6.047 Recitation 2 Notes

Sitara Persad (2017) *

September 15, 2017

Topics For Today

• Linear Space Alignment (via Hirschberg’s Algorithm)
• Multiple Alignment
• Affine Gap Penalty for Sequence Alignment

1 Alignment Review

Why align sequences?

Sequence alignments can reveal many things about the functional, structural or evolutionary significance of regions of DNA, RNA or protein.

Examples:

• The degree of similarity between regions of amino acids can tell us how well that region is conserved, which can provide clues about the structural or functional significance of that region.

• The DNA sequence alignment of multiple species can give us clues about how long ago they diverged from a common ancestor, and help us to build us a phylogenetic tree describing the evolutionary history of those species.

*adapted from material by Melissa Gymrek (2011), Rachel Sealfon (2012) and Max Wolf(2013)
2 Linear Space Alignment

Problem:

We want to find the best alignment of two sequences, say of length \( m \) and \( n \), where \( m < n \). However, in the case where \( m \) and \( n \) are large, the traditional approach of constructing an \( n \times m \) matrix requires too much storage.

We can solve this problem using Hirschberg’s algorithm, which requires \( O(n) \) space, whilst maintaining \( O(mn) \) time.

Summary of Approach

- Note that the optimal path must go through the middle column at some point (in order to get from the left-most column to the right-most column).

- Note that the optimal alignment from top left to bottom right is the same as the optimal alignment from bottom right to top left.

- Thus, we can compute the best alignment from the top left to the middle column, and the best alignment from the bottom right to the middle column. We compute the forward pass scores, and store the middle column pointers, and scores at row \( i \), denoted by \( F(i) \). We compute the reverse pass scores, and store the middle column pointers, and scores at row \( i \), denoted by \( (i) \).

The best path is the path which passing through \( i^* \) such that \( i^* = \arg \max \ F(i) + R(i) \).
Recall that we can compute the optimal score in linear time, by storing only the column we are populating, and the one immediately before it.

Thus, if we store only a the pointers for the middle column where we cross to the other quadrant, we only require linear space overall.

How much work do we have to do? Well, we must do $O(mn)$ work on the matrix each iteration. In each iteration, we approximately halve the size of the matrix that we are working on.

Thus, we have a sum of $O(mn) + O(mn/2) + O(mn/4) + ... = O(2mn) = O(mn)$

**Example of Approach**

See In-Class exercise on Linear Space Alignment

## 3 Multiple Alignments

The methods for sequence alignment we have been discussing thus far have worked well for aligning two sequences against each other. However, suppose we would like to find the optimal alignment of 3 sequences. How might we proceed.

One approach is to construct a three-dimensional matrix and run the Needleman-Wunsch algorithm. Recall that when we align two sequences $S$ and $T$, we choose the maximum of three possibilities for the final position of the alignment (sequence $T$ aligned against a gap, sequence $S$ aligned against a gap or sequence $S$ aligned against sequence $T$).

For three sequences $S$, $T$, and $U$, there are seven possibilities for the final position of the alignment - three ways to have one gap in the final position, three ways to have two gaps, and $S$, $T$, and $U$ all aligned.

$$F_{i,j,k} = \max \left\{ \begin{array}{l} F_{i-1,j,k} + s(S_i, -, -) \\ F_{i,j-1,k} + s(-, T_j, -) \\ F_{i,j,k-1} + s(-, -, U_k) \\ F_{i-1,j-1,k} + s(S_i, T_j, -) \\ F_{i-1,j,k-1} + s(S_j, -, U_k) \\ F_{i,j-1,k-1} + s(-, T_j, U_k) \\ F_{i-1,j-1,k-1} + s(S_i, T_j, U_k) \end{array} \right\}$$

However, this approach is exponential in the number of sequences. If we have $k$ sequences of length $n$, computing the optimal alignment using a $k$-dimensional DP matrix takes $O(2^k n^k)$ time. Because a $k$-cube has $2^k$ vertices, we have to take the maximum over $2^k - 1$ neighbouring positions for each entry in the scoring matrix.
This exponential runtime is not feasible. Thus, for multiple sequence alignment, we usually use a heuristic, which may provide a non-optimal solution.

### 3.1 Aligning many sequences

One commonly used approach for multiple sequence alignment is called **progressive multiple alignment**. The insight is that we build the overall sequence alignment starting with a seed alignment of a pair of sequences. We proceed by performing pairwise alignments of individual sequences or sequence alignments until we have generated a full multiple alignment. The running time of this heuristic is polynomial in the number of sequences. However, we are no longer guaranteed that the final alignment is optimal.

To improve the final alignment, sequences are often added to the alignment in order of increasing dissimilarity, with the most similar pair of sequences aligned first. (The motivation for this heuristic is that the alignment is more likely to be reliable for closely related sequences). For example, one heuristic used to construct the guide tree which describes the hierarchy of similar sequences, is the distance based on the number of identical two letter sub-sequences.

Still, we have no guarantees on the optimality of the final alignment. For example, if our initial alignment is globally suboptimal, the overall multiple alignment will also be flawed.

We know how to align pairs of sequences, but how can we align a sequence to a multiple alignment? One possibility is to perform pairwise alignments of the new sequence with each sequence already in the group. (We assume that any position in the group alignment that is already a gap will remain one). Then we can add the new sequence on to the group alignment based on the best pairwise alignment with the sequences currently in the group. (This approach was devised by Feng and Doolittle, 1987). Alternately, we can devise a function for scoring the alignment of a sequence with an alignment. (Scoring functions for the alignment of a sequence to an alignment are often based on the pairwise sum of the scores at each position).

Designing better multiple sequence alignment tools is an active area of research. There are many different tools and algorithms for multiple sequence alignment. One recent paper even took a crowdsourcing approach to the problem and created a flash game for users to manually improve multiple sequence alignments: (Kawrykow et al., 2012; [http://phylo.cs.mcgill.ca/](http://phylo.cs.mcgill.ca/))

### 4 Affine gap penalties

**Motivation:** So far we have been using a linear gap model, which treats all gaps equally when scoring alignments. However, often the generation of a gap is a single event,
it is more likely that an existing gap will be extended (due to things like RNA polymerase slippage) rather than that a new gap will be opened.

Consider the following two alignments:

<table>
<thead>
<tr>
<th>Table 1: Fixed Gap Cost Alignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>A A A G A A T T C A</td>
</tr>
<tr>
<td>A - A - A - T - C A</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 2: Affine Gap Penalty Alignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>A A A G A A T T C A</td>
</tr>
<tr>
<td>A A A - - - - T C A</td>
</tr>
</tbody>
</table>

Both have the same number of gaps and matches, and thus would be equally likely under a fixed gap cost model. However, the second alignment is often more plausible, and would be preferred by an affine gap penalty model.

Therefore, we should penalize multiple gaps more than longer gaps.

We can define an **affine gap penalty** as follows:

\[
\text{penalty}(n) = \text{gap} + (n) \times \text{extend}
\]

Where \(\text{penalty}(n)\) is the penalty for a \(n\)-position long gap and:

- **open** is the gap open penalty, or the one-time cost of introducing a DNA break
- **extend** is the gap extend penalty, or the linear cost increment for increasing the number of gaps.

![Figure 2](image)

Figure 2: Affine gap penalties have a fixed cost for opening a gap and a linear cost increment for increasing gap size.

How can we extend our dynamic programming to accommodate affine gap penalties and still use \(O(mn)\) running time and space? A greedy extension of the standard global algorithm
will not work (e.g. looking at previous traceback pointers when making each choice).
Sometimes we should make a locally suboptimal choice (open a gap) because it will pay off in the future (resulting in a better-scoring longer single gap instead of multiple smaller gaps).

We will need to keep track of more state. In scoring a single entry in our matrix, we need to remember an extra piece of information:

- Are we continuing a gap in s? (if not, start is more expensive)
- Are we continuing a gap in t? (if not, start is more expensive)
- Are we continuing from a match between $S_i$ and $T_j$?

We can encode this information in three different matrices. For element $(i, j)$, we use three variables:

- $A_{i,j}$: best alignment of $S_1 \ldots S_i$ and $T_1 \ldots T_j$ that aligns $S_i$ with $T_j$.
- $B_{i,j}$: best alignment of $S_1 \ldots S_i$ and $T_1 \ldots T_j$ that aligns gap with $T_j$.
- $C_{i,j}$: best alignment of $S_1 \ldots S_i$ and $T_1 \ldots T_j$ that aligns $S_i$ with gap.

**Update Rule:**

When $S_i$ and $T_j$ are aligned (diagonal movement from all matrices to $A$):

$$A_{i,j} = \max \begin{cases} A_{i-1,j-1} + \text{score}(S_i, T_j) \\ B_{i-1,j-1} + \text{score}(S_i, T_j) \\ C_{i-1,j-1} + \text{score}(S_i, T_j) \end{cases}$$

When $T_j$ aligns with a gap in $S$ (vertical movement from all matrices to $B$):

$$B_{i,j} = \max \begin{cases} A_{i,j-1} + \text{open} + \text{extend} \\ B_{i,j-1} + \text{extend} \\ C_{i,j-1} + \text{open} + \text{extend} \end{cases}$$

When $S_i$ aligns with a gap in $T$ (horizontal movement from all matrices to $C$):

$$C_{i,j} = \max \begin{cases} A_{i-1,j} + \text{open} + \text{extend} \\ B_{i-1,j} + \text{open} + \text{extend} \\ C_{i-1,j} + \text{extend} \end{cases}$$

**Initialization:**

Note that the term $A_{i,j}$ can be interpreted as the score of best alignment between $i$ characters of $S$ and $j$ characters of $T$ that ends in a match.

The term $B_{i,j}$ is score of best alignment between $i$ characters of $S$ and $j$ characters of $T$ that ends in a gap in $S$. 


The term $C_{i,j}$ is score of best alignment between $i$ characters of $S$ and $j$ characters of $T$ that ends in a gap in $T$.

Bearing this in mind, we set:

- $A_{0,0} = 0$
- $B_{0,0} = C_{0,0} = \text{open}$
- $B_{0,j} = \text{open} + j \times \text{extend}$, $C_{i,0} = \text{open} + i \times \text{extend}$
- $B_{i,0} = C_{0,j} = \infty$

**Termination**: When we’re done, we look for the maximum over all three arrays $\max(A_{m,n}, B_{m,n}, C_{m,n})$. To find the alignment, follow the arrows back, skipping from matrix to matrix.

**Example of Multiple Alignment**:

See In-Class exercise on Multiple Alignment.

### 5 References:

For more information, see the following references:

- Course textbook, ch.2-3
- Durbin, Eddy, Krogh, Mitchison, Biological Sequence Analysis, 1998. Ch 2 (Sequence alignment) and ch 6 (multiple sequence alignment).