6.2 Sequence alignment algorithms

6.2.1 Dot-matrix analysis

The first computer aided sequence comparison is called "dot-matrix analysis" or simply dot-plot. The first published account of this method is by Gibbs and McIntyre (1970 The diagram, a method for comparing sequences. Eur. J. Biochem 16: 1-11). Briefly, this method involves constructing a matrix with one of the sequences to be compared running horizontally across the bottom, and the other running vertically along the left-hand side. Each entry of the matrix is a measure of similarity of those two residues on the horizontal and vertical sequence. In the Gibbs and McIntyre paper, they use the simplest scoring system, distinguishes only between identical (dots) and non-identical (blank) residues. However, one can also use graded measures that give chemically similar pairs of bases higher similarity scores such as the BLOSUM and PAM matrices and enter a dot whenever the similarity exceeds a prescribed value.

Similar sequences tend to have many identical or chemically related residues along the main diagonal; hence conspicuous diagonal runs of dots signal regions of similarity. Simple as it is, dot matrix analysis is still a popular tool for researchers to visually inspect the similarity between two sequences. It is often used as a first examination. From its output, the researcher can pick out regions from the two sequences on which more detailed alignment will be performed.

Maizel and Lenk (1981 "Enhanced Graphic Matrix Analysis of Nucleic Acid and Protein Sequences", Proc. Natl. Acad. Sci. USA 78; 7665-7669) generalize the original ideas of Gibbs and McIntyre. At every base of the two sequences, a window of fixed size is laid down. A dot will be entered in the matrix if the total similarity score of the two windowed fragments exceeds a prescribed threshold. Their algorithm is implemented in the GCG program "compare". The output of compare can be fed into the "dot-plot" program to draw the dot-matrix. Figure 6.2 is the dot-plot output of the amino acid sequences of the human hemoglobin α and β chains.

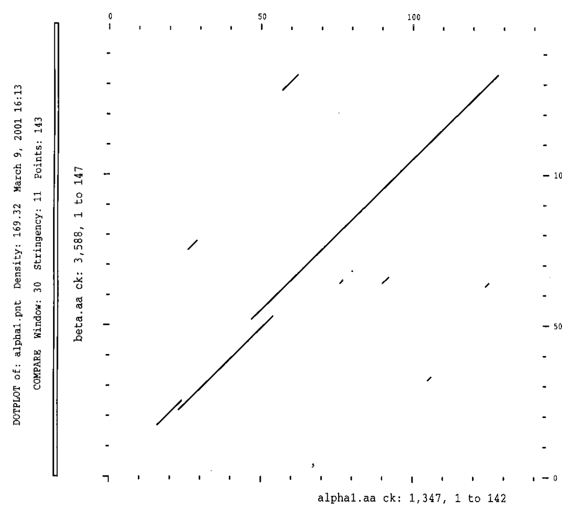

Figure 6.2 The dot-plot output of the amino acid sequences of the human hemoglobin alpha and beta chain.

6.2.2 The dynamic programming algorithm

In 1970, Needleman and Wunsch (1970, A general method applicable to the search for similarities in the amino acid sequence of two proteins. *J. Mol. Biol.*, 48: 443 - 453) introduce an elegant algorithm for comparing two proteins sequences. This general algorithm works also for aligning nucleic acid sequences as well. The algorithm actually belongs to a very large class of algorithms for finding optimal solutions. The essence of the algorithm is a technique known as dynamic programming.

For any letter sequence *s*, the segment of the sequence consisting of the letters from the beginning of the sequence up to the *i*th letter in the sequence is called a prefix, and it is denoted by *s*[1..*i*]. The dynamic programming technique basically tries to find the optimal alignment by taking advantage of the optimal alignments already found for the prefixes of the sequence. Suppose *s* and *t* are two sequences of size *m* and *n* respectively, there are *m*+1 possible prefixes of *s* and *n*+1 prefixes of *t*, including the empty string. To explain the calculations, we arrange our calculations in an (*m*+1) x (*n*+1) where entry (*i*, *j*) contains the similarity between the prefixes *s*[1..*i*] and *t*[1..*j*].

Let us illustrate the dynamic programming algorithm using an example. We shall try to align the two DNA sequences *s* = AGTCA and *t* = GCTC with *m* = 5 and *n* = 4. Every base match receives a similarity score or +1 and every mismatch -1. The gap penalty function is chosen to be $w(k) = -1-2k$, where *k* is the length of the gap. In other words, a penalty of -3 will be given to a gap of length 1, and the penalty increase by multiples of 2 as the gap lengthens.

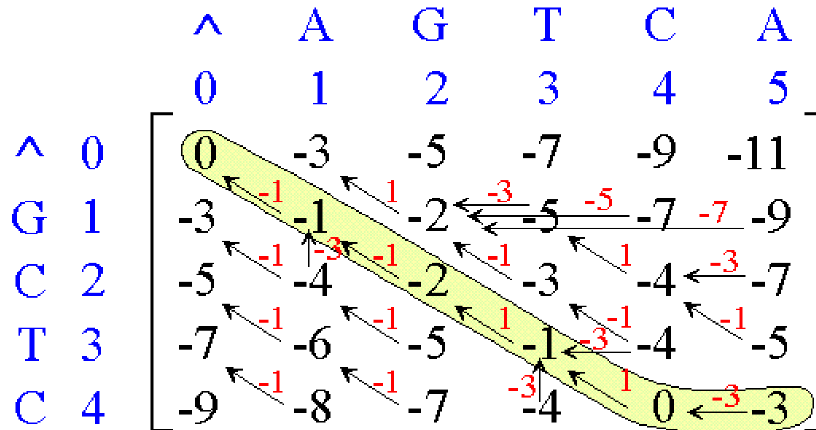

Figure 6.3 Dynamic programming algorithm for global alignment

We place *s* on top and *t* along the left margin of a rectangular array. A special character "^" is introduced to indicate that the sequence will begin at the next position. The 0th row and 0th column are initialized with the gap penalty function with *k* being the length of the "gap" that has to be inserted at the beginning of either sequence. For instance, cell (0,3) has a value -7 because having the 0th character "^" of sequence *t* lining up with the 3rd character "G" of sequence *s*, producing a gap of length 3 at the beginning of the alignment like this:

$$\begin{array}{cccc} A & G & T & \ldots \\ _ & _ & _ & \ldots \end{array}$$

The gap penalty, accordingly, is -1-2(3) = -7.

For the rest of the array, cell (i, j) will be filled the entry with the amount of similarity sim($s[1..i]$, $t[1..j]$) between the prefixes $s[1..i]$ and $t[1..j]$ computed recursively. Suppose we have already filled the entries at (i-1, j), (i-1, j-1), (i, j-1). Then we can compute

$$(6.1) \qquad sim(s[1..i], t[1..j]) = \max \begin{cases} sim(s[1..i], t[1..j-k]) + w(k); & k = 1, \ldots j-1 \\ sim(s[1..i-1], t[1..j-1]) + p(i, j) \\ sim(s[1..i-k], t[1..j]) + w(k); & k = 1, \ldots, i-1 \end{cases}$$

The reason is that there are just these possible ways of obtaining an alignment between $s[1..i]$ and $t[1..j]$:
(A) Align $s[1..i]$ and $t[1..j\text{-}k]$ and match the last space in a gap with length k with $t[j]$, or
(B) Align $s[1..i\text{-}1]$ and $t[1..j\text{-}1]$, and match $s[i]$ with $t[j]$, or
(C) Align $s[1..i\text{-}k]$ and $t[1..j]$, and match $s[i]$ with the last space in a gap with length k.

These possibilities are exhaustive because we cannot have two spaces paired in the last column of the alignment. Scores of the best alignments between smaller prefixes are already stored in the array if we choose an appropriate order in which to compute the entries (e.g., fill the array row by row, left to right in each row; or fill the array column by column, top to bottom on each column).

As we enter each entry in the array following equation (6.1), we draw an arrow to indicate where the maximum value comes from. For instance, the value of $sim[1,3]$ was taken as the maximum among the following numbers.

$$\begin{array}{c} sim[1,2] - 3 = -2 - 3 = -5 \\ sim[1,1] - 5 = -1 - 5 = -6 \\ sim[0,2] - 1 = -5 + 1 = -6 \\ sim[0,3] - 3 = -7 - 3 = -10 \end{array}$$

The maximum value comes from entry (1,2), and that is where the arrow shows. If there are more than one way of getting the maximum, we put arrows to indicate all the possibilities. See, for example, entry (2,1).

After the array has been completely filled, we find the best alignment by tracing back along the arrows. We start at the bottom right corner of the array and move according to the direction of the arrow. The best alignment we get from Figure 6.3 is

```
A G T C A
:  :  |  |
G C T C _
```

When there are multiple arrows emanating from an entry, we can follow any one of them. So it is possible to have more than one optimal alignment. Most computer programs for sequence alignment will report all different optimal alignments. It is important to note that an optimal alignment is optimal only for the particular similarity score matrix and the gap penalty functions. When any of these is altered, the optimal alignment will also change.

The GCG program "Gap" uses the above algorithm to find the best global alignment of two sequences.

Exercise With the same similarity scoring scheme, and gap penalty as in the example above, find the best alignment between the pair of DNA sequences in the beginning of this section.

In the description above, we try to find an alignment that gives an overall best similarity scores between the entirety of the two sequences. This is called an optimal *global alignment*. At times, our aim is to find the best segments from the given pair of the sequences that lines up best with each other. This is called an optimal *local alignment*. Local alignments are particularly useful when a new sequence is just obtained from the laboratory. The researcher would first like to identify any parts of the sequence that have high similarity to known functional domains. The popular database search program BLAST uses a local alignment algorithm.

The dynamic programming local alignment algorithm was developed in the early 1980's (Smith and Waterman 1981) and is frequently referred to as the Smith-Waterman algorithm. It shares the same basic concepts with the global algorithm, differing only in a few details.

First, an extra possibility is added to equation (6.1), allowing $sim(s[1..i], t[1..j])$ to take the value if all other options have value less than 0. That is,

$$sim(s[1..i], t[1..j]) = \max \begin{cases} 0 \\ sim(s[1..i], t[1..j-k]) + w(k); \quad k = 1,...j-1 \\ sim(s[1..i-1], t[1..j-1]) + p(i, j) \\ sim(s[1..i-k], t[1..j]) + w(k); \quad k = 1,..., i-1 \end{cases}$$

Consequently, the top row and left column of the array in Figure 6.3 will now be filled with 0's instead of the $w(k)$'s as in global alignment. Taking the option 0 corresponds to starting a new alignment. Since we are only looking for a local alignment, the alignment can start anywhere in the two sequences. So, if the best alignment up to a certain point has a negative score, it is better to start a new one at that point.

Second, the alignment can end anywhere in the sequences. So, instead of starting the traceback from the bottom right corner, we look for the highest similarity value in the array and start the traceback from there. The traceback ends when we meet a cell with value 0, which corresponds to the start of the alignment.

If we follow equation (6.2) to find the best local alignment to the same pair of sequences *s* and *t*. We will have the array in Figure 6.4, which indicates that the best local alignment between these two sequences is the match of two nucleotide bases TC in the 3rd and 4th positions of both sequences.

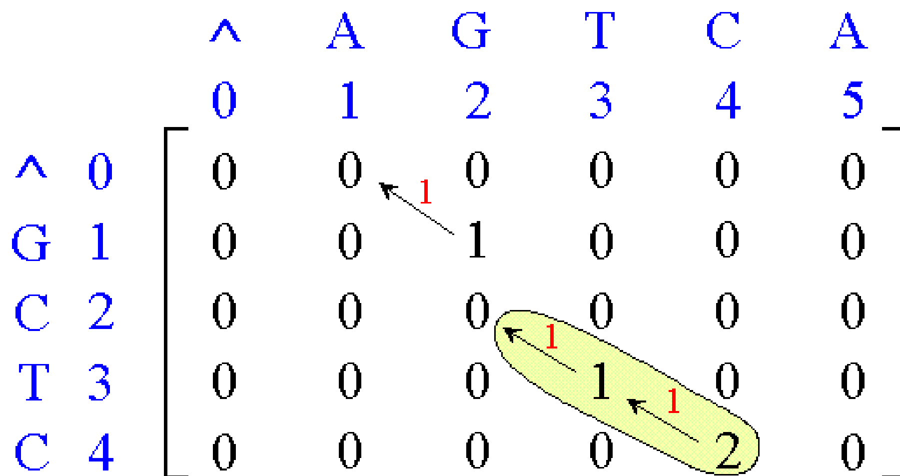

Figure 6.4 Dynamic programming algorithm for local alignment

The GCG programs that use dynamic programming algorithms for local pairwise sequence alignments are BestFit and FrameAlign.