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 $x_1x_2 \dots x_i$ and $y_1y_2 \dots y_j$, where a *suffix alignment* is any alignment of $x_sx_{s+1} \dots x_i$ and $y_ry_{r+1} \dots y_i$ 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.

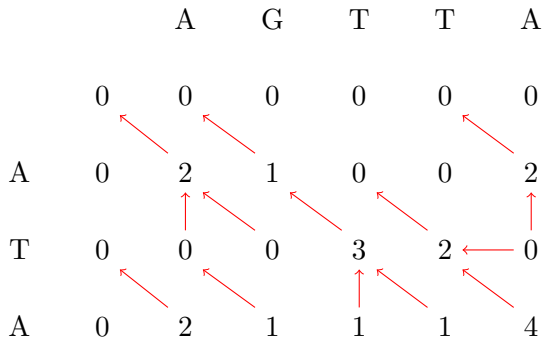
We thus get the recursion

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

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

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

□

16.7.1 Overlap matches

As an example of how easy it is to establish different types of alignment algorithm we consider a special type of alignment known as an overlap alignment.

When we expect one sequence to completely contain another or that they overlap, we want a global type alignment that does not penalize the unmatched overhanging ends. The boundary conditions are $F(i,0) = F(0,j) = 0$ for all i, j, the recurrence relation is just the global recurrence and we start the traceback at the position on the boundary where a maximum is achieved, $F(i,m)$ or $F(n,j)$. The traceback stops when the other border is reached, $F(i,0)$ or $F(0,j)$.

16.8 Pairwise alignment with non-linear gap penalties

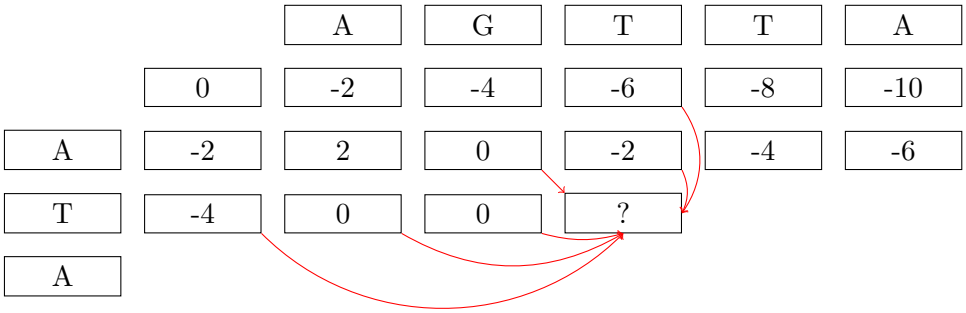
In our pairwise alignment discussion, we only considered linear gap penalties. As we noted earlier, linear penalties are a poor model for biological sequence data where we expect gaps (that is, insertions or deletions) to be quite rare but if there is a gap it may be multiple bases in length. Thus, an affine penalty, which penalises the start of the gap more heavily than any extension to the gap is favoured.

For an arbitrary gap penalty, $\gamma(k)$, we can continue to use a similar dynamic programming approach as before, but a direct adoption of that approach results in a much slower algorithm. Let's investigate: With a general gap penalty, $\gamma(k)$, the recurrence relation becomes

$$F_{i,j} = \max \begin{cases} F(i-1, j-1) + s(x_i, y_j), \\ F(k,j) + \gamma(i-k), & k = 0, \ldots, i-1, \\ F(i,k) + \gamma(j-k), & k = 0, \ldots, j-1. \end{cases}$$

This means that to calculate the value of each cell in the matrix $F(i,j)$ we need to consider $i + j + 1$ other cells — the i previous cells in the row, the j previous cells in the column, and the one adjacent diagonal cell — rather than the 3 as we had with the linear gap penalty (see Figure below). This results in a $O(n^3)$ algorithm rather than a $O(n^2)$

To calculate an unknown cell of F, the scores for gaps of all possible lengths need to be calculated meaning that a calculation for each previous cell in the row and column needs to be made:



16.9 Alignment with affine gap scores

In the case of an affine gap score (which has the form $\gamma(k) = -d-(k-1)e$) it turns out that we can, once again, construct a $O(n^2)$ dynamic programming algorithm to solve the alignment problem. The only difficulty is that we now have to keep track of 3 possible states corresponding to 3 cases:

1. Let $M(i,j)$ be the best score of the alignment up to (i,j) given that x_i is aligned to y_j. This case looks like
   A C C x_i
   A C G y_j
2. Let $I_x(i,j)$ be the best score of the alignment up to (i,j) given that x_i is aligned to a gap. This case looks like
   A C C x_i
   G y_j - -
3. Let $I_y(i,j)$ be the best score of the alignment up to (i,j) given that y_j is aligned to a gap. This case looks like
   C C x_i -
   A C G y_j

Given those definitions, and assuming that a gap cannot directly follow an insertion (that is, we can't go directly from I_x to I_y or vice versa), we have the following recurrence relations:

$$M(i,j) = \max \begin{cases} M(i-1,j-1) + s(x_i, y_j), \\ I_x(i-1,j-1) + s(x_i, y_j), \\ I_y(i-1,j-1) + s(x_i, y_j); \end{cases}$$

$$I_x(i,j) = \max \begin{cases} M(i-1,j) - d, \\ I_x(i-1,j) - e; \end{cases} \quad \text{and,}$$

$$I_y(i,j) = \max \begin{cases} M(i,j-1) - d, \\ I_y(i,j-1) - e. \end{cases}$$

It should be clear that we can calculate this recursion efficiently using tabular computation where we have 3 arrays: one for each of M, I_x and I_y. We can use a similar back-tracking mechanism to find the best alignment once we have calculated the scores.

This results in an quadratic time and space algorithm once again but the coefficient of the quadratic term is greater for this algorithm than the linear gap penalty one. For example, the space requirement here is $3n^2$ while it is only n^2 with a linear gap penalty.

The above recursions can be very neatly represented as a *finite state automaton*, or FSA. In an FSA, each of the three possibilities, match, insertion in x or insertion in y, corresponds to a state (drawn as circles).

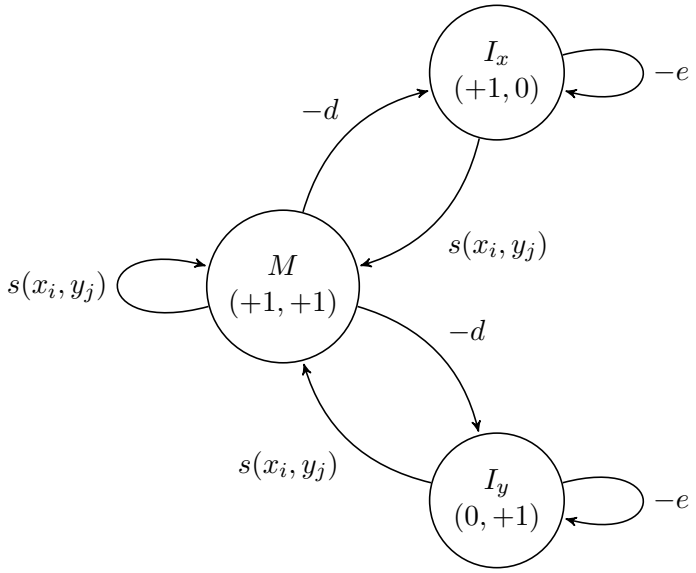


Figure 4: A finite state automaton describing the affine gap alignment recurrence relation. The pairs of numbers below the state names indicate how we increment the position in sequence x and y.

The transitions each carry a score, as indicated next to the arrow.

The new value for the state variable at (i, j) is the maximum of the scores corresponding to the transitions coming into the state. Each transition score is given by is given by the value of the source state at the offsets specified by the $\delta(i, j)$ pair of the target state plus the specified score increment.

An alignment corresponds to a path through the states.

```
V  L  S  P  A  D  -  K
H  L  -  -  A  E  S  K

m  m  Ix Ix m  m  Iy m
```

These automata are known in computer science as Moore machines.

We've already seen one type of FSA: a Markov chain can be represented as a stochastic FSA. We'll look at another stochastic FSA, the hidden Markov model or HMM, shortly.

16.10 Linear space alignment

If sequences are large, even a quadratic algorithm can be difficult to work with. We can't improve the speed of the algorithm but we can reduce the amount of memory we need (currently at that is $O(n^2)$ too).

If all we require is the *score* of the best alignment, we immediately see that we don't need to keep the whole matrix until the end of the alignment. In the case of global alignment, the score of the best alignment is given by the entry $F(m, n)$. To calculate any score in the ith row, all we need to know is the $(i-1)$th row, so we only need to keep the a single row of the matrix in memory. A similar argument can be made for the score of the best local alignment.

If we actually want the best alignment, it turns out that we can still produce a linear space algorithm. We employ a *divide and conquer* approach. Suppose we could find a cell, (i^*, j^*) what we knew to lie on the optimal alignment. Then we could divide the alignment problem into two halves: from $(0, 0)$ to (i^*, j^*) then from (i^*, j^*) to (m, n). In the best case, this reduces the amount of storage space require by 2. This process can be iterated, so (i^{**}, j^{**}) is found to reduce the space required for the $(0, 0)$ to (i^*, j^*) section and so on. It turns out that given an i^*, a suitable j^* can be found. We omit the details here (they are not too difficult) but details and references are given towards the end of chapter 2 in the Durbin et al book.