

Homework 2

Elliott Pryor

7 September 2021

Problem 1

Propose a dynamic programming algorithm to solve longest common substring problem

This is almost identical to the Edit Distance algorithm.

We create $V = nxm$ matrix. We set $V_{0,0} = 0$, and $V_{i,0} = 0, V_{0,j} = 0$ because the longest common subsequence is at least 0 (by matching no characters together).

We then let $V_{i,j} = \begin{cases} V_{i-1,j-1} + 1 & \text{if } S[i] = T[j] \\ \max(V_{i,j-1}, V_{i-1,j}) & \text{otherwise} \end{cases}$ for $i, j \neq 0$. The first case handles a match between two characters. The second case skips a character (because subsequence can be non-consecutive).

The length of the longest common substring is $V_{n,m}$.

We can also construct the alignment of the common subsequence in the same way as in Needleman-Wunch.

PROOF. Correctness.

By induction, we show that $V_{i,j}$ is LCS (longest common substring) of $S[1..i]$ and $T[1..j]$. For $i, j = 0$ this is trivially true by matching empty strings.

We assume that $V_{i-1,j-1}, V_{i,j-1}, V_{i-1,j}$ are the LCS for their respective substrings. Suppose $V_{i,j}$ is not the length of longest common subsequence. Assume $S[i] = T[j]$. Then $V_{i,j} = 1 + V_{i-1,j-1}$. Then, either we shouldn't match $S[i], T[j]$ which we can get a longer LCS at no consequence by matching them. Or $V_{i-1,j-1}$ is not an LCS. A contradiction in both cases.

Then assume $S[i] \neq T[j]$. If we match against $V_{i-1,j-1}$ we break the substring since they don't match. So we want to skip one letter of the string. If we skip a letter in S then we choose $V_{i-1,j}$. Suppose there is a more optimal choice. Then, we shouldn't have skipped a letter in S (and instead in T), which is handled by the other case of the max. Or $V_{i-1,j}$ is not optimal, which is a contradiction. The same holds in reverse for choosing to skip a letter in T .

Thus inductive step holds by contradiction in all cases.

□

Problem 2 Given two sequences S, T (not necessarily the same length), let G, L, H be the scores of the optimal global alignment, optimal local alignment, and optimal global alignment without penalizing leading or trailing spaces.

- a) Give an example of S, T so that the three scores are different
 - b) Prove or disprove the statement $L \geq H \geq G$
-

a) $S = \text{aaatata}, T = \text{tatc}$ $G = 2, L = 6, H = 5$

- b) It is correct. **PROOF.** We first prove $L \geq H$ by contradiction. Suppose not, then $L < H$. Given some global alignment $S, T = H$ (S, T) are globally aligned strings. Let a be the first non-space character in T , and b be the last non-space character in T . Similarly, let c be the first non-space character in S , and d be the last non-space character in S . Then construct substrings $S' = S[a..b], T' = T[c..d]$. Then the alignment of $S', T' = H$. Since leading and trailing spaces are not penalized in H , S', T' consist of only the characters that are scored in H . Since S', T' are valid substrings, we have a contradiction.

Now we prove $H \geq G$ by contradiction. We assume that $H < G$. Since G is the optimal global alignment. We can choose an alignment for H to be the same as this global alignment. Then clearly $H \geq G$, since if G has no leading or trailing spaces $H = G$. Thus a contradiction.

Therefore $L \geq H \geq G$

□

Problem 3

Implement the global alignment algorithm from class. Your program should read in a FASTA file (see HW1). You can assume that the file just contains two sequences, e.g.

```
> seq1
ACTGGGAAA
> seq2
CTGGAACA
```

The filename should be supplied as a command-line parameter.

Align the first string with the second string. Print out one optimal alignment.

You can assume a simplified scoring function δ that has the following form: $\delta(\text{match}) = 2$ $\delta(\text{mismatch}) = -1$ $\delta(\text{insertion/deletion}) = -1$

Demonstrate your algorithms on two test cases (use screen shots to show runs).

```
PS C:\Users\pryor\Documents\GithubProjects\CompBio\HW2> python .\EditDistance.py ./example1.fasta
Opening: ./example1.fasta
Computing alignment between S=seq 1 and T=seq 2
Optimal Score: 2.0
[[ 0. -1. -2. -3. -4.]
 [-1. -1. -2. -3. -4.]
 [-2. -2.  1.  0. -1.]
 [-3.  0.  0.  0.  2.]]
*****
_cat
taat
*****
_cat
taat
*****
_c_at
taat
```

Figure 1: Example 1, using cat and taat from class

```
PS C:\Users\pryor\Documents\GithubProjects\CompBio\HW2> python .\EditDistance.py ./example2.fasta
Opening: ./example2.fasta
Computing alignment between S=seq1 and T=seq2
Optimal Score: 11.0
[[ 0. -1. -2. -3. -4. -5. -6. -7. -8.]
 [-1. -1. -2. -3. -4. -2. -3. -4. -5.]
 [-2.  1.  0. -1. -2. -3. -3. -1. -2.]
 [-3.  0.  3.  2.  1.  0. -1. -2. -2.]
 [-4. -1.  2.  5.  4.  3.  2.  1.  0.]
 [-5. -2.  1.  4.  7.  6.  5.  4.  3.]
 [-6. -3.  0.  3.  6.  6.  5.  4.  3.]
 [-7. -4. -1.  2.  5.  8.  8.  7.  6.]
 [-8. -5. -2.  1.  4.  7. 10.  9.  9.]
 [-9. -6. -3.  0.  3.  6.  9.  9. 11.]]
*****
ACTGGGAA_A
_CT_GGAACA
*****
ACTGGGAA_A
_CT_GGAACA
*****
ACTGGGAA_A
_CTG_GAACA
```

Figure 2: Example 2, using input from the homework description

I did the bonus and we can see that it prints all optimal alignments. It produces duplicated results because I don't use a set, and the first move can take 2 paths and produce the same path. Consider example 1 from class. We can go $(0,0) \rightarrow (0,1) \rightarrow (2,1)$ or $(0,0) \rightarrow (1,1) \rightarrow (2,1)$. Both just add a space to the front of cat. So they are indeed 2 different paths with the same 'phenotype'.

```

1 import sys
2 import numpy as np
3
4
5 def delta(c1, c2):
6     # Helper function used for scoring
7     if c1 == c2:
8         return 2
9     else:
10        return -1
11
12
13 def construct_alignment(P, S, T, i=-1, j=-1, out_S='', out_T=''):
14     # Reconstruct the optimal alignment from the output of needleman(S,T)
15     if i == -1 and j == -1:
16         i, j = len(S), len(T)
17
18     if i == 0 and j == 0: # base case
19         print('*' * 80) # print out the optimal alignment
20         print(out_S)
21         print(out_T)
22         return out_S, out_T
23
24     retS, retT = '', ''
25     moves = P[i][j]
26     for move in moves:
27         if move == 0: # is just a part of the output of P, due to its construction.
28             # So skip move
29             continue
30         elif move == 1: # diagonal
31             out_S = S[i - 1] + out_S
32             out_T = T[j - 1] + out_T
33             i = i - 1
34             j = j - 1
35         elif move == 2: # delete
36             out_S = S[i - 1] + out_S
37             out_T = '_' + out_T
38             i = i - 1
39             j = j
40         elif move == 3: # insert
41             out_S = '_' + out_S
42             out_T = T[j-1] + out_T
43             i = i
44             j = j - 1
45         retS, retT = construct_alignment(P, S, T, i, j, out_S, out_T) # recur
46
47     return retS, retT # return the last optimal alignment
48

```

```

49 def needleman(S, T):
50     V = np.zeros((len(S) + 1, len(T) + 1)) # value
51     P = np.zeros((len(S) + 1, len(T) + 1)) # path
52
53     # initialize V
54     V[:, 0] = [-i for i in range(len(S) + 1)]
55     V[0, :] = [-i for i in range(len(T) + 1)]
56     P[:, 0] = 2 * np.ones(len(S) + 1) # vertical case 2
57     P[0, :] = 3 * np.ones(len(T) + 1) # horizontal case 1
58     P[0, 0] = 0 # reset 0, 0
59
60     P = [[[x] for x in row] for row in P] # convert to list of lists
61     for r in range(1, len(V)): # 2nd row onwards
62         for c in range(1, len(V[r])): # 2nd col onwards
63
64             replace = V[r-1, c-1] + delta(S[r-1], T[c-1])
65             delete = V[r-1, c] + delta(S[r-1], '_')
66             insert = V[r, c - 1] + delta('_', T[c-1])
67
68             arr = np.array([replace, delete, insert])
69
70             V[r, c] = max(arr)
71             if replace == max(arr):
72                 P[r][c].append(1)
73             if delete == max(arr):
74                 P[r][c].append(2)
75             if insert == max(arr):
76                 P[r][c].append(3)
77
78     return V, P
79
80
81 def read_fasta(filepath):
82     """
83     Returns a dict key=sequence_name, value=sequence from the fasta file
84     :param filepath:
85     :return:
86     """
87     file = open(filepath)
88     print(f"Opening: {filepath}")
89     started = False
90     header = ''
91     text = ''
92     sequences = {}
93     for line in file: # reads in fasta file. Allows for sequence to continue on
94         # multiple lines
95         if line.startswith('>'):
96             if started:
97                 sequences[header] = text.replace('\n', '')
98                 text = ''
99                 started = True
100             header = line.replace('>', '').strip()
101         else:
102             text += line.strip()
103     sequences[header] = text.replace('\n', '')

```

```
103     return sequences
104
105
106 if __name__ == '__main__':
107     if len(sys.argv) == 2:
108         # run fasta file
109         # expect arguments z_algorithm.py <path to file>
110         filepath = sys.argv[1]
111         seq = read_fasta(filepath)
112         vals = list(seq.values())
113         keys = list(seq.keys())
114         print(f"Computing alignment between S={keys[0]} and T={keys[1]}")
115         V, P = needleman(vals[0], vals[1])
116         print(f"Optimal Score: {V[-1,-1]}")
117         print(V)
118         s, t = construct_alignment(P, vals[0], vals[1])
119     else:
120         V, P = needleman('cat', 'taat')
121         print(f"Optimal Score: {V[-1, -1]}")
122         print(V)
123         s, t = construct_alignment(P, 'cat', 'taat')
124     pass
```