16.5 Global alignment: Needleman-Wunsch algorithm

We can't calculate all possible alignments of a pair of sequences. (There are $\binom{n+m}{m}$ possible alignments for a pair of sequences of length n and m.) We use *dynamic programming* approaches that allow is to quickly calculate the best possible alignment (that is, the one that gives us the highest score).

Dynamic programming is technique of solving complex problems by reducing them to a number of much simpler subproblems that we can easily solve then re-assemble to find the answer to the complex problem. It uses the structure of the problem itself and so is only applicable to problems that possess a certain type of structure and to which we can apply the Principle of Optimality: "a sub-optimal solution of a sub-problem cannot be part of optimal solution of original problem".

The the alignment context, the principle of optimality holds in that if we know the score of an optimal alignment of length k then the score of the first $k-1$ parts of the alignment must be optimal.

To see why this is, let $F_{i,j}$ be the score of the optimal alignment between $x[1:i]$ and $y[1:j]$. Let $s(x_i, y_j)$ be the score for matching residue x_i to residue y_j and assume a linear gap penalty (so that the penalty for adding the gap $(x_i, -)$ or $(-, y_j)$ is d). The optimal alignment up to x_i, y_j either has x_i and y_j aligned, or y_j aligned to a gap or x_i aligned to a gap. For example, it looks like

$$\begin{array}{cccc} I & G & A & y_j \\ L & G & V & x_i \end{array} \quad \text{or} \quad \begin{array}{cccc} I & G & A & y_j \\ V & x_i & - & - \end{array} \quad \text{or} \quad \begin{array}{cccc} G & A & y_j & - \\ L & G & V & x_i \end{array}$$

In any case, the first part of the alignment must be optimal (If that first $k-1$ parts were not optimal, we'd find the optimal alignment for the first parts, add the kth bit on in one of the 3 possible ways and have a better alignment for the first k parts, contradicting our assumption that our original alignment was optimal for length k). Thus, if we know score of the best alignment for $k-1$ parts, we can extend it to the best alignment for k parts. This observation allows us to write the problem as a recurrence relation:

$$F(i,j) = \max \begin{cases} F(i-1, j-1) + s(x_i, y_j), \\ F(i-1, j) - d, \\ F(i, j-1) - d. \end{cases}$$

The first case we have matched x_i, y_j, the second case we have matched x_i to a gap and the final case we have matched y_j to a gap.

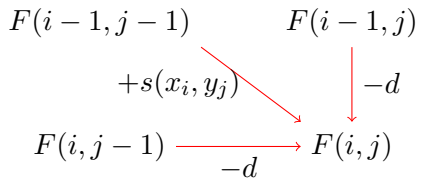
So we want to find $F_{n,m}$ and we have the boundary conditions $F(0,0) = 0$ (start at 0), $F(i,0) = -id$ and $F(0,j) = -jd$ (linear gap penalties for initial gaps).

Note that all the above can be phrased in terms of mismatches and penalties, rather than matches and scores. To do so, simply reverse the signs of the scores and take minimums rather than maximums.

If we use a naive recursive method to calculate $F(n,m)$, we still get an exponential number of calls. But notice that there are only $m \times n$ possible combinations we need to

calculate. We can do this in a tabular manner, calculating the matrix F from the top left to the bottom right in a progressive fashion.

To calculate the (i, j)th entry, we only need to know the 3 entries to the left and above it. The (i, j)th entry s then a maximum over 3 numbers. We keep a pointer to indicate which cell the (i, j)th entry was derived from.

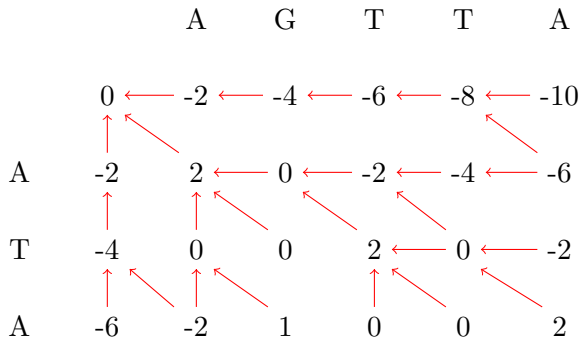


Once we have filled out the matrix, we trace back from $F(n, m)$, following the path that led us here. That is, the score at the (n, m)th position came from one of position $(n-1, m)$, $(n, m-1)$, or $(n-1, m-1)$ by adding a gap or a match. We move to whichever position it came from either adding the gap or the match in the process. In doing so, we build up the alignment from right to left, eventually arriving at $F(0, 0)$ at which point we can reverse the alignment to get the full

Our method is thus based on three things: a recurrence relation, tabular computing and then traceback. These methods turn an what is naively an exponential algorithm into a quadratic algorithm ($O(nm)$).

Example: Align `x = ATA` and `y = AGTTA` with the following scores: the purines are A and G, while the pyrimidines are C and T. Let $s(a, b) = 2$ if $a = b$, 1 if a is purine and b is a purine or a is a pyrimidine and b is a pyrimidine, and -2 if a is a purine and b is a pyrimidine or vice versa. Let the gap score be $d = -2$.

Solution: Filling out the matrix and drawing arrows to show where each entry is derived from we get the following:



The score of the best alignment is given in the bottom right: $F(3, 5) = 2$. To find the alignment with the best score, we traceback from this point. At $F(2, 4)$ there are two choices that produce the same score. One alignment, found by following the arrow from

$F(2,4)$ to $F(1,3)$ is

$$\begin{array}{ccccc} A & - & - & T & A \\ A & G & T & T & A \end{array}$$

while the other is obtained by following the arrow from $F(2,4)$ to $F(2,3)$ and looks like

$$\begin{array}{ccccc} A & - & T & - & A \\ A & G & T & T & A \end{array}$$

□

16.6 Elements of an alignment algorithm

We emphasise that these dynamic programming algorithms for sequence alignment are based on following elements:

- a recurrence relation for the quantity we are trying to optimise, including specification of the boundary conditions,
- tabular computing to efficiently calculate the recurrence, and
- traceback (includes specifying rules for where to start and stop the traceback).

By altering the recurrence relation, the boundary conditions or the traceback, we will find different types of best alignment. Local alignment is the most common form and is defined below.

16.7 Local Alignment: Smith Waterman algorithm

The Needleman-Wunsch algorithm looks only at completely aligning two sequences. More commonly, we want to find the best alignment for some subsequence of two sequences. This is the local alignment problem.

The resulting algorithm that solves this problem is very similar to the one that solve the global alignment problem. We derive it as follows. Redefine $F(i,j)$ to be the score of the best suffix alignment of $x_1x_2\ldots x_i$ and $y_1y_2\ldots y_j$, where a *suffix alignment* is any alignment of $x_sx_{s+1}\ldots x_i$ and $y_ry_{r+1}\ldots y_i$ for some $1 \leq s \leq i$ and $1 \leq r \leq j$. Note that this suffix alignment could be the empty alignment with score 0.

We thus get the recursion

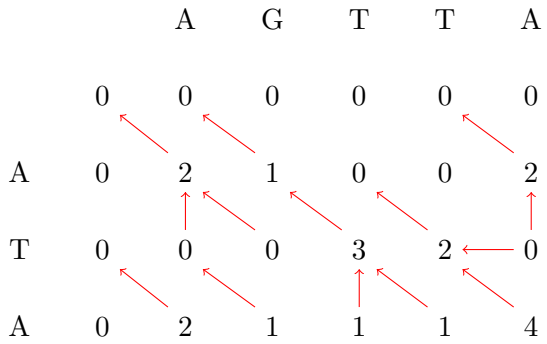
$$F_{i,j} = \max \begin{cases} 0 \\ F(i-1,j-1) + s(x_i, y_i), \\ F(i-1,j) - d, \\ F(i,j-1) - d. \end{cases}$$

So instead of a letting a score for an alignment to go negative, we start a new alignment. To find the best subsequence alignment, then, we simply look for the best score and

trace it back until we hit a 0. Note than we can now start and finish anywhere in the matrix.

Example: Find the best local alignment using the score matrix and sequences given in the previous example: `x = ATA` and `y = AGTTA`

Solution: Fill out the matrix, drawing an arrow when a cell has a predecessor to get the following.



The score of the highest scoring local alignment is the largest entry in this matrix. We find this at $(5, 3)$ where $F(5, 3) = 4$. The sub alignment is found by tracing back from that cell and stopping at the first cell with no predecessor (or at the first 0 encountered). This produces the local alignment

$T \quad A$
$T \quad A$

□