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 = aaatata, T = 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 $L \geq H \geq G$ $\square$

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 delta that has the following form: delta(match) = 2 delta(mismatch) = -1 delta(insertion/deletion) = -1

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

```python
import sys
import numpy as np


def delta(c1, c2):
    # Helper function used for scoring
    if c1 == c2:
        return 2
    else:
        return -1


def construct_alignment(P, S, T, i=-1, j=-1, out_S='', out_T=''):
    # Reconstruct the optimal alignment from the output of needleman(S,T)
    if i == -1 and j == -1:
        i, j = len(S), len(T)

    if i == 0 and j == 0:  # base case
        print('*' * 80)  # print out the optimal alignment
        print(out_S)
        print(out_T)
        return out_S, out_T

    retS, retT = '', ''
    moves = P[i][j]
    for move in moves:
        if move == 0:  # is just a part of the output of P, due to its construction.
 So skip move
            continue
        elif move == 1:  # diagonal
            out_S = S[i - 1] + out_S
            out_T = T[j - 1] + out_T
            i = i -1
            j = j -1
        elif move == 2:  # delete
            out_S = S[i - 1] + out_S
            out_T = '_' + out_T
            i = i - 1
            j = j
        elif move == 3:  # insert
            out_S = '_' + out_S
            out_T = T[j-1] + out_T
            i = i
            j = j - 1
        retS, retT = construct_alignment(P, S, T, i, j, out_S, out_T)  # recur

    return retS, retT  # return the last optimal alignment


def needleman(S, T):
    V = np.zeros((len(S) + 1, len(T) + 1))  # value
    P = np.zeros((len(S) + 1, len(T) + 1))  # path

    # initialize V
    V[:, 0] = [-i for i in range(len(S) + 1)]
    V[0, :] = [-i for i in range(len(T) + 1)]
    P[:, 0] = 2 * np.ones(len(S) + 1)  # vertical case 2
    P[0, :] = 3 * np.ones(len(T) + 1)  # horizontal case 1
    P[0, 0] = 0  # reset 0, 0

    P = [[[x] for x in row] for row in P ]  # convert to list of lists
    print(P)
    for r in range(1, len(V)):  # 2nd row onwards
        for c in range(1, len(V[r])):  # 2nd col onwards

            replace = V[r-1, c-1] + delta(S[r-1], T[c-1])
            delete = V[r-1, c] + delta(S[r-1], '_')
            insert = V[r, c - 1] + delta('_', T[c-1])

            arr = np.array([replace, delete, insert])

            V[r, c] = max(arr)
            if replace == max(arr):
                P[r][c].append(1)
            if delete == max(arr):
                P[r][c].append(2)
            if insert == max(arr):
                P[r][c].append(3)

    return V, P


def read_fasta(filepath):
    """
    Returns a dict key=sequence_name, value=sequence from the fasta file
    :param filepath:
    :return:
    """
    file = open(filepath)
    print(f"Opening: {filepath}")
    started = False
    header = ''
```

```python
    text = ''
    sequences = {}
    for line in file:  # reads in fasta file. Allows for sequence to continue on multiple lines
        if line.startswith('>'):
            if started:
                sequences[header] = line.replace('\n', '')
                text = ''
            started = True
            header = line.replace('>', '').strip()
        else:
            text += line.strip()
    sequences[header] = text.replace('\n', '')
    return sequences


if __name__ == '__main__':
    if len(sys.argv) == 2:
        # run fasta file
        # expect arguments z_algorithm.py <path to file>
        filepath = sys.argv[1]
        seq = read_fasta(filepath)
        vals = list(seq.values())
        keys = list(seq.keys())
        print(f"Computing alignment between S={keys[0]} and T={keys[1]}")
        V, P = needleman(vals[0], vals[1])
        print(f"Optimal Score: {V[-1,-1]}")
        s, t = construct_alignment(P, vals[0], vals[1])
    else:
        V, P = needleman('cat', 'taat')
        print('*' * 80)
        print(V)
        print()
        print(P)
        print(f"Optimal Score: {V[-1, -1]}")
        print('*' * 80)

        s, t = construct_alignment(P, 'cat', 'taat')
        print(s)
        print(t)
        pass
```