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 each cell represents the optimal alignment score of aligning a substring of one sequence with a substring of the other sequences. The algorithm 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

```
1     def unrestricted_alignment(seq1, seq2,
         align_length):
2     # Used to implement Needleman-Wunsch scoring
3     MATCH = -3
4     INDEL = 5
5     SUB = 1
6     seq1 = seq1[:align_length]
7     seq2 = seq2[:align_length]
8     seq1_len = len(seq1)
9     seq2_len = len(seq2)
10
11    def check_boundaries(row, col):
12        if row < 0 or row > seq1_len or col < 0 or
             col > seq2_len:
13            return False
14        return True
15
16    cost = []
17    # initializing the cost matrix
18    for row in range(seq1_len + 1):
19        current_row = []
20        for col in range(seq2_len + 1):
21            if row == 0 or col == 0:
22                current_row.append(row * INDEL if col
                     == 0 else col * INDEL)
23            else:
24                current_row.append(float("inf"))
25        cost.append(current_row)
26
27    backpointer = {}
28    # intializing back pointer
29    for row in range(seq1_len + 1):
30        for col in range(seq2_len + 1):
31            if col == 0:
```

```
32                 backpointer[(row, col)] = (row - 1,
                       col)
33             elif row == 0:
34                 backpointer[(row, col)] = (row, col -
                       1)
35
36     # populating the cost matrix and backpointer
37     for row in range(seq1_len + 1):
38         for col in range(seq2_len + 1):
39             if not check_boundaries(row, col):
40                 continue
41             if check_boundaries(row - 1, col - 1):
42                 cost[row][col] = cost[row - 1][col -
                       1] + (
43                     MATCH if seq1[row - 1] ==
                           seq2[col - 1] else SUB
44                 )
45                 backpointer[(row, col)] = (row - 1,
                       col - 1)
46             if check_boundaries(row - 1, col):
47                 prev_above_cost = cost[row - 1][col]
48                 new_above_cost = prev_above_cost +
                       INDEL
49                 if new_above_cost <= cost[row][col]:
50                     cost[row][col] = new_above_cost
51                     backpointer[(row, col)] = (row -
                           1, col)
52             if check_boundaries(row, col - 1):
53                 prev_left_cost = cost[row][col - 1]
54                 new_left_cost = prev_left_cost + INDEL
55                 if new_left_cost <= cost[row][col]:
56                     cost[row][col] = new_left_cost
57                     backpointer[(row, col)] = (row,
                           col - 1)
58
59     # aligning sequences
60     alignment1 = ""
61     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. intializing 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

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

Using banded alignment

1. intializing 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

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

3 Observations and Results

3.1 Observation

In this project, I have used both the unrestricted and banded alignment 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 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 variable to hold the alignments. Using the backpointer, I compared the value of current vs 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 apended $sequence1[current-1]$ to alignment1 and a gap in alignment2.

If the left matches the current value, I apended $sequence2[current-1]$ to alignment2 and a gap in alignment1.

Sample output obtained after following the above process:
sequence1: ataagagtgattggcgtccgtacgtacccttttctactctcaaactcttgttagtttaaatct
sequence2: ataagagtgattggcgtccgtacgtacccttttctactctcaaactcttgttagtttaaatct

First 100 characters of sequence3 and sequence10 computed using unrestricted alignment with k= 1000
sequence3: gattgcgagcgatttgcgtgcgtgcatcccgcttc-actg–at-ctcttgttagatcttttcataatctaaactttataaaaacatccactccc

sequence10: -ataa-gagtgattggcgtccgtacgtacccttttctactctcaaactcttgttagtttaaatc-taatctaaactttataaa–cggc-acttcctgtgt

First 100 characters of sequence3 and sequence10 computed using banded alignment with k= 4000
sequence3: gattgcgagcgatttgcgtgcgtgcatcccgcttc-actg–at-ctcttgttagatcttttcataatctaaactttataaaaacatccactccc

sequence10: -ataa-gagtgattggcgtccgtacgtacccttttctactctcaaactcttgttagtttaaatc-taatctaaactttataaa–cggc-acttcctgtgt

3.2 Results

.

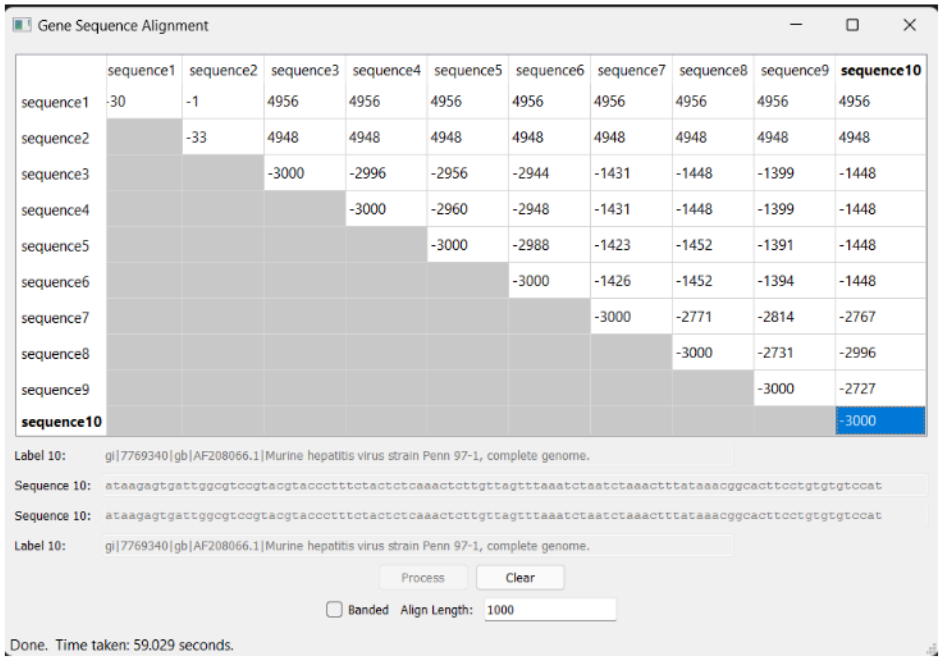

Figure 1: Unrestricted alignment for align length 1000

The figure (in figure 1) show 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) show 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 restricted and banded alignment algorithm. I successfully got the time taken by the program to solve the given sequences.