

**BCB 567/CprE 548 Bioinformatics I**  
**Fall 2007**  
**Homework 2 Solutions**

1. Consider the cells in which we look for the best alignment score as sets labeled  $M$ . For global alignment, we look only at the lower right cell:  $M_g = \{S[i, j] | i = n \text{ AND } j = m\}$ . For semi-global alignment, we look at the entire last row and column:  $M_s = \{S[i, j] | i = n \text{ OR } j = m\}$ . For local alignment we consider the entire table:  $M_l = \{S[i, j]\}$ . We see that  $M_g \subset M_s \subset M_l$ . Because  $A \subset B \rightarrow \text{Max}(A) \leq \text{MAX}(B)$ , we conclude  $\text{global} \leq \text{semiglobal} \leq \text{local}$ .

One final consideration is that, for any cell, the score in the local alignment table are greater than or equal to the score in the semiglobal alignment table, which is in turn greater than or equal to the score in the global alignment table. This is because in each case, additional negative values are replaced with zeros when filling out the table. Thus the difference in scores between the tables reinforce the relationships indicated above.

2. The best scoring path traveling outside of the the k-band will have exactly two gaps of total length  $2k$  and  $n - k$  matches. The first gap is used to travel outside the k-band, while the second is used to travel back to main diagonal. The total score of this path is  $2g + 2kh + (n - k)\alpha$ .
3. Initialize the first row and first column of the table with zeros. When filling in the a cell in the first  $t$  rows or first  $t$  columns of the table, use the equation for local alignment:

$$S[i, j] = \begin{cases} S[i - 1, j - 1] + \delta(i - 1, j - 1) \\ S[i, j - 1] + \gamma \\ S[i - 1, j] + \gamma \\ 0 \end{cases}$$

When filling in all other cells, use the equation for global and semi-global alignment:

$$S[i, j] = \begin{cases} S[i - 1, j - 1] + \delta(i - 1, j - 1) \\ S[i, j - 1] + \gamma \\ S[i - 1, j] + \gamma \end{cases}$$

Once the table is complete, find the maximum in the last  $t$  rows and last  $t$  columns of the table. This the score of the optimal alignment. Trace back until finding a zero in the first  $t$  rows or  $t$  columns.

4. We will consider part b. For part a,  $\sigma$  can be replaced with 0 wherever it appears. The  $O(n^3)$  solution is to treat the weight function as a general weight function:

$$W[i] = \begin{cases} i\gamma & \text{if } i < X \\ \sigma & \text{if } i \geq X \end{cases}$$

We can then run the  $O(n^3)$  algorithm for finding the best alignment using a general weight function as described in class. (The details are in the slides on affine gap alignment.)

The  $O(n^2)$  algorithm involves keeping track of two data structures.  $R_X[0..m]$  holds the maximum value for the first  $0..(i-X)$  rows of the table, when filling out row  $i$ . For simplicity of explanation, we will use a corresponding data structure  $C_X[0..n]$  to hold the maximum for the first  $(j-X)$  columns, although a single number would suffice.

We first initialize

$$R_X[j] = -\infty$$

$$C_X[i] = -\infty$$

$$S[0,0] = 0$$

$$S[0,j] = \begin{cases} j\gamma & \text{if } j < X \\ \sigma & \text{if } j \geq X \end{cases}$$

$$S[i,0] = \begin{cases} i\gamma & \text{if } i < X \\ \sigma & \text{if } i \geq X \end{cases}$$

When filling in cell  $S[i,j]$ , we use the following logic. First, if  $i \geq X$ , then we update  $R_X[j]$ :

$$R_X[j] = \max(R_X[j], S[i-X, j])$$

If  $j \geq X$ , then we update  $C_X[i]$ :

$$C_X[i] = \max(C_X[i], S[i, j-X])$$

The formula for filling the cell is:

$$S[i,j] = \max \begin{cases} S[i-1, j-1] + \delta(i-1, j-1) \\ S[i-1, j] + \gamma \\ S[i, j-1] + \gamma \\ R_X[j] + \sigma \\ C_X[i] + \sigma \end{cases}$$

When performing traceback, if we don't find a local up/left/diagonal move to make, we must search for the maximum to the left in the current row and above in the current column, starting  $X$  elements away in each case. This maximum will be the point from which we continue traceback.

Because a constant amount of work is done for each cell while filling out the table, the total runtime is  $O(nm)$ . Because we are using additional space in the order of  $n + m$  for dealing with  $X$ , the total space requirement remains  $O(nm)$ , the space required for the dynamic programming table.

I believe that if you think about applying space saving techniques to this problem, you will see that this can be done with at most  $O(n + Xm)$  space in  $O(nm)$  time. This is because you will need  $X$  rows above and below the intersection to find the point at which the path crosses between the upper and lower tables. You will also need to store  $X$  rows to update  $R_X[j]$ . I have not carefully evaluated the time this would take to see if it remains  $O(nm)$ .

5. The most obvious answer to this question is to reduce the runtime from  $O(n^2m^2)$  as described in the problem to  $O(n^2m)$ . We notice that redundant work is being done. Instead of computing an alignment between all pairs of linearizations  $(\mathcal{A}_i, \mathcal{B}_j)$ , we only compute a subset of these alignments.

We can see that if we take any alignment between  $\mathcal{A}_i$  and  $\mathcal{B}_j$ , we can break it in half at any column, such that the first half is an optimal alignment between prefixes of  $\mathcal{A}_i$  and  $\mathcal{B}_j$ , and the second half is an optimal alignment between suffixes of  $\mathcal{A}_i$  and  $\mathcal{B}_j$ . We see that we can take the first half of the alignment and place it behind the second half, producing a new optimal alignment between  $\mathcal{A}_{i'}$  and  $\mathcal{B}_{j'}$  with the same score. Moreover, there exists some break point such that  $i' = 0$ .

This implies that for all  $i, j$  there exists some  $k$  such that the score of the alignment between  $\mathcal{A}_i$  and  $\mathcal{B}_j$  is equal to the score of the alignment between  $\mathcal{A}_0$  and  $\mathcal{B}_k$ . This means, instead of looking at all pairs  $(\mathcal{A}_i, \mathcal{B}_j)$ , we can instead look at only those pairs  $(\mathcal{A}_0, \mathcal{B}_j)$ . As a result we only do  $n$  alignments, for a total runtime of  $O(n^2m)$ .

One might be tempted to try to solve the problem in  $O(nm)$  time by doing a semiglobal or local alignment between  $\mathcal{A}_0$  and  $\mathcal{B}_0$  appended to  $\mathcal{B}_0$ . However, there is no way to guarantee that you will use exactly  $m$  characters from  $\mathcal{B}_0\mathcal{B}_0$  during the alignment. While in practice you will likely find an alignment similar to the method above, especially for two similar strings, this method is not correct in general. Once the semi-global or local alignment is found, you can use the center point of the alignment in each string to choose new linearizations of each string to use for global alignment. The final global alignment proceeds as normal. Describing this methods adequately did result in full credit.