

BCB 567/CprE 548 Bioinformatics I
 Fall 2007
 Homework 1 Solutions

- 1.
- 2.
3. Run the algorithm for filling in the dynamic programming table, keeping track of exactly two rows, as per the method described in class. In addition, keep track of the current maximum, max , and the coordinates of the cell that corresponds to the maximum: r and t . To update max, r , and t , after calculating $S[i, j]$, we must make the following check: if $S[i, j] > max$ then $\{max \leftarrow S[i, j]; r \leftarrow i; t \leftarrow j\}$. Once we have filled in the entire table, (r, t) will be the coordinates of the maximum value in the table.

To find the starting cell of the best local alignment, we will run the dynamic programming algorithm backwards, keeping track of exactly two rows at a time, on the subtable $S[0\dots r, 0\dots t]$. This means that we initialize the last row and last column and use the reverse recursion, as described in class.

We stop filling the table when $S[i, j] = max$. We have now found r, t, i , and j . The substrings $\mathcal{A}[i + 1, r]$ and $\mathcal{B}[j + 1, t]$ are aligned in the optimal local alignment.

Because we only store the strings, two rows of the dynamic programming table at a time, and a constant number of additional elements to keep track of the maximum, the space requirement of the algorithm is $O(n + m)$. The running time is $O(nm)$ because we do a constant amount of work for each cell, and there are nm total cells.

4. (a) True. The path string is exactly the length of the alignment, because each character in the path string corresponds exactly to increasing the alignment by 1 column.
- (b) We can construct the path string in the following manner. I will use the notation (j_i, k_i) for intersection i , $0 \leq i \leq n$. Each intersection corresponds to a vertical or horizontal move. For an intersection of the form (j_i, j_i) , we write a V . For an intersection of the form $(j_i, j_i + 1)$, we write a D . Between each V and D , there are zero or more H s. To calculate out the number of H s between intervals (j_i, k_i) and (j_{i+1}, k_{i+1}) , we subtract $j_{i+1} - k_i$. The number of initial H s in the path string is the value j_0 . The number of trailing H s is the difference $m - k_n$.
- (c) Keep track of an index i into \mathcal{A} , an index j into \mathcal{B} , and an index k into \mathcal{A}_A and \mathcal{B}_A . Initialize i, j , and k to 0. Run the following logic.
 - “D”: $\mathcal{A}_A[k] \leftarrow \mathcal{A}[i]; \mathcal{B}_A[k] \leftarrow \mathcal{B}[j]; i \leftarrow i + 1; j \leftarrow j + 1; k \leftarrow k + 1$
 - “V”: $\mathcal{A}_A[k] \leftarrow \mathcal{A}[i]; \mathcal{B}_A[k] \leftarrow \text{'-'}; i \leftarrow i + 1; k \leftarrow k + 1$
 - “H”: $\mathcal{A}_A[k] \leftarrow \text{'-'}; \mathcal{B}_A[k] \leftarrow \mathcal{B}[j]; j \leftarrow j + 1; k \leftarrow k + 1$

- (d) Shortest Path String: $2n$. Longest Path String: $4n$. Intersection List: $n(\lceil \log_2(n) \rceil + 1)$.
 - (e) Positives of the intersection list: Natural representation for use with space saving, can reconstruct parts of the path easily, same size for all possible alignments between two strings. Positives of path string: Takes the least amount of space, easy to use to write out alignment.
5. We call all the matching characters in the alignment the common subsequence *induced by* the alignment. Therefore there is a many to one mapping between alignments and subsequences. In order to solve the Longest Common Subsequence problem, we wish to concurrently maximize the number of matches while we maximize the alignment score. The following simple proof shows that if we do this, we will be solving LCS using alignment.

Assume, for a contradiction, that the LCS is longer than the number of matches in the optimal alignment. We can construct a new alignment by aligning all characters in the LCS and filling in the rest of the alignment with gaps or mismatches. Because this new alignment has more matches than our original optimal alignment, we have found a contradiction. Therefore, any choice of parameters that maximizes the number matches while concurrently maximizing the score can be used to find the LCS.

Assume $\beta \leq 2\gamma$. If beta is strictly less than two gamma, then the optimal alignment can never contain a mismatch. This is because any mismatching column can be replaced by two gaps to increase the score. If beta is equal to two gamma, then any optimal alignment that contains mismatches can be recast as an alignment with the same score that contains no mismatches. For this reason, we can assume that the optimal alignment contains no mismatches.

Let X be the number of matches and Z be the number of gaps in an optimal alignment. Then the following equations hold:

$$\begin{aligned} E(\mathcal{A}_A, \mathcal{B}_A) &= X\alpha + Z\gamma \\ n + m &= 2X + Z \\ n + m - 2X &= Z \end{aligned}$$

Substituting, we have:

$$E(\mathcal{A}_A, \mathcal{B}_A) = X\alpha + (n + m - 2X)\gamma$$

Arranging terms:

$$E(\mathcal{A}_A, \mathcal{B}_A) = X(\alpha - 2\gamma) + (n + m)\gamma$$

Therefore, as long as $\beta \leq 2\gamma$ and $\alpha > 2\gamma$, optimizing the alignment score will concurrently optimize the number of matches, resulting in an LCS between the two strings. For example $\alpha = 1, \beta = 0, \gamma = 0$ will work.

The LCS is not unique, for example $\mathcal{A} = \text{“ABCD”}$ and $\mathcal{B} = \text{“CDAB”}$ have two LCSs of length 2.