35/40

William Argus

A12802324

ECE 208 Homework 2 Writeup

1) The recursive formula of the dynamic programming

The recursive formula, similar to that seen in the lecture slides, is:

```
E(i,j,k) =
        If i*j*k == 0
                Return max( i, j, k)
        Else
           Return max(
                E(i,j-1,k) + gapPenalty + gapPenalty,
                E(i,j-1,k-1) + Blosum(s2(j-1), s3(k-1)) + gapPenalty + gapPenalty,
                E(i,j,k-1) + gapPenalty + gapPenalty,
                E(i-1,j-1,k) + Blosum(s1(i-1), s2(j-1))  + gapPenalty + gapPenalty,
                E(i-1,j-1,k-1) + Blosum(s1(i-1), s2(j-1)) + Blosum(s3(k-1), s2(j-1)) + Blosum(s2(j-1), s3(k-1)),
                E(i-1,j,k-1) + Blosum(s3(k-1), s2(j-1)) + gapPenalty + gapPenalty,
                E(i-1,j,k) + gapPenalty + gapPenalty,
```

Note that in this formula, Blosum(s1(i-1), s2(j-1)) refers to a function which will look up the value in the Blossum matrix for the characters given at the given indices (in this case i-1 and j-1) of s1 and s2.

The actual code of a recursive implementation done on Python is pasted below as a reference.

```python
def go(i,j,k): #s1: i, s2: j, s3: k
   '''
   if cube[i,j,k] != empty:
      return cube[i,j,k]
      dynamic[i,j,k] = 1+dynamic[i,j,k]
      print('dynamic')
      print(i,j,k)
   '''
   if (abs(i-j)> diff) or (abs(k-j)> diff) or (abs(i-k)> diff): #keep it diagonalized to reduce program runtime
      return -999999999999999999999999999999999999999
   else:
      if cube[i,j-1,k] != empty: #one
         one = cube[i,j-1,k] + gap_Penalty_two #1
         #print('dynamic')
      else:
         one = go(i,j-1,k) + gap_Penalty_two #1
      if cube[i, j-1,k-1] != empty: #two
         two = cube[i, j-1,k-1] + BLOSUM62[AMINOS.index(s2[j-1])][AMINOS.index(s3[k-1])]+ gap_Penalty_two #2
         #print('dynamic')
      else:
         two = go(i,j-1, k-1) + BLOSUM62[AMINOS.index(s2[j-1])][AMINOS.index(s3[k-1])]+ gap_Penalty_two #2
      if cube[i, j,k-1] != empty: #three
         three = cube[i, j,k-1] + gap_Penalty_two #3
         #print('dynamic')
      else:
         three = go(i, j, k-1) + gap_Penalty_two #3
```

```
if cube[i-1, j-1,k] != empty: #four
   four = cube[i-1, j-1,k] + BLOSUM62[AMINOS.index(s1[i-1])][AMINOS.index(s2[j-1])] + gap_Penalty_two #4
   #print('dynamic')
else:
   four = go(i-1, j-1, k) + BLOSUM62[AMINOS.index(s1[i-1])][AMINOS.index(s2[j-1])] + gap_Penalty_two #4
if cube[i-1,j-1,k-1] != empty: #five
   five = cube[i-1,j-1,k-1] + BLOSUM62[AMINOS.index(s1[i-1])][AMINOS.index(s2[j-1])] + BLOSUM62[AMINOS.index(s2[j-1])][AMINOS.index(s3[k-1])] + BLOSUM62[AMINOS.index(s1[i-1])][AMINOS.index(s3[k-1])] #5
   #print('dynamic')
else:
   five = go(i-1,j-1,k-1) + BLOSUM62[AMINOS.index(s1[i-1])][AMINOS.index(s2[j-1])] + BLOSUM62[AMINOS.index(s2[j-1])][AMINOS.index(s3[k-1])] + BLOSUM62[AMINOS.index(s1[i-1])][AMINOS.index(s3[k-1])] #5
if cube[i-1, j, k-1] != empty: #six
   six = cube[i-1, j, k-1] + BLOSUM62[AMINOS.index(s1[i-1])][AMINOS.index(s3[k-1])] + gap_Penalty_two#6
   #print('dynamic')
else:
   six = go(i-1,j, k-1) + BLOSUM62[AMINOS.index(s1[i-1])][AMINOS.index(s3[k-1])] + gap_Penalty_two#6
if cube[i-1,j,k] != empty: #seven
   seven = cube[i-1,j,k] + gap_Penalty_two #7
   #print('dynamic')
else:
   seven = go(i-1, j, k) + gap_Penalty_two #7
options = [one, two, three, four, five, six, seven]
maxi = max(options)
check[i,j,k] = 1+check[i,j,k]
cube[i,j,k] = maxi
return maxi
```

2) The base case(s) of the dynamic programming

If the recursive function is E(i,j,k), then base case is i*j*k == 0. This implies that the end of the cube has been reached, meaning that one or more of the sequences has been recursed all the way back to the point of being past the start. This means that the recursion has reached its end and the maximum of (i,j,k) will be returned.

3) The asymptotic running time of the algorithm

Due to the fact that the sequences being used are from real animals, it can expected that they will be relatively close in length. Therefore the sequences are assumed to all be of length n, with n being the length of the longest sequence. The running time will be derived in terms of n1, n2, and n3, and then simplified to just the value of n, length of longest sequence.

First there is the process of filling out the three outside "faces" of the 3D array (when visualizing the 3D array as a cube). These faces are (n1 x n2), (n1 x n3), and (n2 x n3). Then there is the process of filling out the rest of the "cube". This involves iterating through the (n1 -1) parts of the first sequence, and at each instance, iterating through (n2 – 1) parts of the second sequence, and at each instance of that, iterating through (n3 – 1) instances of the third sequence, representing the worst case running time of the algorithm. However, at each of these instances, there is a look-up cost of the Blossum62 array, but this can be ignored as it does not depend on n, it is constant, so the cost of filling out the interior of this 3D array is (n1 – 1)(n2 – 1)(n3 – 1). In total, filling out this 3D array, "cube", is represented is (n1 x n2) + (n1 x n3) + (n2 x n3) + (n1 – 1)(n2 – 1)(n3 – 1).

After filling out the cube, it must be backtracked in order to create the aligned sequences. The worst case scenario of this instance would be if for every letter sequence 1 advanced, sequence 2 and 3 were given gaps, and likewise with s1 and s3 being given gaps when s2 advances, and s1 and s2 being given gaps when s3 advances, leading to n1 + n2 + n3 operations to complete the backtracking.

The total the running cost can be described as: (n1 x n2) + (n1 x n3) + (n2 x n3) + (n1 – 1)(n2 – 1)(n3 – 1) + n1 + n2 + n3. Using the approximation described above wherein n1, n2, and n3 are approximately equal to n, which n being the longest of the three sequences, the total worst case cost of the algorithm is O($3n^2 + n^3$) for the 3D array creation, and O($3n$) for the backtracking. Therefore the total running cost of the algorithm is $O(n^3)$.

4) Your logic behind setting the gap penalty.

Since the three-way sequence alignment was done using the Needleman-Wunsch algorithm, the gap penalty was used from that algorithm as well. Specifically, the gap penalty $d = -log_2(\beta)$. Using the hint given on canvas, $\beta = P_{indel}$. Where P_{indel} is given in the equation indel rate, $r = \frac{P_{indel}}{P_{subs}}$. $P_{subs} = P(A\ not\ equal\ B)$. Hence $\underline{P_{subs} = \frac{1}{20}}$. So the gap penalty was calculated for a given indel rate as follows: $d = -log_2(r * P_{subs})$.

this is incorrect!
-5 pts