MATH578A Part 2, Homework 1

Raktim Mitra
USC ID: 1487079265 email: raktimmi@usc.edu

April 19, 2020

Q1.

I present an $O(mn)$ solution for the k-core alignment problem. The steps of the algorithm are as follows:

1. First we calcalate scores for all possible k-stretch of match/mismatch between the two sequences s_1 and s_2. If we do this naively we can do it in $O(kmn)$ time (O(mn) possibilities which take O(k) time each). However, this can be done in O(mn). Let, M be one alignment score matrix filled only using the diagonal steps (i.e. normal global alignment except we takae diagonal steps always. i.e. $M[i][j] = M[i-1][j-1] + score(s_1[i], s_2[j])$)(takes O(mn) time). Now, we can calculate the required K-stretch scores $K[i][j] = M[i][j] - M[i-k][j-k]$ for all valid values of $i, j, i-k, j-k$(takes $O(mn)$ time). K can be calculated by replacing values in M, without creating a new matrix, only if the values are calculated backwards.

   Also, we can calculate K directly in $O(mn)$ using a sliding window-pair approach, where, whenever we slide the window-pair, we substruct the first position score and add the new position score to get the scores for that window-pair uin O(1).

2. Now we calculate the normal global alignment score matrix between s_1 and s_2. We call this matrix F. (We also keep path pointers to help in constructing the alignment later.) Takes $O(mn)$ time.

3. Next we calculate the normal global alignment score matrix between $reverse(s_1)$ and $reverse(s_2)$. We call this matrix R. (Again, also keep path pointers to help in constructing the alignment later.)Takes $O(mn)$ time.

4. now, we just need to find the maximum i,j such that $F[i][j] + K[i][j] + R[m-(i+k)][n-(j+k)]$ is maximised. Clearly, this can be done in $O(mn)$ time. Subsequently we can construct the required alignment from the path pointer stored for F and M and the indices i and j found as optimal in this step. $O(mn)$ time.

All the steps described above takes O(mn) time and gives us the optimal k-core alignment.

Q2.

We want to modify the Hirschberg algorithm to have an affine alignment penalty with gap open g, and gap extend e. Here we have to hold onto the theme of the Hirschberg algorithm i.e. the omdfied algorithm will still be a divide and conquer algorithm where in the divide step we find

an optimal split point j and then recursively compute the alignment for the two subproblems. Which means the main modification happens in the divide step.

In the normal version of the algorithm we find an index j that minimises sum of optimal alignment cost of $s_1[0..\lfloor m/2 \rfloor]$ and $s_2[0..j]$, and that of $reverse(s_1)[0..\lceil m/2 \rceil]$ and $reverse(s_2)[0..n-j]$. We don't have to care about how those alignments look like.

However, for affine gap penalty we have to consider the fact that we can have two types of alignments for a particular split point j. First, the split point j is a mismatch in optimal alignment (which does not give us any problem) or second, the split point j happens to be part of a gap in optimal alignment. In second case, if we used the same divide step as the original algorithm then we would be wrongly counting the gap creation penalty g one extra time because both the recursive calls will count it although in the final alignment they consist of only one gap. So, the divide step has to be modified in the following way:

let,
$C(s_1, s_2)$ = minimum alignment cost of aligning s_1 and s_2
$D(s_1, s_2)$ = minimum alignment cost of aligning s_1 and s_2 while forcing a deletion at the end
$E(s_1, s_2)$ = minimum alignment cost of aligning s_1 and s_2 while forcing a deletion at the start
$F(s_1, s_2)$ = minimum alignment cost of aligning s_1 and s_2 while forcing an insertion at the end
$G(s_1, s_2)$ = minimum alignment cost of aligning s_1 and s_2 while forcing an insertion at the start
Notice, C, D, E, F, G can be calculated is O(mn) time and O(n) space for m and n length strings using the linear space score-only version of affine alignment.

modified divide step of our algorithm $AffineAlign(s_1, s_2)$:

$$\hat{j} = argmin_j \left[min \begin{cases} C[s_1[0..\lfloor m/2 \rfloor], s_2[0..j]] + C[rev(s_1)[0..\lceil m/2 \rceil], rev(s_2)[0..n-j]] & type1 \\ D[s_1[0..\lfloor m/2 \rfloor], s_2[0..j]] + E[rev(s_1)[0..\lceil m/2 \rceil], rev(s_2)[0..n-j]] - g & type2a \\ F[s_1[0..\lfloor m/2 \rfloor], s_2[0..j]] + G[rev(s_1)[0..\lceil m/2 \rceil], rev(s_2)[0..n-j]] - g & type2b \end{cases} \right]$$

The above modified divide step will give us the correct j and also we can get the type of j which is needed because the following recursive calls has to be made accordingly accordingly:

If j is a type 1 split:
The recursive calls will just be $AffineAlign(s_1[0..\lfloor m/2 \rfloor], s_2[0..j])$ and $AffineAlign(s_1[0..\lfloor m/2 \rfloor], s_2[0..j])$.

Else if j is a type 2a split:
The first recursive call will be $AffineAlign(s_1[0..\lfloor m/2 \rfloor], s_2[0..j-1])$ and append the deletion at the end of it.
The second recursive call will be $AffineAlign(rev(s_1)[0..\lceil m/2 \rceil], rev(s_2)[1..n-j])$ and add the deletion at the start of it. However, in this call, we should not penalise by g for initial deletions if there are any (other than the one we added). This can be done by adding another flag argument to the function $AffineAlign$ which tells the recursive call to take care of that.

Similarly, Else if j is a type 2b split:
The first recursive call will be $AffineAlign(s_1[0..\lfloor m/2 \rfloor - 1], s_2[0..j-1])$ and append the insertion at the end of it.
The second recursive call will be $AffineAlign(rev(s_1)[1..\lceil m/2 \rceil], rev(s_2)[0..n-j])$ and add the insertion at the start of it. However, in this call, we should not penalise by g for initial

insertions if there are any (other than the one we added). This can be done by adding another flag argument to the function $AffineAlign$ which tells the recursive call to take care of that.

The above modifications to Hirschberg algorithm, with proper base cases, will give us Affine gap alignment in $O(mn)$ time and $O(n)$ space. [Answer]

Q3.

Assuming length of the sequences are similar for convenience in writing, i.e. $m = O(n)$.
a. Writing them as tuples serves the following purposes:

- we want to process and calculate anything that has to do with the cand array in $O(log(n))$ time per iteration. Writing the values explicitly may potentially lead to spending $O(n)$ time updating/writing the array which defeats the whole purpose.
- saves space because writing them explicitly is just repetition.

b. If just a list were used then, in each iteration, updating the list would be worst case $O(n)$ time, which will make the total run time $O(n^3)$.
c.
B can be be an array of tuples but instead of just count, we need to save the start and end position of the stretch of values. i.e. insted of (index, count), we need to have (index, start, end), where end - start = count.
d. For this part, we simply check the condition for the last position of each tuple starting from the last tuple to find the first tuple whose last index satisfies the condition. If no such tuple exists, we return no such j exists.

Otherwise, we take that tuple and do a binary search inside to find the first position which satisfies the condition. The full pseudocode is shown in the next page:

P.T.O.

Algorithm 1: Search for the first position j in B for which
$score[i] - gap(j - i) < score(B[j]) - gap(j - B[j])$

Input: A, i, *score* //A is array of tuples representing B

```
k ← length(A)                                          //number of tuples
k_old ← k
while k ≠ 0 do                                         //using 1 based indexing
    if score[i] − gap(A[k].end − i)
    < score(A[k].index) − gap(A[k].end − A[k].index) then
        k−−
    else
        break                                          //end the while loop
    end if
end while
if k == k_old then
    return "No such j exist"
end if
left ← A[k+1].start; right ← A[k+1].end;
val ← A[k+1].index
while left ≤ right do                                  //binary search
    mid ← ⌊(left + right)/2⌋
    check1 ← score[i] − gap(mid − i) − score(val) − gap(mid − val)
    check2 ← score[i] − gap(mid + 1 − i) − score(val) − gap(mid + 1 − val)
    if check1 > 0 and check2 < 0 then
        return mid + 1
    else if check1 < 0 then
        right ← mid
    else
        left ← mid
    end if
end while
return A[k+1].start   //note that the above while loop must return a value
    of correct j if it is not at the start of A[k+1], in that case the loop will finish
    and now we have to return A[k+1].start
```

Q4.

Part a.

T _ G C A C C
T _ G _ A C C
T _ G A A C G
G _ G _ A C C
G _ A _ A C C
A A G _ A C C
T _ G _ A C C

note: in the above, I prioritied mismatches over 2 gaps (insertion-deletion pair).

Part b.

Let, $s(S_i, S_j)$ be the optimal pairwise alignment for strings S_i and S_j and only one optimal pairwise alignment is possible per pair. In an optimal alignment consistent with a given tree, by definition, the induced alignment $t(S_i, S_j) = s(S_i, S_j)$ for all i, j if there is an edge between S_i and S_j.

Now, suppose we get 2 different alignmentes, A and B, by picking the edges in different orders. Also, we assume we have a preference among insertion, deletion and substitutions (e.g. insertion score > deletion score > substitution score, otherwise multiple MSA is possible even for one particular order of picking edges.). Now if A and B are different, they have to have different induced alignment for some edges. Let, one such edge is between S_i and S_j. since, A and B are both MSAs consistent with the tree we shall have $t_A(S_i, S_j) = s(S_i, S_j)$ and $t_B(S_i, S_j) = s(S_i, S_j) \implies t_A(S_i, S_j) = t_B(S_i, S_j)$ Now, since, there is only one possible optimal pairwise alignment of S_i and S_j. The above means, both A and B has to give same induced alignment for the strings S_i and S_j. This applies for all possible differences between A and B. Hence, A and B cannot be different(contradiction).

$\implies$ When constructing an MSA consistent with a tree the order does not matter. [Proved]
