sub cost)
        set previous pointer
```

Figure 13: Calculate Edit Distance for the Middle Section of the Banded Algorithm Pseudocode.

```
def calculate_mid_edit_distance(self, seq1, seq2, i, j, bigger_seq_j):
    # If the current character in each sequence match
    if seq1[i - 1] == seq2[bigger_seq_j - 1]:
        self.memo[i][j].edit_dist = self.memo[i-1][j].edit_dist + self.match
        # Assign the previous pointer
        self.memo[i][j].prev_i = i - 1
        self.memo[i][j].prev_j = j
    # If the jth term is the first index, find the minimum of deletion and substitution (no insertion)
    elif j == 0:
        self.memo[i][j].edit_dist = min(self.memo[i-1][j+1].edit_dist + self.indel,     # Deletion
                                        self.memo[i-1][j].edit_dist + self.sub)         # Substitution
        # Assign the previous pointer
        if self.memo[i][j].edit_dist == self.memo[i-1][j+1].edit_dist + self.indel:     # Deletion
            self.memo[i][j].prev_i = i - 1
            self.memo[i][j].prev_j = j + 1
        elif self.memo[i][j].edit_dist == self.memo[i-1][j].edit_dist + self.sub:       # Substitution
            self.memo[i][j].prev_i = i - 1
            self.memo[i][j].prev_j = j
    # If the jth term is the last index, find the minimum of substitution and insertion (no deletion)
    elif j == self.bandwidth - 1:
        self.memo[i][j].edit_dist = min(self.memo[i][j-1].edit_dist + self.indel,       # Insertion
                                        self.memo[i-1][j].edit_dist + self.sub)         # Substitution
        # Assign the previous pointer
        if self.memo[i][j].edit_dist == self.memo[i][j-1].edit_dist + self.indel:       # Insertion
            self.memo[i][j].prev_i = i
            self.memo[i][j].prev_j = j - 1
        elif self.memo[i][j].edit_dist == self.memo[i-1][j].edit_dist + self.sub:       # Substitution
            self.memo[i][j].prev_i = i - 1
            self.memo[i][j].prev_j = j
    # If not a special case of the first or last index, find the minimum
    else:
        self.memo[i][j].edit_dist = min(self.memo[i][j-1].edit_dist + self.indel,       # Insertion
                                        self.memo[i-1][j+1].edit_dist + self.indel,     # Deletion
                                        self.memo[i-1][j].edit_dist + self.sub)         # Substitution

        # Assign  the previous pointer
        if self.memo[i][j].edit_dist == self.memo[i][j-1].edit_dist + self.indel:       # Insertion
            self.memo[i][j].prev_i = i
            self.memo[i][j].prev_j = j - 1
        elif self.memo[i][j].edit_dist == self.memo[i-1][j+1].edit_dist + self.indel:   # Deletion
            self.memo[i][j].prev_i = i - 1
            self.memo[i][j].prev_j = j + 1
        elif self.memo[i][j].edit_dist == self.memo[i-1][j].edit_dist + self.sub:       # Substitution
            self.memo[i][j].prev_i = i - 1
            self.memo[i][j].prev_j = j
```

Figure 14: Calculate Edit Distance for the Middle Section of the Banded Algorithm.

Gene Sequencing Algorithm

Figure 15 depicts an algorithm that calculates the edit distance, alignment 1, and alignment 2 between two sequences of a matrix of sequences 1-10 by sequences 1-10. Only the sequences along the diagonal and above the diagonal are calculated since calculations are repeated below the diagonal. The code is split into two sections depending on whether the banded checkbox is checked or not. When a test sequence (i=0, i=1) is paired with a Coronavirus sequence (j>1), return “No Alignment Possible.” If the provided characters to align is less than the 2 sequence lengths, (i.e. not a test sequence), the edit_distance( ) function is called on either the banded_alg or unrestricted_alg objects with the size of the sequences provided by the user, MaxCharactersToAlign. Otherwise, if the sequences are either of the test sequences (polynomial or exponential), the sizes of the sequences are set to the length of the first and second sequences and passed in as parameters into the edit_distance( ) function.

```
def align( self, sequences, table, banded, align_length):
    self.banded = banded
    self.MaxCharactersToAlign = align_length
    self.d = 3
    results = []

    for i in range(len(sequences)):
        jresults = []
        for j in range(len(sequences)):

            if(j < i):
                s = {}
            else:
                # If the banded checkbox is checked, run the banded algorithm
                if self.banded:
                    banded_alg = Banded()
                    # If the test sequence is paired with a Coronavirus sequence, report no alignment possible
                    if (i == 0 or i == 1) and j > 1:
                        score, alignment1, alignment2 = float('inf'), "No Alignment Possible", "No Alignment Possible"
                    # If the provided characters to align is less than the 2 sequence lengths (i.e. not a test case)
                    elif self.MaxCharactersToAlign < len(sequences[i]) and self.MaxCharactersToAlign < len(sequences[j]):
                        score, alignment1, alignment2 = banded_alg.edit_distance(sequences[i], sequences[j],
                            self.MaxCharactersToAlign, self.MaxCharactersToAlign,  self.d)
                    # If either the test cases of polynomial or exponential
                    else:
                        m, n = len(sequences[i]), len(sequences[j])
                        score, alignment1, alignment2 = banded_alg.edit_distance(sequences[i], sequences[j],
                            m, n, self.d)
                # If the banded checkbox is not checked, run the unrestricted algorithm
                else:
                    unrestricted_alg = Unrestricted()
                    # If the test sequence is paired with a Coronavirus sequence, report no alignment possible
                    if (i == 0 or i == 1) and j > 1:
                        score, alignment1, alignment2 = float('inf'), "No Alignment Possible", "No Alignment Possible"
                    # If the provided characters to align is less than the 2 sequence lengths (i.e. not a test case)
                    elif self.MaxCharactersToAlign < len(sequences[i]) and self.MaxCharactersToAlign < len(
                            sequences[j]):
                        score, alignment1, alignment2 = unrestricted_alg.edit_distance(sequences[i], sequences[j],
                                                                                       self.MaxCharactersToAlign,
                                                                                       self.MaxCharactersToAlign)
                    # If either the test cases of polynomial or exponential
                    else:
                        m, n = len(sequences[i]), len(sequences[j])
                        score, alignment1, alignment2 = unrestricted_alg.edit_distance(sequences[i], sequences[j],
                                                                                       m, n)
                s = {'align_cost':score, 'seqi_first100':alignment1[0:100], 'seqj_first100':alignment2[0:100]}
                table.item(i,j).setText('{}'.format(int(score) if score != math.inf else score))
                table.update()
            jresults.append(s)
        results.append(jresults)
    return results
```

Figure 15: Align Function in the Gene Sequencing Class.

Results

Unrestricted Algorithm

The unrestricted algorithm took 81.577 seconds which is 38.4 seconds faster or 1.47 times faster than the performance requirement of 120 seconds, Figure 16. The edit distance between sequence 3 and sequence 4 is -2996. The edit distance between sequence 9 and sequence 10 is -2727. These numbers validate the correctness of the unrestricted algorithm.

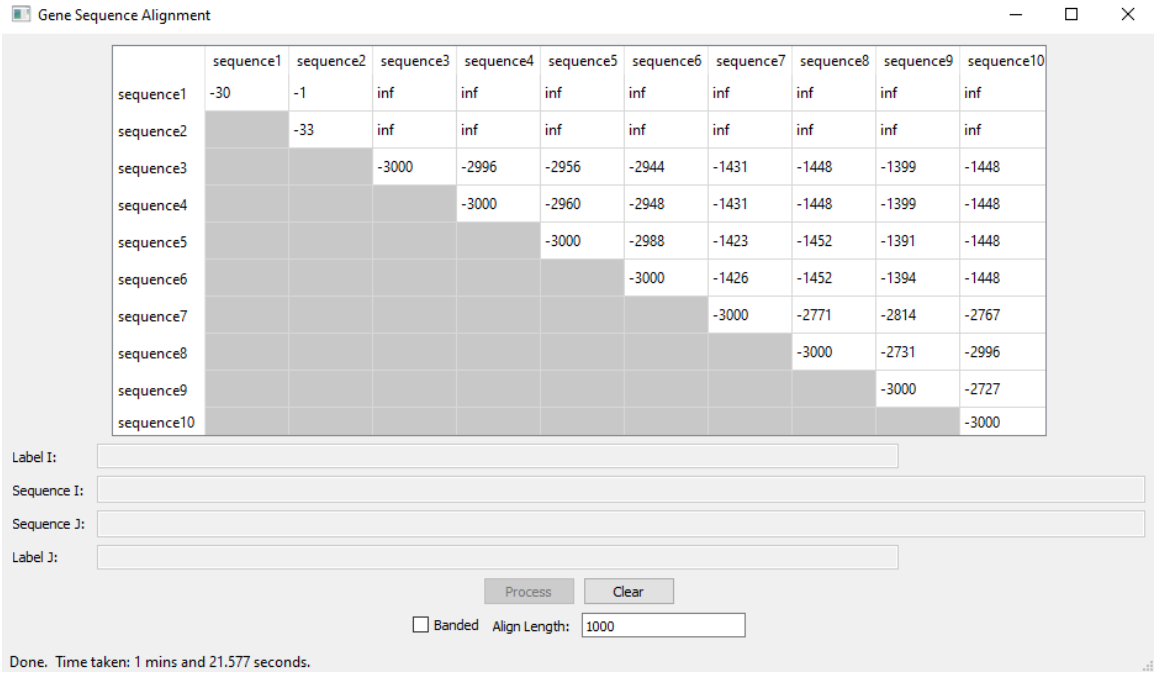

Figure 16: Unrestricted Algorithm with an Alignment Length of 1,000.

Banded Algorithm

The banded algorithm took 1.970 seconds which is 8.03 seconds faster or 5.08 times faster than the performance requirement of 10 seconds, Figure 17. The edit distance between sequence 3 and sequence 4 is -8984. The edit distance between sequence 9 and sequence 10 is -1315. These numbers validate the correctness of the banded algorithm.

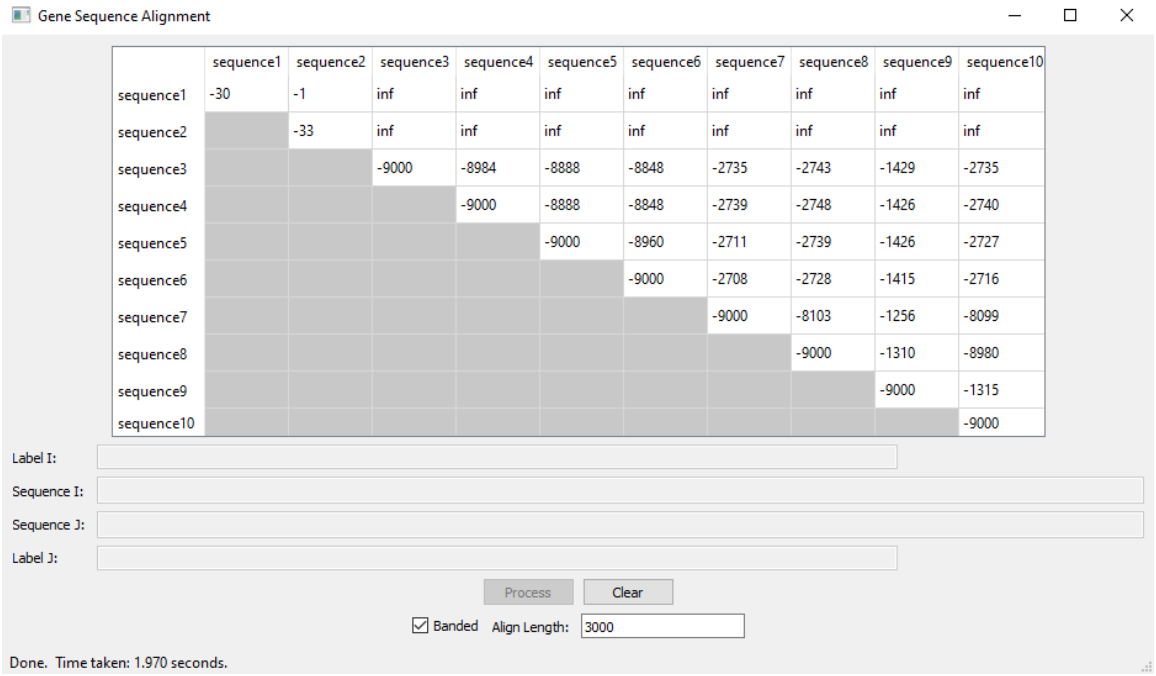

Figure 17: Banded Algorithm with an Alignment Length of 3,000.

Extracted Alignment for the Unrestricted and Banded Algorithm

Both the unrestricted algorithm and the banded algorithm had the same first 100-letter sequence, Figure 18 and 19. The first letter of each sequence in the output is the same letter in the genome.txt file where sequence 3 starts with a “g” and sequence 10 starts with an “a”. Below is the comparison between the sequences.

```
seq 3: gattgcgagcgatttgcgtgcgtgcatcccgcttc-actg--at-ctcttgttagat
seq10: -ataa-gagtgattggcgtccgtacgtacccttttctactctcaaactcttgttagtt
```

```
seq 3: cttttcataatctaaactttataaaaacatccactccctgta-
seq10: taaatc-taatctaaactttat--aaacgg-cacttcctgtgt
```

Figure 18: Unrestricted Algorithm with an Alignment Length of 1,000 at Sequence 3 and Sequence 10.

Figure 19: Banded Algorithm with an Alignment Length of 3,000 at Sequence 3 and Sequence 10.

Space and Time Complexity

Unrestricted Algorithm

There are two for loops of range [0, m] and [0, n] which has a space and time complexity of O(mn). Inside the for loops, there is indexing into the array to determine the edit distance and set the previous node. These operations occur in constant time. The alignment extraction section contains a while loop that linearly loops through the length of the two sequences where the runtime is O(m + n). Inside the while loops, there is indexing into the array to determine the previous node which is O(1). The total space and time complexity is m * (n * (1 + 1 + … + 1)) + (m+ n) * (1 + 1 + … + 1) = O(mn + m + n) = O(mn).

Banded Algorithm

There are three separate sets of two for loops with a combined range of [0, n] and [0, k) where k is the bandwidth and n is the length of the smaller sequence. The space and time complexity is O(kn). Inside the for loops, there is indexing into the array to determine the edit distance and set the previous node. These operations occur in constant time. The alignment extraction section contains a while loop that linearly loops through the length of the two sequences where the runtime is O(m + n). Inside the while loops, there is indexing into the array to determine the previous node which is O(1). The total space and time complexity is n * (k *(1 + 1 … + 1)) + (m + n) * (1 + 1 + … + 1) = O(kn + m + n) = O(kn).