Ali Ramazani
Winter 2024
CS362

Implementing Hirschberg's Algorithm

My final project involved studying Hirschberg's algorithm, which is a space-efficient version of the Needleman-Wunsch algorithm. The main problem that this algorithm addresses is memory bottleneck when it comes to aligning long sequences of genetic information like DNA and proteins. Thus, the algorithm introduces a divide-and-conquer approach that can find the optimal global alignment of two given sequences.

The first part of the algorithm involves finding the alignment score in linear space. The general idea is that when calculating the alignment score for a specific cell in the grid, we only need to look at the three neighboring cells, which are left, diagonal, and top cells. We check which one gives us the minimum cost /maximum score to determine the score for the current cell. This also indicates that we only need to look back at the previous column and the current column to find the alignment score for the cell in the current column. Thus, we can discard any other previously filled columns to save memory and reuse the previous column to store the alignment score for the next column. Since the size of columns depends on the length of our sequence 2, we can say that our space complexity for finding the alignment score is O(2m) since we use two columns. We can ignore the constant and say that the space complexity is O(m), which is linear. Here is my implementation of this step (Algorithm 1):

```python
def linear_space_alignment_score(s1, s2, match=1, mismatch=-1, gap=-2):
  n, m = len(s1), len(s2)
  prev_score = np.zeros(m+1)
  curr_score = np.zeros(m+1)
  for j in range(m+1):
    prev_score[j] = j * gap
  for i in range(1, n+1):
    curr_score[0] = i * gap
    for j in range(1, m+1):
      character_match = s1[i-1] == s2[j-1]
      if character_match:
        score = match
      else:
        score = mismatch
      curr_score[j] = max(prev_score[j-1] + score, prev_score[j] + gap,
curr_score[j-1] + gap)
    prev_score = curr_score.copy()
  return curr_score
```

This algorithm takes in two sequences as input and a pre-defined scoring scheme, which includes the score for a match, mismatch, and a gap penalty. Next, we initialize two arrays of length n + 1 and m + 1, where n is the length of sequence 1 and m is the length of sequence 2. The first array is used to store the alignment score for the previous column and the second array is used to store the alignment score for the current column. In the next line, we go through each character in sequence 2 and align it to a gap and populate the first column with the alignment score, which in this case is just the gap penalty. Now that we have our previous column filled, we can start filling the current column, which is called curr_score in the code. We go through each character in the first sequence and align it with the characters in the second sequence. The first value in the current column is another gap penalty since we are aligning a gap from sequence 2 to a character in sequence 1. We then use a conditional statement to check for a match or mismatch and use a variable to store the resulting score for the alignment. Next, we add the score to the value of the diagonal cell, indicating a match or mismatch. Similarly, we add the gap penalty to the left cell, indicating insertion, and adding the gap penalty to the top cell, indicating deletion. We choose the maximum value from these three neighboring cells to update the alignment score for the current cell. Lastly, we make a copy of the current column and store it in the previous column, and populate the current column with the alignment score for the next column. We repeat this process until we find the alignment score for the last column. To summarize, you can think of this algorithm as finding the alignment scores in the last column of the Needleman-Wunsch scoring matrix. Thus, we were able to find the optimal alignment score in linear space.

Testing the algorithm on an example from Wikipedia:

seq1 = "TATGC"
seq2 = "AGTACGCA"
match=2, mismatch=-1, gap=-2

Here is my result: [-10. -6. -2. -1. -3. 1. -1. 3. 1], which corresponds to the last column in the following Needleman–Wunsch matrix:

		T	A	T	G	C
	0	-2	-4	-6	-8	-10
A	**-2**	-1	0	-2	-4	-6
G	**-4**	-3	-2	-1	0	-2
T	-6	**-2**	-4	0	-2	-1
A	-8	-4	**0**	-2	-1	-3
C	-10	-6	-2	**-1**	-3	1
G	-12	-8	-4	-3	**1**	-1
C	-14	-10	-6	-5	-1	**3**
A	-16	-12	-8	-7	-3	**1**

Next, we will discuss how to find the actual alignment in linear space. Hirschberg's algorithm uses a divide-and-conquer approach to find the optimal alignment. It starts by dividing the grid in half to find the middle column. Next, it aims to find the middle node in the middle column, which is a node where an optimal alignment path crosses the middle column. To do this, it uses the above code that we just discussed to find the optimal alignment score for all the columns in the left half of the grid and returns the alignment scores for the last column, which in this case is the middle column of the original grid. Similarly, we apply the score alignment algorithm for the right half of the grid and return the alignment scores for the middle column. Next, we sum the scores from the two resulting middle columns and find the maximum value, which is the middle node that we were looking for, and we want to store its position as part of the solution. Lastly, we recursively call Hirschberg's algorithm to find the optimal alignment from the top left node (source node) in the grid to the middle node. Similarly, we do the same process to find the optimal alignment from the middle node to the bottom right node (sink node) in the grid. The combined results will give us the overall optimal alignment from source to sink. Here is my algorithm for finding the middle node (Algorithm 2):

```python
def hirschberg_alignment(seq1, seq2):
  n = len(seq1)
  m = len(seq2)
  middle = n // 2
  optimal_nodes = []
  seq1_left = seq1[:middle+1]
  left_score = linear_space_alignment_score(seq1_left, seq2)
  print(left_score)
  seq1_right = seq1[middle:]
  right_score = linear_space_alignment_score(seq1_right[::-1], seq2[::-1])
  print(right_score)
  sum_of_score = left_score + right_score[::-1]
  print(sum_of_score)
  max_score = max(sum_of_score)
  optimal_node = np.argmax(sum_of_score)
  optimal_nodes.append((middle, max_score, optimal_node))

  return optimal_nodes
```

This algorithm finds the middle column by dividing the length of sequence 1 in half. Next, we split our sequence 1 in half and call our Algorithm 1 to find the optimal alignment scores for the left half. We do the same process for the right half. However, this time, we need to invert the roles of source and sink. This is why the code reverses the right side of sequence 1 and sequence 2. We then sum the two resulting scores but keep in mind that we have to reverse the right score one more time to undo the first reversal before adding the score. We find the maximum value to get the middle node and store its position.

Testing the algorithm on an example from Wikipedia:

seq1 = "TATGC"
seq2 = "AGTACGCA"
match=2, mismatch=-1, gap=-2

My results:
Score left: [-8. -4. 0. -2. -1. -3.]
Score right: [-3. -1. 1. 0. -4. -8.]
Score sum: [-11. -5.   1. -2. -5. -11.]
(middle, max_score, optimal_node): [(4, **1.0**, 2)]

Wikipedia results

One starts with the top level call to $\mathrm{Hirschberg}(\mathrm{AGTACGCA}, \mathrm{TATGC})$, which splits the first argument in half: $X = \mathrm{AGTA} + \mathrm{CGCA}$. The call to $\mathrm{NWScore}(\mathrm{AGTA}, Y)$ produces the following matrix:

```
      T   A   T   G   C
  0  -2  -4  -6  -8 -10
A -2  -1   0  -2  -4  -6
G -4  -3  -2  -1   0  -2
T -6  -2  -4   0  -2  -1
A -8  -4   0  -2  -1  -3
```

Likewise, $\mathrm{NWScore}(\mathrm{rev}(\mathrm{CGCA}), \mathrm{rev}(Y))$ generates the following matrix:

```
     C   G   T   A   T
  0 -2  -4  -6  -8 -10
A -2 -1  -3  -5  -4  -6
C -4  0  -2  -4  -6  -5
G -6 -2   2   0  -2  -4
C -8 -4   0   1  -1  -3
```

Their last lines (after reversing the latter) and sum of those are respectively

```
ScoreL      = [ -8 -4  0 -2 -1 -3 ]
rev(ScoreR) = [ -3 -1  1  0 -4 -8 ]
Sum         = [-11 -5  1 -2 -5 -11]
```

Discussion:

Unfortunately, my implementation failed when I tried to recursively call the function to find all the optimal nodes. It gave me an error that said something along the lines of “maximum recursion depth exceeded.” Therefore, I decided to remove the recursion part to make sure it worked. I included the sequence example from Wikipedia to test my code. Overall, my results matched the results from Wikipedia, which indicates a step in the right direction.

Note: I requested an extension to work on the algorithm and figure out if I can fix the recursion part. I apologize again for all the inconvenience, and I want to thank you for being supportive of me and helping me in some of the difficult moments of my life.