2 Pairwise alignment

FASTA

Another widely used heuristic sequence searching package is FASTA [Pearson & Lipman 1988]. It uses a multistep approach to finding local high scoring alignments, starting from exact word matches, through maximal scoring ungapped extensions, to finally identify gapped alignments.

The first step uses a lookup table to locate all identically matching words of length kmatch between the two sequences. For proteins, kmatch is typically 1 or 2; for DNA it may be 4 or 6. It then looks for diagonals with many mutually supporting word matches. This is a very fast operation, which for example can be done by sorting the matches on the difference of indices (i−j).

The best diagonals are pursued further in step (2), which is analogous to the hit extension step of the BLAST algorithm, extending the exact word matches to find maximal scoring ungapped regions (and in the process possibly joining together several seed matches).

Step (3) then checks to see if any of these ungapped regions can be joined by a gapped region, allowing for gap costs. In the final step, the highest scoring candidate matches in a database search are realigned using the full dynamic programming algorithm, but restricted to a region of the dynamic programming matrix forming a band around the candidate heuristic match.

Because the last stage of FASTA uses standard dynamic programming, the scores it produces can be handled exactly like those from the full algorithms described earlier in the chapter. There is a tradeoff between speed and sensitivity in the choice of the parameter kmatch: higher values of kmatch are faster, but more likely to miss true significant matches. To achieve sensitivities close to those of full local dynamic programming for protein sequences it is necessary to set kmatch ≈4.

2.6 Linear space alignments

Aside from time, another computational resource that can limit dynamic programming alignment is memory usage. All the algorithms described so far calculate score matrices such as F(i, j), which have overall size nm, the product of the sequence lengths. For two protein sequences, of typical length a few hundred residues, this is well within the capacity of modern desktop computers; but if one or both of the sequences is a DNA sequence tens or hundreds of thousands of bases long, the required memory for the full matrix can exceed a machine’s physical capacity. Fortunately, we are in a better situation with memory than speed: there are techniques that give the optimal alignment in limited memory, of order n+m rather than nm, with no more than a doubling in time. These are commonly referred to as linear space methods. Underlying them is an important basic technique in pairwise sequence dynamic programming.

In fact, if only the maximal score is needed, the problem is simple. Since the recurrence relation for F(i, j) is local, depending only on entries one row back, we can throw away rows of the matrix that are further than one back from the current point. If looking for a local alignment we need to find the maximum score in the whole matrix, but it is easy to keep track of the maximum value as the matrix is being built. However, while this will get us the score, it will not find the alignment; if we throw away rows to avoid O(nm) storage, then we also lose the traceback pointers. A new approach must be used to obtain the alignment.

Let us assume for now that we are looking for the optimal global alignment, using linear gap scoring. The method will extend easily to the other types of alignment. We use the principle of divide and conquer.

Let u = ⌊21/2⌋, the integer part of 21/2. Let us suppose for now that we can identify a v such that cell (u, v) is on the optimal alignment, i.e. v is the row where the alignment crosses the i = u column of the matrix. Then we can split the dynamic programming problem into two parts, from top left (0, 0) to (u, v), and from (u, v) to (n, m). The optimal alignment for the whole matrix will be the concatenation of the optimal alignments for these two separate submatrices. (For this to work precisely, define the alignment not to include the origin.) Once we have split the alignment once, we can fill in the whole alignment recursively, by successively halving each region, at every step pinning down one more aligned pair of residues. This can either continue down until sequences of zero length are being aligned, which is trivial and means that the region is completely specified, or alternatively, when the sequences are short enough, the standard O(n^2) alignment and traceback method can be used.

So how do we find v? For i > u let us define c(i, j) such that (u, c(i, j)) is on the optimal path from (1, 1) to (i, j). We can update c(i, j) as we calculate F(i, j). If (j', j) is the preceding cell to (i, j) from which F(i, j) is derived, then set c(i, j) = j' if i = u, else c(i, j) = c(i, j'). Clearly this is a local operation, for which we only need to maintain the previous row of c, just as we only maintain the previous row of F. We can now read out from the final cell of the matrix the value we desire: v = c(n, m).

As far as we are aware, this procedure for finding v has not been published by any of the people who use it. A more widely known procedure first appeared in the computer science literature [Hirschberg 1975] and was introduced into computational biology by Myers & Miller [1988], and thus is usually called the Myers–Miller algorithm in the sequence analysis field. The Myers–Miller algorithm does not propagate the traceback pointer c(i, j), but instead finds the alignment midpoint (u, v) by combining the results of forward and backward dynamic programming passes at row u (see their paper for details). Myers–Miller is an elegant recursive algorithm, but it is a little more difficult to explain in detail. Waterman [1995, p. 211] gives a third linear space approach. Chao, Hardison & Miller [1994] give a review of linear space algorithms in pairwise alignment.