# Report on Gene Sequencing

Sabal Subedi
Master's in Computer Science
Idaho State University
Pocatello, ID, 83209 USA
sabalsubedi@isu.edu

March 27, 2024

**Abstract**

This project report summarizes the use of dynamic programming for gene sequencing. The algorithm tries to find the alignment between two sequences.

## 1 Introduction

Gene sequencing using the dynamic programming to find the alignments between two sequences. It can be achieved using the Needleman-Wunsch algorithm. It constructs a matrix where cell represents the optimal alignment score of aligning a substring of one sequence with a substring of the other sequences. The algorith then fills the matrix iteratively considering three operations: match, insert and delete. After filling the matrix, the optimal alignment can be tracked back from the bottom-right cell to the top-left cell to obtain the aligned sequences.

# 2 Pseudocode and Asymptotic Analysis

## 2.1 Pseudocode for unrestricted alignment

```
def unrestricted_alignment(seq1, seq2,
   align_length):
# Used to implement Needleman-Wunsch scoring
MATCH = -3
INDEL = 5
SUB = 1

seq1 = seq1[:align_length]
seq2 = seq2[:align_length]

seq1_len = len(seq1)
seq2_len = len(seq2)

def check_boundaries(row, col):
    if row < 0 or row > seq1_len or col < 0 or
       col > seq2_len:
        return False
    return True

cost = []
# initializing the cost matrix
for row in range(seq1_len + 1):
    current_row = []
    for col in range(seq2_len + 1):
        if row == 0 or col == 0:
            current_row.append(row * INDEL if col
               == 0 else col * INDEL)
        else:
            current_row.append(float("inf"))
    cost.append(current_row)

backpointer = {}
# intializing back pointer
for row in range(seq1_len + 1):
```

```
32          for col in range(seq2_len + 1):
33              if col == 0:
34                  backpointer[(row, col)] = (row - 1,
                        col)
35              elif row == 0:
36                  backpointer[(row, col)] = (row, col -
                        1)
37
38      # populating the cost matrix and backpointer
39      for row in range(seq1_len + 1):
40          for col in range(seq2_len + 1):
41              if not check_boundaries(row, col):
42                  continue
43
44              if check_boundaries(row - 1, col - 1):
45                  cost[row][col] = cost[row - 1][col -
                        1] + (
46                      MATCH if seq1[row - 1] ==
                            seq2[col - 1] else SUB
47                  )
48                  backpointer[(row, col)] = (row - 1,
                        col - 1)
49
50              if check_boundaries(row - 1, col):
51                  prev_above_cost = cost[row - 1][col]
52                  new_above_cost = prev_above_cost +
                        INDEL
53                  if new_above_cost <= cost[row][col]:
54                      cost[row][col] = new_above_cost
55                      backpointer[(row, col)] = (row -
                            1, col)
56
57              if check_boundaries(row, col - 1):
58                  prev_left_cost = cost[row][col - 1]
59                  new_left_cost = prev_left_cost + INDEL
60                  if new_left_cost <= cost[row][col]:
61                      cost[row][col] = new_left_cost
```

```
                    backpointer[(row, col)] = (row,
                        col - 1)

    # aligning sequences
    alignment1 = ""
    alignment2 = ""
    (cur_row, cur_col) = (seq1_len, seq2_len)
    while cur_row > 0 or cur_col > 0:
        (prev_row, prev_col) = backpointer[(cur_row,
            cur_col)]
        if prev_row < cur_row and prev_col < cur_col:
            alignment1 += seq1[cur_row - 1]
            alignment2 += seq2[cur_col - 1]
        elif prev_row < cur_row:
            alignment1 += seq1[cur_row - 1]
            alignment2 += "-"
        elif prev_col < cur_col:
            alignment1 += "-"
            alignment2 += seq2[cur_col - 1]
        (cur_row, cur_col) = (prev_row, prev_col)

    alignment1 = alignment1[::-1][:100]
    alignment2 = alignment2[::-1][:100]

    score = cost[-1][-1]
    return score, alignment1, alignment2
```

**Time complexity and space complexity**
**Using unrestricted alignment**

1. initializing the cost and backpointer takes $O(n*m)$ time and $O(n*m)$ space
2. populating the cost matrix and backpointer takes $O(n*m)$ time and $O(n*m)$ space
3. aligning the sequences takes $O(n)$ times and $O(n)$ space

$$\begin{aligned} Time\_complexity &= O(n*m) + O(n*m) + O(n) \approx O(n*m) \\ Space\_complexity &= O(n*m) + O(n*m) + O(n) \approx O(n*m) \end{aligned} \tag{1}$$

## 2.2 Pseudocode for banded alignment

```
def banded_alignment(seq1, seq2, align_length, banded):
# Used to compute the bandwidth for banded version
MAXINDELS = 3

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

seq1 = seq1[:align_length]
seq2 = seq2[:align_length]

seq1_len = len(seq1)
seq2_len = len(seq2)

def check_boundaries(row, col):
    if row < 0 or row > seq1_len or col < 0 or col > seq2_len:
        return False
    if banded == True:
        # check if row and col is within the banded region
        if col < row - MAXINDELS or col > row + MAXINDELS:
            return False
        if row < col - MAXINDELS or row > col + MAXINDELS:
            return False
    return True

# adjusting the column
def adjust_column(row, col):
    if row > MAXINDELS:
        return col - (row - MAXINDELS)
    return col
```

```
32
33      # check sequence lengths and return inf if true
34      if abs(seq1_len - seq2_len) > MAXINDELS:
35          return float("inf"), "No Alignmnet Possible",
              "No Alignment Possible"
36
37      cost = []
38      # initializing cost matrix
39      for row in range(seq1_len + 1):
40          row_values = []
41          for col in range(row - MAXINDELS, row +
              MAXINDELS + 1):
42              if check_boundaries(row, col):
43                  if col == 0:
44                      value = row * INDEL
45                  elif row == 0:
46                      value = col * INDEL
47                  else:
48                      value = float("inf")
49                  row_values.append(value)
50          cost.append(row_values)
51
52      backpointer = {}
53      # intializing back pointer
54      for row in range(seq1_len + 1):
55          for col in range(seq2_len + 1):
56              if col == 0:
57                  backpointer[(row, col)] = (row - 1,
                      col)
58              elif row == 0:
59                  backpointer[(row, col)] = (row, col -
                      1)
60
61      # populating the cost matrix and backpointer
62      for row in range(seq1_len + 1):
63          for col in range(row - MAXINDELS, row +
              MAXINDELS + 1):
64              if not check_boundaries(row, col):
```

```
65                 continue
66
67             (row_curr, col_curr) = (row,
                   adjust_column(row, col))
68
69             if check_boundaries(row - 1, col - 1):
70                 cost[row_curr][col_curr] = cost[row -
                       1][
71                     adjust_column(row - 1, col - 1)
72                 ] + (MATCH if seq1[row - 1] ==
                       seq2[col - 1] else SUB)
73                 backpointer[(row, col)] = (row - 1,
                       col - 1)
74
75             if check_boundaries(row - 1, col):
76                 prev_above_cost = cost[row -
                       1][adjust_column(row - 1, col)]
77                 new_above_cost = prev_above_cost +
                       INDEL
78                 if new_above_cost <=
                       cost[row_curr][col_curr]:
79                     cost[row_curr][col_curr] =
                           new_above_cost
80                     backpointer[(row, col)] = (row -
                           1, col)
81
82             if check_boundaries(row, col - 1):
83                 prev_left_cost =
                       cost[row][adjust_column(row, col -
                       1)]
84                 new_left_cost = prev_left_cost + INDEL
85                 if new_left_cost <=
                       cost[row_curr][col_curr]:
86                     cost[row_curr][col_curr] =
                           new_left_cost
87                     backpointer[(row, col)] = (row,
                           col - 1)
88
```

```
89      alignment1 = ""
90      alignment2 = ""
91      (cur_row, cur_col) = (seq1_len, seq2_len)
92      while cur_row > 0 or cur_col > 0:
93          (prev_row, prev_col) = backpointer[(cur_row,
              cur_col)]
94          if prev_row < cur_row and prev_col < cur_col:
95              alignment1 += seq1[cur_row - 1]
96              alignment2 += seq2[cur_col - 1]
97          elif prev_row < cur_row:
98              alignment1 += seq1[cur_row - 1]
99              alignment2 += "-"
100         elif prev_col < cur_col:
101             alignment1 += "-"
102             alignment2 += seq2[cur_col - 1]
103         (cur_row, cur_col) = (prev_row, prev_col)
104
105     alignment1 = alignment1[::-1][:100]
106     alignment2 = alignment2[::-1][:100]
107
108     score = cost[-1][-1]
109     return score, alignment1, alignment2
```

**Time complexity and space complexity**
**Using unrestricted alignment**

1. initializing the cost and backpointer takes $O(n*k)$ time and $O(n*k)$ space
2. populating the cost matrix and backpointer takes $O(n*k)$ time and $O(n*k)$ space
3. aligning the sequences takes $O(n)$ times and $O(n)$ space

$$\begin{aligned} Time\_complexity &= O(n*k) + O(n*k) + O(n) \approx O(n*m) \\ Space\_complexity &= O(n*k) + O(n*k) + O(n) \approx O(n*m) \end{aligned} \tag{2}$$

# 3 Observations and Results

## 3.1 Observation

In this project, I have used both the unrestricted and banded algorithm to align the sequences. In order to populate the cost matrix, first, I performed check boundaries i.e. check if the current indices of cost matrix is within the valid region (in case of banded, the valid region is computed using $2 * MAXINDELS + 1$). Then, I have computed the cost for the diagonal, above and left (i.e. MATCH or SUB, INDEL operation) and updated the cost matrix at current indices by the lowest value among the three.
After populating the cost matrix, I initialized two empty string vairiable to hold the alignments. Using the backpointer, I compared the value of current version either diagonal, above or left and compute the alignment accordingly.
If the diagonal matches the current value, I appended $sequence1[current - 1]$ to alignment1 and $sequence2[current - 1]$ to alignment2.
If the above matches the current value, I appended $sequence1[current - 1]$ to alignment1 and a gap in sequence2.
If the left matches the current value, I appended $sequence2[current - 1]$ to alignment2 and a gap in sequence1.

**First 100 characters of sequence3 and sequence10 computed using the unrestricted alignment with k = 1000**

sequence3: gattgcgagcgatttgcgtgcgtgcatcccgcttc-actg–at-ctcttgttagatcttttcataatctaaactttataaaaa
catccactccctgta-
sequence10:-ataa-gagtgattggcgtccgtacgtacccttttctactctcaaactcttgttagtttaaatc-taatctaaactttataaa–
cggc-acttcctgtgt

**First 100 characters of sequence3 and sequence10 computed using the banded alignment with k = 3000**

sequence3: gattgcgagcgatttgcgtgcgtgcatcccgcttc-actg–at-ctcttgttagatcttttcataatctaaactttataaaaa
catccactccctgta-
sequence10:-ataa-gagtgattggcgtccgtacgtacccttttctactctcaaactcttgttagtttaaatc-taatctaaactttataaa–
cggc-acttcctgtgt

## 3.2 Results

.

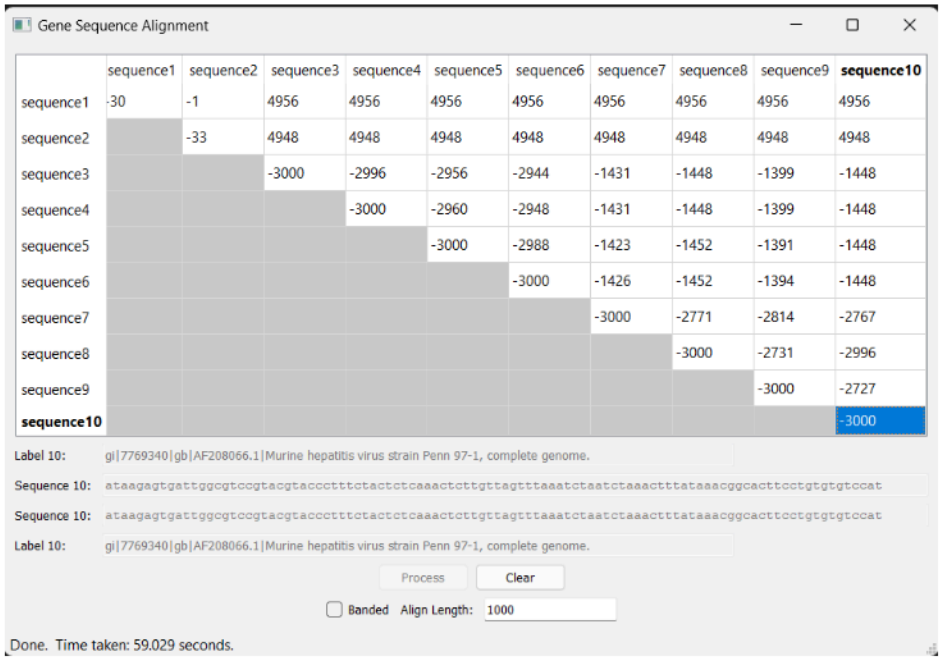


Figure 1: Unrestricted alignment for align length 1000

The figure (in figure 1) shows the optimal cost for each sequences and the time taken to solve the problem. Here, the align length is 1000 and it took 59.029 seconds to solve it using unrestricted alignment.


Figure 2: Banded alignment for align length 3000

The figure (in figure 2) shows the optimal cost for each sequences and the time taken to solve the problem. Here, the align length is 3000 and it took 09.671 seconds to solve it using banded alignment.

# 4 Conclusion

I was able to generate the code to compute both the unrestricted and banded alignment algorithm. I successfully got the time taken by the program to solve the given sequences.