2

Pairwise alignment

2.1 Introduction

The most basic sequence analysis task is to ask if two sequences are related. This is usually done by first aligning the sequences (or parts of them) and then deciding whether that alignment is more likely to have occurred because the sequences are related, or just by chance. The key issues are: (1) what sorts of alignment should be considered; (2) the scoring system used to rank alignments; (3) the algorithm used to find optimal (or good) scoring alignments; and (4) the statistical methods used to evaluate the significance of an alignment score.

Figure 2.1 shows an example of three pairwise alignments, all to the same region of the human alpha globin protein sequence (SWISS-PROT database identifier HBA_HUMAN). The central line in each alignment indicates identical positions with letters, and 'similar' positions with a plus sign. ('Similar' pairs of residues are those which have a positive score in the substitution matrix used to score the alignment; we will discuss substitution matrices shortly.) In the first

```
(a)
HBA_HUMAN   GSAQVKGHGKKVADALTNAVAHVDDMPNALSALSDLHAHKL
            G+ +VK+HGKKV  A+++++AH+D++ +++++LS+LH  KL
HBB_HUMAN   GNPKVKAHGKKVLGAFSDGLAHLDNLKGTFATLSELHCDKL

(b)
HBA_HUMAN   GSAQVKGHGKKVADALTNAVAHV---D--DMPNALSALSDLHAHKL
            ++ ++++H+ KV   + +A  ++           +L+ L+++H+ K
LGB2_LUPLU  NNPELQAHAGKVFKLVYEAAIQLQVTGVVVTDATLKNLGSVHVSKG

(c)
HBA_HUMAN   GSAQVKGHGKKVADALTNAVAHVDDMPNALSALSD----LHAHKL
            GS+ + G +   +D L  ++ H+ D+  A +AL D    ++AH+
F11G11.2    GSGYLVGDSLTFVDLL--VAQHTADLLAANAALLDEFPQFKAHQE
```

Figure 2.1 *Three sequence alignments to a fragment of human alpha globin. (a) Clear similarity to human beta globin. (b) A structurally plausible alignment to leghaemoglobin from yellow lupin. (c) A spurious high-scoring alignment to a nematode glutathione S-transferase homologue named F11G11.2.*

alignment there are many positions at which the two corresponding residues are identical; many others are functionally conservative, such as the pair D–E towards the end, representing an alignment of an aspartic acid residue with a glutamic acid residue, both negatively charged amino acids. Figure 2.1b also shows a biologically meaningful alignment, in that we know that these two sequences are evolutionarily related, have the same three-dimensional structure, and function in oxygen binding. However, in this case there are many fewer identities, and in a couple of places gaps have been inserted into the alpha globin sequence to maintain the alignment across regions where the leghaemoglobin has extra residues. Figure 2.1c shows an alignment with a similar number of identities or conservative changes. However, in this case we are looking at a spurious alignment to a protein that has a completely different structure and function.

How are we to distinguish cases like Figure 2.1b from those like Figure 2.1c? This is the challenge for pairwise alignment methods. We must give careful thought to the scoring system we use to evaluate alignments. The next section introduces the issues in how to score alignments, and then there is a series of sections on methods to find the best alignments according to the scoring scheme. The chapter finishes with a discussion of the statistical significance of matches, and more detail on parameterising the scoring scheme. Even so, it will not always be possible to distinguish true alignments from spurious alignments. For example, it is in fact extremely difficult to find significant similarity between the lupin leghaemoglobin and human alpha globin in Figure 2.1b using pairwise alignment methods.

2.2 The scoring model

When we compare sequences, we are looking for evidence that they have diverged from a common ancestor by a process of mutation and selection. The basic mutational processes that are considered are *substitutions*, which change residues in a sequence, and *insertions* and *deletions*, which add or remove residues. Insertions and deletions are together referred to as *gaps*. Natural selection has an effect on this process by screening the mutations, so that some sorts of change may be seen more than others.

The total score we assign to an alignment will be a sum of terms for each aligned pair of residues, plus terms for each gap. In our probabilistic interpretation, this will correspond to the logarithm of the relative likelihood that the sequences are related, compared to being unrelated. Informally, we expect identities and conservative substitutions to be more likely in alignments than we expect by chance, and so to contribute positive score terms; and non-conservative changes are expected to be observed less frequently in real alignments than we expect by chance, and so these contribute negative score terms.

Using an additive scoring scheme corresponds to an assumption that we can consider mutations at different sites in a sequence to have occurred independently (treating a gap of arbitrary length as a single mutation). All the algorithms in this chapter for finding optimal alignments depend on such a scoring scheme. The assumption of independence appears to be a reasonable approximation for DNA and protein sequences, although we know that interactions between residues play a very critical role in determining protein structure. However, it is seriously inaccurate for structural RNAs, where base pairing introduces very important long-range dependencies. It is possible to take these dependencies into account, but doing so gives rise to significant computational complexities; we will delay the subject of RNA alignment until the end of the book (Chapter 10).

Substitution matrices

We need score terms for each aligned residue pair. A biologist with a good intuition for proteins could invent a set of 210 scoring terms for all possible pairs of amino acids, but it is extremely useful to have a guiding theory for what the scores mean. We will derive substitution scores from a probabilistic model.

First, let us establish some notation. We will be considering a pair of sequences, x and y, of lengths n and m, respectively. Let x_i be the ith symbol in x and y_j be the jth symbol of y. These symbols will come from some alphabet $\mathcal{A}$; in the case of DNA this will be the four bases {A, G, C, T}, and in the case of proteins the twenty amino acids. We denote symbols from this alphabet by lower-case letters like a, b. For now we will only consider ungapped global pairwise alignments: that is, two completely aligned equal-length sequences as in Figure 2.1a.

Given a pair of aligned sequences, we want to assign a score to the alignment that gives a measure of the relative likelihood that the sequences are related as opposed to being unrelated. We do this by having models that assign a probability to the alignment in each of the two cases; we then consider the ratio of the two probabilities.

The unrelated or *random* model R is simplest. It assumes that letter a occurs independently with some frequency q_a, and hence the probability of the two sequences is just the product of the probabilities of each amino acid:

$$P(x, y|R) = \prod_i q_{x_i} \prod_j q_{y_j}. \tag{2.1}$$

In the alternative *match* model M, aligned pairs of residues occur with a joint probability p_{ab}. This value p_{ab} can be thought of as the probability that the residues a and b have each independently been derived from some unknown original residue c in their common ancestor (c might be the same as a and/or b). This gives a probability for the whole alignment of

$$P(x,y|M) = \prod_i p_{x_i y_i}.$$

The ratio of these two likelihoods is known as the *odds ratio*:

$$\frac{P(x,y|M)}{P(x,y|R)} = \frac{\prod_i p_{x_i y_i}}{\prod_i q_{x_i} \prod_i q_{y_i}} = \prod_i \frac{p_{x_i y_i}}{q_{x_i} q_{y_i}}.$$

In order to arrive at an additive scoring system, we take the logarithm of this ratio, known as the *log-odds ratio*:

$$S = \sum_i s(x_i, y_i), \tag{2.2}$$

where

$$s(a,b) = \log\left(\frac{p_{ab}}{q_a q_b}\right) \tag{2.3}$$

is the log likelihood ratio of the residue pair (a,b) occurring as an aligned pair, as opposed to an unaligned pair.

As we wanted, equation (2.2) is a sum of individual scores $s(a,b)$ for each aligned pair of residues. The $s(a,b)$ scores can be arranged in a matrix. For proteins, for instance, they form a 20×20 matrix, with $s(a_i,a_j)$ in position i,j in the matrix, where a_i,a_j are the ith and jth amino acids (in some numbering). This is known as a *score matrix* or a *substitution matrix*. An example of a substitution matrix derived essentially as above is the BLOSUM50 matrix, shown in Figure 2.2. We can use these values to score Figure 2.1a and get a score of 130. Another commonly used set of substitution matrices are called the PAM matrices. A detailed description of the way that the BLOSUM and PAM matrices are derived is given at the end of the chapter.

An important result is that even if an intuitive biologist were to write down an *ad hoc* substitution matrix, the substitution matrix implies 'target frequencies' p_{ab} according to the above theory [Altschul 1991]. Any substitution matrix is making a statement about the probability of observing ab pairs in real alignments.

Exercise

2.1 Amino acids D, E and K are all charged; V, I and L are all hydrophobic. What is the average BLOSUM50 score within the charged group of three? Within the hydrophobic group? Between the two groups? Suggest reasons for the pattern observed.

	A	R	N	D	C	Q	E	G	H	I	L	K	M	F	P	S	T	W	Y	V
A	**5**	−2	−1	−2	−1	−1	−1	0	−2	−1	−2	−1	−1	−3	−1	1	0	−3	−2	0
R	−2	**7**	−1	−2	−4	1	0	−3	0	−4	−3	3	−2	−3	−3	−1	−1	−3	−1	−3
N	−1	−1	**7**	2	−2	0	0	0	1	−3	−4	0	−2	−4	−2	1	0	−4	−2	−3
D	−2	−2	2	**8**	−4	0	2	−1	−1	−4	−4	−1	−4	−5	−1	0	−1	−5	−3	−4
C	−1	−4	−2	−4	**13**	−3	−3	−3	−3	−2	−2	−3	−2	−2	−4	−1	−1	−5	−3	−1
Q	−1	1	0	0	−3	**7**	2	−2	1	−3	−2	2	0	−4	−1	0	−1	−1	−1	−3
E	−1	0	0	2	−3	2	**6**	−3	0	−4	−3	1	−2	−3	−1	−1	−1	−3	−2	−3
G	0	−3	0	−1	−3	−2	−3	**8**	−2	−4	−4	−2	−3	−4	−2	0	−2	−3	−3	−4
H	−2	0	1	−1	−3	1	0	−2	**10**	−4	−3	0	−1	−1	−2	−1	−2	−3	2	−4
I	−1	−4	−3	−4	−2	−3	−4	−4	−4	**5**	2	−3	2	0	−3	−3	−1	−3	−1	4
L	−2	−3	−4	−4	−2	−2	−3	−4	−3	2	**5**	−3	3	1	−4	−3	−1	−2	−1	1
K	−1	3	0	−1	−3	2	1	−2	0	−3	−3	**6**	−2	−4	−1	0	−1	−3	−2	−3
M	−1	−2	−2	−4	−2	0	−2	−3	−1	2	3	−2	**7**	0	−3	−2	−1	−1	0	1
F	−3	−3	−4	−5	−2	−4	−3	−4	−1	0	1	−4	0	**8**	−4	−3	−2	1	4	−1
P	−1	−3	−2	−1	−4	−1	−1	−2	−2	−3	−4	−1	−3	−4	**10**	−1	−1	−4	−3	−3
S	1	−1	1	0	−1	0	−1	0	−1	−3	−3	0	−2	−3	−1	**5**	2	−4	−2	−2
T	0	−1	0	−1	−1	−1	−1	−2	−2	−1	−1	−1	−1	−2	−1	2	**5**	−3	−2	0
W	−3	−3	−4	−5	−5	−1	−3	−3	−3	−3	−2	−3	−1	1	−4	−4	−3	**15**	2	−3
Y	−2	−1	−2	−3	−3	−1	−2	−3	2	−1	−1	−2	0	4	−3	−2	−2	2	**8**	−1
V	0	−3	−3	−4	−1	−3	−3	−4	−4	4	1	−3	1	−1	−3	−2	0	−3	−1	**5**

Figure 2.2 *The* BLOSUM50 *substitution matrix. The log-odds values have been scaled and rounded to the nearest integer for purposes of computational efficiency. Entries on the main diagonal for identical residue pairs are highlighted in bold.*

Gap penalties

We expect to penalise gaps. The standard cost associated with a gap of length g is given either by a linear score

$$\gamma(g) = -gd \tag{2.4}$$

or an affine score

$$\gamma(g) = -d - (g-1)e \tag{2.5}$$

where d is called the *gap-open* penalty and e is called the *gap-extension* penalty. The gap-extension penalty e is usually set to something less than the gap-open penalty d, allowing long insertions and deletions to be penalised less than they would be by the linear gap cost. This is desirable when gaps of a few residues are expected almost as frequently as gaps of a single residue.

Gap penalties also correspond to a probabilistic model of alignment, although this is less widely recognised than the probabilistic basis of substitution matrices. We assume that the probability of a gap occurring at a particular site in a given sequence is the product of a function $f(g)$ of the length of the gap, and the combined probability of the set of inserted residues,

$$P(\text{gap}) = f(g) \prod_{i \text{ in gap}} q_{x_i}. \tag{2.6}$$

The form of (2.6) as a product of $f(g)$ with the q_{x_i} terms corresponds to an assumption that the length of a gap is not correlated to the residues it contains.

The natural values for the q_a probabilities here are the same as those used in the random model, because they both correspond to unmatched independent residues. In this case, when we divide by the probability of this region according to the random model to form the odds ratio, the q_{x_i} terms cancel out, so we are left only with a term dependent on length $\gamma(g) = \log(f(g))$; gap penalties correspond to the log probability of a gap of that length.

On the other hand, if there is evidence for a different distribution of residues in gap regions then there should be residue-specific scores for the unaligned residues in gap regions, equal to the logs of the ratio of their frequencies in gapped versus aligned regions. This might happen if, for example, it is expected that polar amino acids are more likely to occur in gaps in protein alignments than indicated by their average frequency in protein sequences, because the gaps are more likely to be in loops on the surface of the protein structure than in the buried core.

Exercises

2.2 Show that the probability distributions $f(g)$ that correspond to the linear and affine gap schemes given in equations (2.4) and (2.5) are both geometric distributions, of the form $f(g) = k\mathrm{e}^{-\lambda g}$.

2.3 Typical gap penalties used in practice are $d = 8$ for the linear case, or $d = 12, e = 2$ for the affine case, both expressed in half bits. A *bit* is the unit obtained when one takes log base 2 of a probability, so in natural log units these correspond to $d = (8 \log 2)/2$ and $d = (12 \log 2)/2$, $e = (2 \log 2)/2$ respectively. What are the corresponding probabilities of a gap (of any length) starting at some position, and the distributions of gap length given that there is a gap?

2.4 Using the BLOSUM50 matrix in Figure 2.2 and an affine gap penalty of $d = 12$, $e = 2$, calculate the scores of the alignments in Figure 2.1b and Figure 2.1c.

2.3 Alignment algorithms

Given a scoring system, we need to have an algorithm for finding an optimal alignment for a pair of sequences. Where both sequences have the same length n, there is only one possible global alignment of the complete sequences, but things become more complicated once gaps are allowed (or once we start looking for local alignments between subsequences of two sequences). There are

$$\binom{2n}{n} = \frac{(2n)!}{(n!)^2} \simeq \frac{2^{2n}}{\sqrt{\pi n}} \tag{2.7}$$

possible global alignments between two sequences of length n. It is clearly not computationally feasible to enumerate all these, even for moderate values of n.

The algorithm for finding optimal alignments given an additive alignment score of the type we have described is called *dynamic programming*. Dynamic programming algorithms are central to computational sequence analysis. All the remaining chapters in this book except the last, which covers mathematical methods, make use of dynamic programming algorithms. The simplest dynamic programming alignment algorithms to understand are pairwise sequence alignment algorithms. The reader should be sure to understand this section, because it lays an important foundation for the book as a whole. Dynamic programming algorithms are guaranteed to find the optimal scoring alignment or set of alignments. In most cases heuristic methods have also been developed to perform the same type of search. These can be very fast, but they make additional assumptions and will miss the best match for some sequence pairs. We will briefly discuss a few approaches to heuristic searching later in the chapter.

Because we introduced the scoring scheme as a log-odds ratio, better alignments will have higher scores, and so we want to maximise the score to find the optimal alignment. Sometimes scores are assigned by other means and interpreted as *costs* or *edit distances*, in which case we would seek to minimise the cost of an alignment. Both approaches have been used in the biological sequence comparison literature. Dynamic programming algorithms apply to either case; the differences are trivial exchanges of 'min' for 'max'.

We introduce four basic types of alignment. The type of alignment that we want to look for depends on the source of the sequences that we want to align. For each alignment type there is a slightly different dynamic programming algorithm. In this section, we will only describe pairwise alignment for linear gap scores, with cost d per gap residue. However, the algorithms we introduce here easily extend to more complex gap models, as we will see later in the chapter.

We will use two short amino acid sequences to illustrate the various alignment methods, `HEAGAWGHEE` and `PAWHEAE`. To score the alignments, we use the BLOSUM50 score matrix, and a gap cost per unaligned residue of $d = -8$. Figure 2.3 shows a matrix s_{ij} of the local score $s(x_i, y_j)$ of aligning each residue pair from the two example sequences. Identical or conserved residue pairs are highlighted in bold. Informally, the goal of an alignment algorithm is to incorporate as many of these positively scoring pairs as possible into the alignment, while minimising the cost from unconserved residue pairs, gaps, and other constraints.

	H	E	A	G	A	W	G	H	E	E
P	−2	−1	−1	−2	−1	−4	−2	−2	−1	−1
A	−2	−1	**5**	0	**5**	−3	0	−2	−1	−1
W	−3	−3	−3	−3	−3	**15**	−3	−3	−3	−3
H	**10**	0	−2	−2	−2	−3	−2	**10**	0	0
E	0	**6**	−1	−3	−1	−3	−3	0	**6**	**6**
A	−2	−1	**5**	0	**5**	−3	0	−2	−1	−1
E	0	**6**	−1	−3	−1	−3	−3	0	**6**	**6**

Figure 2.3 *The two example sequences we will use for illustrating dynamic programming alignment algorithms, arranged to show a matrix of corresponding* BLOSUM50 *values per aligned residue pair. Positive scores are in bold.*

Exercises

2.5 Show that the number of ways of intercalating two sequences of lengths n and m to give a single sequence of length $n+m$, while preserving the order of the symbols in each, is $\binom{n+m}{m}$.

2.6 By taking alternating symbols from the upper and lower sequences in an alignment, then discarding the gap characters, show that there is a one-to-one correspondence between gapped alignments of the two sequences and intercalated sequences of the type described in the previous exercise. Hence derive the first part of equation (2.7).

2.7 Use Stirling's formula ($x! \simeq \sqrt{2\pi} x^{x+\frac{1}{2}} e^{-x}$) to prove the second part of equation (2.7).

Global alignment: Needleman–Wunsch algorithm

The first problem we consider is that of obtaining the optimal global alignment between two sequences, allowing gaps. The dynamic programming algorithm for solving this problem is known in biological sequence analysis as the Needleman–Wunsch algorithm [Needleman & Wunsch 1970], but the more efficient version that we describe was introduced by Gotoh [1982].

The idea is to build up an optimal alignment using previous solutions for optimal alignments of smaller subsequences. We construct a matrix F indexed by i and j, one index for each sequence, where the value $F(i,j)$ is the score of the best alignment between the initial segment $x_{1...i}$ of x up to x_i and the initial segment $y_{1...j}$ of y up to y_j. We can build $F(i,j)$ recursively. We begin by initialising $F(0,0) = 0$. We then proceed to fill the matrix from top left to bottom right. If $F(i-1,j-1)$, $F(i-1,j)$ and $F(i,j-1)$ are known, it is possible to calculate $F(i,j)$. There are three possible ways that the best score

I G A x_i	A I G A x_i	G A x_i – –
L G V y_j	G V y_j – –	S L G V y_j

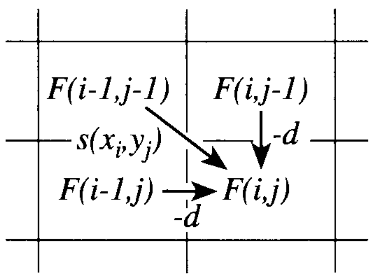
Figure 2.4 *The three ways an alignment can be extended up to* (i,j)*:* x_i *aligned to* y_j*,* x_i *aligned to a gap, and* y_j *aligned to a gap.*

$F(i,j)$ of an alignment up to x_i, y_j could be obtained: x_i could be aligned to y_j, in which case $F(i,j) = F(i-1,j-1) + s(x_i,y_j)$; or x_i is aligned to a gap, in which case $F(i,j) = F(i-1,j) - d$; or y_j is aligned to a gap, in which case $F(i,j) = F(i,j-1) - d$ (see Figure 2.4). The best score up to (i,j) will be the largest of these three options.

Therefore, we have

$$F(i,j) = \max \begin{cases} F(i-1,j-1) + s(x_i,y_j), \\ F(i-1,j) - d, \\ F(i,j-1) - d. \end{cases} \tag{2.8}$$

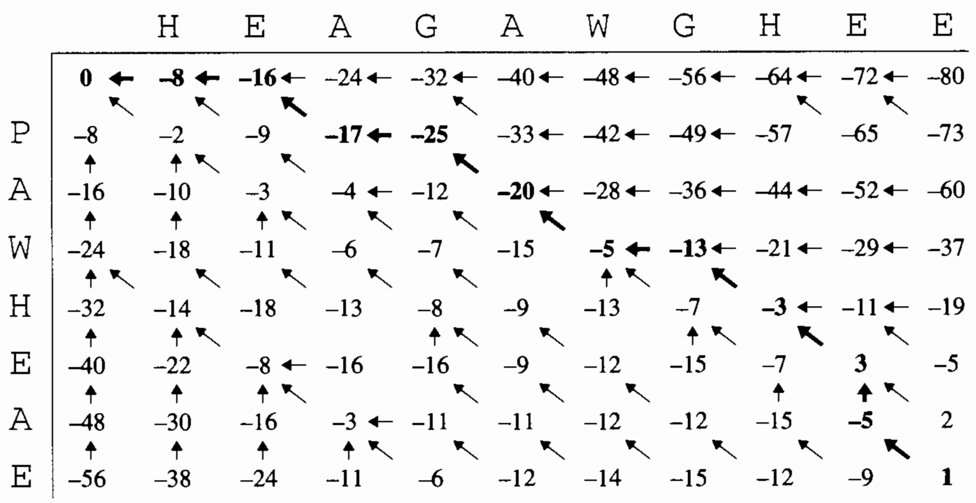
This equation is applied repeatedly to fill in the matrix of $F(i,j)$ values, calculating the value in the bottom right-hand corner of each square of four cells from one of the other three values (above-left, left, or above) as in the following figure.



As we fill in the $F(i,j)$ values, we also keep a pointer in each cell back to the cell from which its $F(i,j)$ was derived, as shown in the example of the full dynamic programming matrix in Figure 2.5.

To complete our specification of the algorithm, we must deal with some boundary conditions. Along the top row, where $j = 0$, the values $F(i,j-1)$ and $F(i-1,j-1)$ are not defined so the values $F(i,0)$ must be handled specially. The values $F(i,0)$ represent alignments of a prefix of x to all gaps in y, so we can define $F(i,0) = -id$. Likewise down the left column $F(0,j) = -jd$.

The value in the final cell of the matrix, $F(n,m)$, is by definition the best score for an alignment of $x_{1...n}$ to $y_{1...m}$, which is what we want: the score of the best global alignment of x to y. To find the alignment itself, we must find the path of choices from (2.8) that led to this final value. The procedure for doing this is known as a *traceback*. It works by building the alignment in reverse, starting from the final cell, and following the pointers that we stored when building the



		H	E	A	G	A	W	G	H	E	E
	0	**−8**	**−16**	−24	−32	−40	−48	−56	−64	−72	−80
P	−8	−2	−9	**−17**	**−25**	−33	−42	−49	−57	−65	−73
A	−16	−10	−3	−4	−12	**−20**	−28	−36	−44	−52	−60
W	−24	−18	−11	−6	−7	−15	**−5**	**−13**	−21	−29	−37
H	−32	−14	−18	−13	−8	−9	−13	−7	**−3**	−11	−19
E	−40	−22	−8	−16	−16	−9	−12	−15	−7	**3**	−5
A	−48	−30	−16	−3	−11	−11	−12	−12	−15	**−5**	2
E	−56	−38	−24	−11	−6	−12	−14	−15	−12	−9	**1**

```
HEAGAWGHE-E
--P-AW-HEAE
```

Figure 2.5 *Above, the global dynamic programming matrix for our example sequences, with arrows indicating traceback pointers; values on the optimal alignment path are shown in bold. Below, a corresponding optimal alignment, which has total score* 1.

matrix. At each step in the traceback process we move back from the current cell (i, j) to the one of the cells $(i-1, j-1)$, $(i-1, j)$ or $(i, j-1)$ from which the value $F(i, j)$ was derived. At the same time, we add a pair of symbols onto the front of the current alignment: x_i and y_j if the step was to $(i-1, j-1)$, x_i and the gap character '−' if the step was to $(i-1, j)$, or '−' and y_j if the step was to $(i, j-1)$. At the end we will reach the start of the matrix, $i = j = 0$. An example of this procedure is shown in Figure 2.5.

Note that in fact the traceback procedure described here finds just one alignment with the optimal score; if at any point two of the derivations are equal, an arbitrary choice is made between equal options. The traceback algorithm is easily modified to recover more than one equal-scoring optimal alignment. The set of all possible optimal alignments can be described fairly concisely using a sequence graph structure [Altschul & Erickson 1986; Hein 1989a]. We will use sequence graph structures in Chapter 7 where we describe Hein's algorithm for multiple alignment.

The reason that the algorithm works is that the score is made of a sum of independent pieces, so the best score up to some point in the alignment is the best score up to the point one step before, plus the incremental score of the new step.

Big-O notation for algorithmic complexity

It is useful to know how an algorithm's performance in CPU time and required memory storage will scale with the size of the problem. From the algorithm above, we see that we are storing $(n+1) \times (m+1)$ numbers, and each number costs us a constant number of calculations to compute (three sums and a max). We say that the algorithm takes $O(nm)$ time and $O(nm)$ memory, where n and m are the lengths of the sequences. '$O(nm)$' is a standard notation, called *big-O* notation, meaning 'of order nm', i.e. that the computation time or memory storage required to solve the problem scales as the product of the sequence lengths nm, up to a constant factor. Since n and m are usually comparable, the algorithm is usually said to be $O(n^2)$. The larger the exponent of n, the less practical the method becomes for long sequences. With biological sequences and standard computers, $O(n^2)$ algorithms are feasible but a little slow, while $O(n^3)$ algorithms are only feasible for very short sequences.

Exercises

2.8 Find a second equal-scoring optimal alignment in the dynamic programming matrix in Figure 2.5.

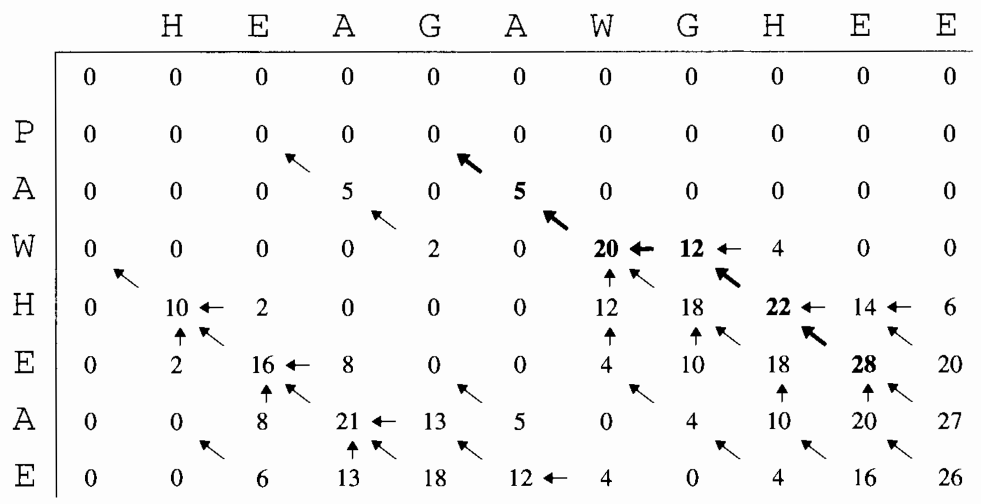
2.9 Calculate the dynamic programming matrix and an optimal alignment for the DNA sequences GAATTC and GATTA, scoring +2 for a match, -1 for a mismatch, and with a linear gap penalty of $d = 2$.

Local alignment: Smith–Waterman algorithm

So far we have assumed that we know which sequences we want to align, and that we are looking for the best match between them from one end to the other. A much more common situation is where we are looking for the best alignment between *subsequences* of x and y. This arises for example when it is suspected that two protein sequences may share a common domain, or when comparing extended sections of genomic DNA sequence. It is also usually the most sensitive way to detect similarity when comparing two very highly diverged sequences, even when they may have a shared evolutionary origin along their entire length. This is because usually in such cases only part of the sequence has been under strong enough selection to preserve detectable similarity; the rest of the sequence will have accumulated so much noise through mutation that it is no longer alignable. The highest scoring alignment of subsequences of x and y is called the best *local* alignment.

The algorithm for finding optimal local alignments is closely related to that described in the previous section for global alignments. There are two differences. First, in each cell in the table, an extra possibility is added to (2.8), allowing $F(i,j)$ to take the value 0 if all other options have value less than 0:

$$F(i,j) = \max \begin{cases} 0, \\ F(i-1,j-1) + s(x_i,y_j), \\ F(i-1,j) - d, \\ F(i,j-1) - d. \end{cases} \qquad (2.9)$$



		H	E	A	G	A	W	G	H	E	E
	0	0	0	0	0	0	0	0	0	0	0
P	0	0	0	0	0	0	0	0	0	0	0
A	0	0	0	5	0	**5**	0	0	0	0	0
W	0	0	0	0	2	0	**20**	**12**	4	0	0
H	0	10	2	0	0	0	12	18	**22**	14	6
E	0	2	16	8	0	0	4	10	18	**28**	20
A	0	0	8	21	13	5	0	4	10	20	27
E	0	0	6	13	18	12	4	0	4	16	26

```
AWGHE
AW-HE
```

Figure 2.6 *Above, the local dynamic programming matrix for the example sequences. Below, the optimal local alignment, with score* 28.

Taking the option 0 corresponds to starting a new alignment. If the best alignment up to some point has a negative score, it is better to start a new one, rather than extend the old one. Note that a consequence of the 0 is that the top row and left column will now be filled with 0s, not $-id$ and $-jd$ as for global alignment.

The second change is that now an alignment can end anywhere in the matrix, so instead of taking the value in the bottom right corner, $F(n,m)$, for the best score, we look for the highest value of $F(i,j)$ over the whole matrix, 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. An example is given in Figure 2.6, which shows the best local alignment of the same two sequences whose best global alignment was found in Figure 2.5. In this case the local alignment is a subset of the global alignment, but that is not always the case.

For the local alignment algorithm to work, the expected score for a random match must be negative. If that is not true, then long matches between entirely unrelated sequences will have high scores, just based on their length. As a consequence, although the algorithm is local, the maximal scoring alignments would be global or nearly global. A true subsequence alignment would be likely to be masked by a longer but incorrect alignment, just because of its length. Similarly, there must be some $s(a,b)$ greater than 0, otherwise the algorithm won't find any alignment at all (it finds the best score or 0, whichever is higher).

What is the precise meaning of the requirement that the expected score of a random match be negative? In the ungapped case, the relevant quantity to consider is the expected value of a fixed length alignment. Because successive positions are independent, we need only consider a single residue position, giving the condition

$$\sum_{a,b} q_a q_b s(a,b) < 0, \tag{2.10}$$

where q_a is the probability of symbol a at any given position in a sequence. When $s(a,b)$ is derived as a log likelihood ratio, as in the previous section, using the same q_a as for the random model probabilities, then (2.10) is always satisfied. This is because

$$\sum_{a,b} q_a q_b s(a,b) = -\sum_{a,b} q_a q_b \log \frac{q_a q_b}{p_{ab}} = -H(q^2||p)$$

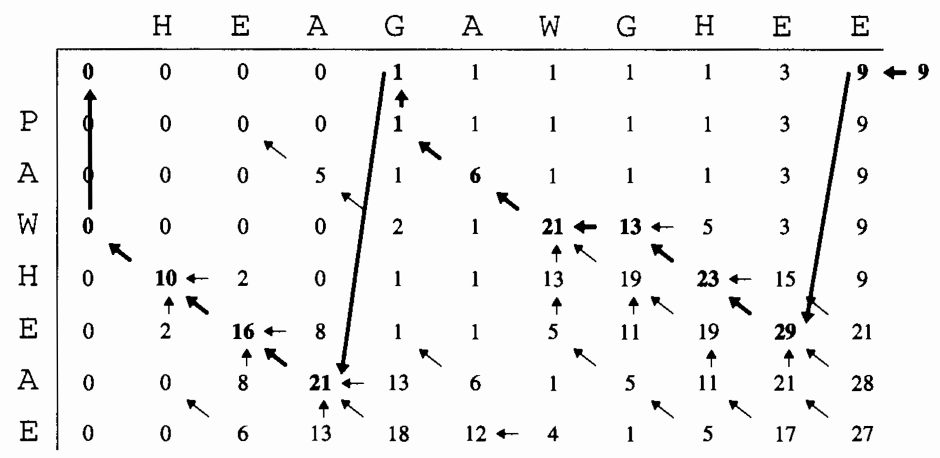
where $H(q^2||p)$ is the relative entropy of distribution $q^2 = q \times q$ with respect to distribution p, which is always positive unless $q^2 = p$ (see Chapter 11). In fact $H(q^2||p)$ is a natural measure of how different the two distributions are. It is also, by definition, a measure of how much information we expect per aligned residue pair in an alignment.

Unfortunately we cannot give an equivalent analysis for optimal gapped alignments. There is no analytical method for predicting what gap scores will result in local versus global alignment behaviour. However, this is a question of practical importance when setting parameter values in the scoring system (the match and gap scores $s(a,b)$ and $\gamma(g)$), and tables have been generated for standard scoring schemes showing local/global behaviour, along with other statistical properties [Altschul & Gish 1996]. We will return to this subject later, when considering the statistical significance of scores.

The local version of the dynamic programming sequence alignment algorithm was developed in the early 1980s. It is frequently known as the *Smith–Waterman* algorithm, after Smith & Waterman [1981]. Gotoh [1982] formulated the efficient affine gap cost version that is normally used (affine gap alignment algorithms are discussed on page 29).

Repeated matches

The procedure in the previous section gave the best single local match between two sequences. If one or both of the sequences are long, it is quite possible that there are many different local alignments with a significant score, and in most cases we would be interested in all of these. An example would be where there are many copies of a repeated domain or motif in a protein. We give here a method for finding such matches. This method is asymmetric: it finds one or more non-overlapping copies of sections of one sequence (e.g. the domain or motif) in the other. There is another widely used approach for finding multiple matches due to Waterman & Eggert [1987], which will be described in Chapter 4.



```
HEAGAWGHEE
HEA.AW-HE.
```

Figure 2.7 *Above, the repeat dynamic programming matrix for the example sequences, for* $T = 20$ *. Below, the optimal alignment, with total score* $9 = 29 - 20$*. There are two separate match regions, with scores* 1 *and* 8*. Dots are used to indicate unmatched regions of* x*.*

Let us assume that we are only interested in matches scoring higher than some threshold T. This will be true in general, because there are always short local alignments with small positive scores even between entirely unrelated sequences. Let y be the sequence containing the domain or motif, and x be the sequence in which we are looking for multiple matches.

An example of the repeat algorithm is given in Figure 2.7. We again use the matrix F, but the recurrence is now different, as is the meaning of $F(i, j)$. In the final alignment, x will be partitioned into regions that match parts of y in gapped alignments, and regions that are unmatched. We will talk about the score of a completed match region as being its standard gapped alignment score minus the threshold T. All these match scores will be positive. $F(i, j)$ for $j \geq 1$ is now the best sum of match scores to $x_{1...i}$, assuming that x_i is in a matched region, and the corresponding match ends in x_i and y_j (they may not actually be aligned, if this is a gapped section of the match). $F(i,0)$ is the best sum of completed match scores to the subsequence $x_{1...i}$, i.e. assuming that x_i is in an unmatched region.

To achieve the desired goal, we start by initialising $F(0,0) = 0$ as usual, and then fill the matrix using the following recurrence relations:

$$F(i,0) = \max\begin{cases} F(i-1,0), \\ F(i-1,j) - T, \qquad j = 1,\dots,m; \end{cases} \tag{2.11}$$

$$F(i,j) = \max \begin{cases} F(i,0), \\ F(i-1,j-1)+s(x_i,y_j), \\ F(i-1,j)-d, \\ F(i,j-1)-d. \end{cases} \qquad (2.12)$$

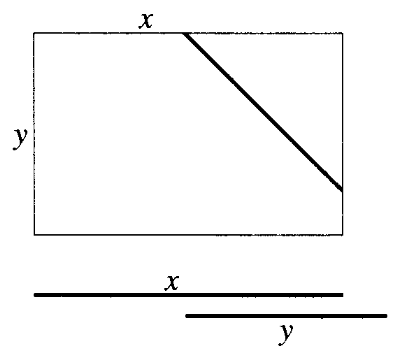
Equation (2.11) handles unmatched regions and ends of matches, only allowing matches to end when they have score at least T. Equation (2.12) handles starts of matches and extensions. The total score of all the matches is obtained by adding an extra cell to the matrix, $F(n+1,0)$, using (2.11). This score will have T subtracted for each match; if there were no matches of score greater than T it will be 0, obtained by repeated application of the first option in (2.11).

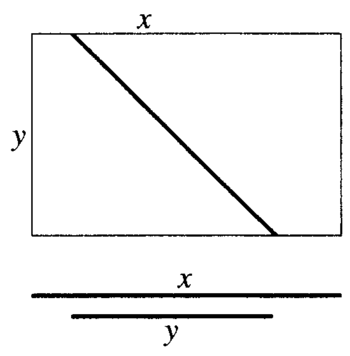
The individual match alignments can be obtained by tracing back from cell $(n+1,0)$ to $(0,0)$, at each point going back to the cell that was the source of the score in the current cell in the max() operation. This traceback procedure is a global procedure, showing what each residue in x will be aligned to. The resulting global alignment will contain sections of more conventional gapped local alignments of subsequences of x to subsequences of y.

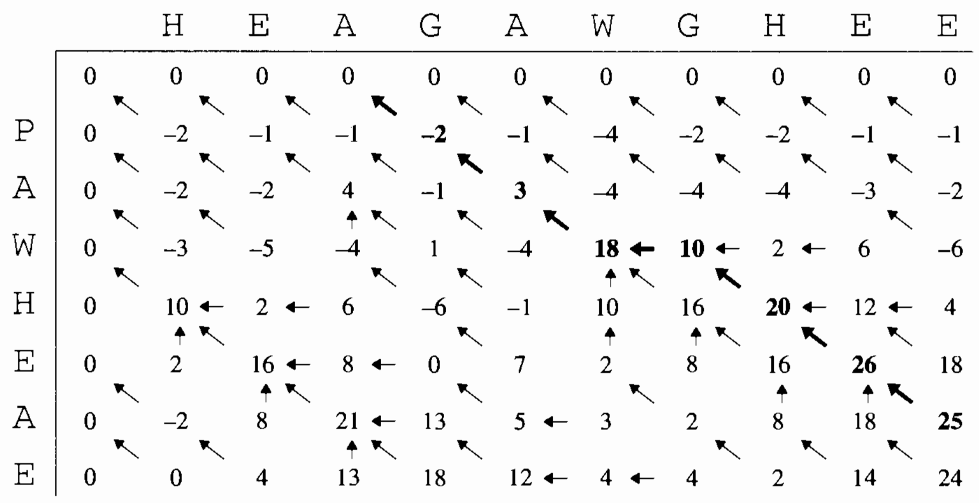
Note that the algorithm obtains all the local matches in one pass. It finds the maximal scoring set of matches, in the sense of maximising the combined total of the excess of each match score above the threshold T. Changing the value of T changes what the algorithm finds. Increasing T may exclude matches. Decreasing it may split them, as well as finding new weaker ones. A locally optimal match in the sense of the preceding section will be split into pieces if it contains internal subalignments scoring less than $-T$. However, this may be what is wanted: given two similar high scoring sections significant in their own right, separated by a non-matching section with a strongly negative score, it is not clear whether it is preferable to report one match or two.

Overlap matches

Another type of search is appropriate when we expect that one sequence contains the other, or that they overlap. This often occurs when comparing fragments of genomic DNA sequence to each other, or to larger chromosomal sequences. Several different types of configuration can occur, as shown here:







		H	E	A	G	A	W	G	H	E	E
	0	0	0	0	0	0	0	0	0	0	0
P	0	-2	-1	-1	**-2**	-1	-4	-2	-2	-1	-1
A	0	-2	-2	4	-1	**3**	-4	-4	-4	-3	-2
W	0	-3	-5	-4	1	-4	**18**	**10**	2	6	-6
H	0	10	2	6	-6	-1	10	16	**20**	12	4
E	0	2	16	8	0	7	2	8	16	**26**	18
A	0	-2	8	21	13	5	3	2	8	18	**25**
E	0	0	4	13	18	12	4	4	2	14	24

```
GAWGHEE
PAW-HEA
```

Figure 2.8 *Above, the overlap dynamic programming matrix for the example sequences. Below, the optimal overlap alignment, with score* 25.

What we want is really a type of global alignment, but one that does not penalise overhanging ends. This gives a clue to what sort of algorithm to use: we want a match to start on the top or left border of the matrix, and finish on the right or bottom border. The initialisation equations are therefore that $F(i,0) = 0$ for $i = 1,\ldots,n$ and $F(0,j) = 0$ for $j = 1,\ldots,m$, and the recurrence relations within the matrix are simply those for a global alignment (2.8). We set F_{max} to be the maximum value on the right border $(i,m), i = 1,\ldots,n$, and the bottom border $(n,j), j = 1,\ldots,m$. The traceback starts from the maximum point and continues until the top or left edge is reached.

There is a repeat version of this overlap match algorithm, in which the analogues of (2.11) and (2.12) are

$$F(i,0) = \max\begin{cases} F(i-1,0), \\ F(i-1,m) - T; \end{cases} \tag{2.13}$$

$$F(i,j) = \max\begin{cases} F(i-1,j-1) + s(x_i,y_j), \\ F(i-1,j) - d, \\ F(i,j-1) - d. \end{cases} \tag{2.14}$$

Note that the line (2.13) in the recursion for $F(i,0)$ is now just looking at complete matches to $y_{1\ldots m}$, rather than all possible subsequences of y as in (2.11) in the previous section. However, (2.11) is still used in its original form for obtaining $F(n+1,0)$, so that matches of initial subsequences of y to the end of x can be obtained.

Hybrid match conditions

By now it should be clear that a wide variety of different dynamic programming variants can be formulated. All of the alignment methods given above have been expressed in terms of a matrix $F(i,j)$, with various differing boundary conditions and recurrence rules. Given the common framework, we can see how to provide hybrid algorithms. We have already seen one example in the repeat version of the overlap algorithm. There are many possible further variants.

For example, where a repetitive sequence y tends to be found in tandem copies not separated by gaps, it can be useful to replace (2.14) for $j = 1$ with

$$F(i,1) = \max \begin{cases} F(i-1,0) + s(x_i, y_1), \\ F(i-1,n) + s(x_i, y_1), \\ F(i-1,1) - d, \\ F(i,0) - d. \end{cases}$$

This allows a bypass of the $-T$ penalty in (2.11), so the threshold applies only once to each tandem cluster of repeats, not once to each repeat.

Another example might be if we are looking for a match that starts at the beginning of both sequences but can end at any point. This would be implemented by setting only $F(0,0) = 0$, using (2.8) in the recurrence, but allowing the match to end at the largest value in the whole matrix.

In fact, it is even possible to consider mixed boundary conditions where, for example, there is thought to be a significant prior probability that an entire copy of a sequence will be found in a larger sequence, but also some probability that only a fragment will be present. In this case we would set penalties on the boundaries or for starting internal matches, calculating the penalty costs as the logarithms of the respective probabilities. Such a model would be appropriate when looking for members of a repeat family in genomic DNA, since normally these are whole copies of the repeat, but sometimes only fragments are seen.

When performing a sequence similarity search we should ideally always consider what types of match we are looking for, and use the most appropriate algorithm for that case. In practice, there are often only good implementations available of a few of the standard cases, and it is often more convenient to use those, and postprocess the resulting matches afterwards.

2.4 Dynamic programming with more complex models

So far we have only considered the simplest gap model, in which the gap score $\gamma(g)$ is a simple multiple of the length. This type of scoring scheme is not ideal for biological sequences: it penalises additional gap steps as much as the first, whereas, when gaps do occur, they are often longer than one residue. If we are given a general function for $\gamma(g)$ then we can still use all the dynamic programming versions described in Section 2.3, with adjustments to the recurrence relations as typified by the following:

$$F(i,j) = \max \begin{cases} F(i-1,j-1)+s(x_i,y_j), & \\ F(k,j)+\gamma(i-k), & k=0,\ldots,i-1, \\ F(i,k)+\gamma(j-k), & k=0,\ldots,j-1. \end{cases} \tag{2.15}$$

which gives a replacement for the basic global dynamic relation. However, this procedure now requires $O(n^3)$ operations to align two sequences of length n, rather than $O(n^2)$ for the linear gap cost version, because in each cell (i,j) we have to look at $i+j+1$ potential precursors, not just three as previously. This is a prohibitively costly increase in computational time in many cases. Under some conditions on the properties of $\gamma()$ the search in k can be bounded, returning the expected computational time to $O(n^2)$, although the constant of proportionality is higher in these cases [Miller & Myers 1988].

Alignment with affine gap scores

The standard alternative to using (2.15) is to assume an affine gap cost structure as in (2.5): $\gamma(g) = -d-(g-1)e$. For this form of gap cost there is once again an $O(n^2)$ implementation of dynamic programming. However, we now have to keep track of multiple values for each pair of residue coefficients (i,j) in place of the single value $F(i,j)$. We will initially explain the process in terms of three variables corresponding to the three separate situations shown in Figure 2.4, which we show again here for convenience.

```
I G A x_i        A I G A x_i        G A x_i - -
L G V y_j        G V y_j - -        S L G V y_j
```

Let $M(i,j)$ be the best score up to (i,j) given that x_i is aligned to y_j (left case), $I_x(i,j)$ be the best score given that x_i is aligned to a gap (in an insertion with respect to y, central case), and finally $I_y(i,j)$ be the best score given that y_j is in an insertion with respect to x (right case).

The recurrence relations corresponding to (2.15) now become

$$\begin{aligned} M(i,j) &= \max \begin{cases} M(i-1,j-1)+s(x_i,y_j), \\ I_x(i-1,j-1)+s(x_i,y_j), \\ I_y(i-1,j-1)+s(x_i,y_j); \end{cases} \\ I_x(i,j) &= \max \begin{cases} M(i-1,j)-d, \\ I_x(i-1,j)-e; \end{cases} \\ I_y(i,j) &= \max \begin{cases} M(i,j-1)-d, \\ I_y(i,j-1)-e. \end{cases} \end{aligned} \tag{2.16}$$

In these equations, we assume that a deletion will not be followed directly by an

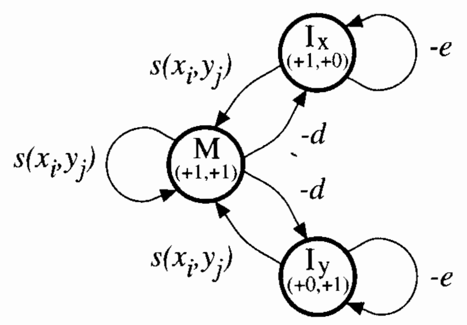


Figure 2.9 *A diagram of the relationships between the three states used for affine gap alignment.*

insertion. This will be true for the optimal path if $-d-e$ is less than the lowest mismatch score. As previously, we can find the alignment itself using a traceback procedure.

The system defined by equations (2.16) can be described very elegantly by the diagram in Figure 2.9. This shows a state for each of the three matrix values, with transition arrows between states. The transitions each carry a score increment, and the states each specify a $\Delta(i,j)$ pair, which is used to determine the change in indices i and j when that state is entered. The recurrence relation for updating each matrix value can be read directly from the diagram (compare Figure 2.9 with equations (2.16)). The new value for a 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 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. This type of description corresponds to a *finite state automaton* (FSA) in computer science. An alignment corresponds to a path through the states, with symbols from the underlying pair of sequences being transferred to the alignment according to the $\Delta(i,j)$ values in the states. An example of a short alignment and corresponding state path through the affine gap model is shown in Figure 2.10.

It is in fact frequent practice to implement an affine gap cost algorithm using

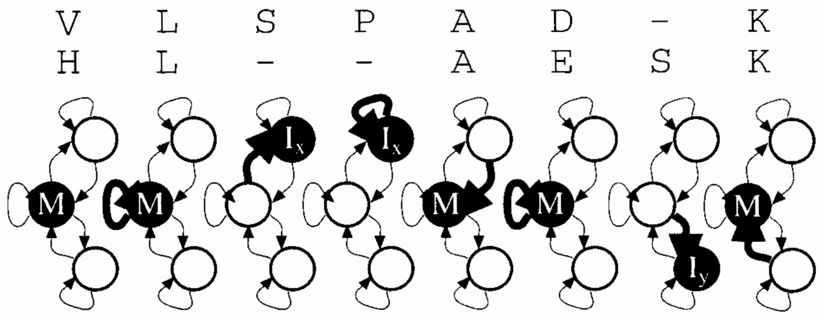


Figure 2.10 *An example of the state assignments for an alignment using the affine gap model.*

only two states, M and I, where I represents the possibility of being in a gapped region. Technically, this is only guaranteed to provide the correct result if the lowest mismatch score is greater than or equal to $-2e$. However, even if there are mismatch scores below $-2e$, the chances of a different alignment are very small. Furthermore, if one does occur it would not matter much, because the alignment differences would be in a very poorly matching gapped region. The recurrence relations for this version are

$$\begin{aligned} M(i,j) &= \max\begin{cases} M(i-1,j-1)+s(x_i,y_j), \\ I(i-1,j-1)+s(x_i,y_j); \end{cases} \\ I(i,j) &= \max\begin{cases} M(i,j-1)-d, \\ I(i,j-1)-e, \\ M(i-1,j)-d, \\ I(i-1,j)-e. \end{cases} \end{aligned}$$

These equations do not correspond to an FSA diagram as described above, because the I state may be used for $\Delta(1,0)$ or $\Delta(0,1)$ steps. There is, however, an alternative FSA formulation in which the $\Delta(i,j)$ values are associated with the transitions, rather than the states. This type of automaton can account for the two-state affine gap algorithm, using extra transitions for the deletion and insertion alternatives. In fact, the standard one-state algorithm for linear gap costs can be expressed as a single-state transition emitting FSA with three transitions corresponding to different $\Delta(i,j)$ values ($\Delta(1,1)$, $\Delta(1,0)$ and $\Delta(0,1)$). For those interested in pursuing the subject, the simpler state-based automata are called *Moore machines* in the computer science literature, and the transition-emitting systems are called *Mealy machines* (see Chapter 9).

More complex FSA models

One advantage of the FSA description of dynamic programming algorithms is that it is easy to see how to generate new types of algorithm. An example is given in Figure 2.11, which shows a four-state FSA with two match states. The idea here is that there may be high fidelity regions of alignment without gaps, corresponding to match state A, separated by lower fidelity regions with gaps, corresponding to match state B and gap states I_x and I_y. The substitution scores $s(a,b)$ and $t(a,b)$ can be chosen to reflect the expected degrees of similarity in the different regions. Similarly, FSA algorithms can be built for alignments of transmembrane proteins with separate match states for intracellular, extracellular or transmembrane regions, or for other more complex scenarios [Birney & Durbin 1997]. Searls & Murphy [1995] give a more abstract definition of such FSAs and have developed interactive tools for building them.

One feature of these more complex algorithms is that, given an alignment path, there is also an implicit attachment of labels to the symbols in the original se-

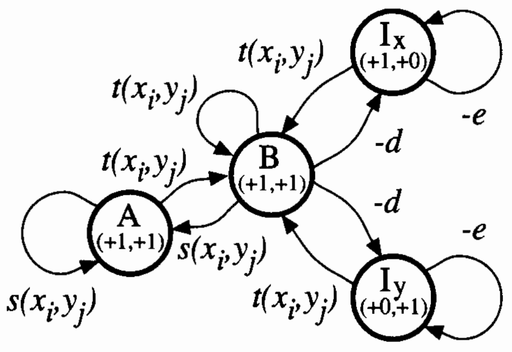


Figure 2.11 *The four-state finite state automaton with separate match states A and B for high and low fidelity regions. Note that this FSA emits on transitions with costs $s(x_i, y_j)$ and $t(x_i, y_j)$, rather than emitting on states, a distinction discussed earlier in the text.*

quences, indicating which state was used to match them. For example, with the transmembrane protein matching model, the alignment will assign sections of each protein to be transmembrane, intracellular or extracellular at the same time as finding the optimal alignment. In many cases this labelling of the sequence may be as important as the alignment information itself.

We will return to state models for pairwise alignment in Chapter 4.

Exercise

2.10 Calculate the score of the example alignment in Figure 2.10, with $d = 12, e = 2$.

2.5 Heuristic alignment algorithms

So far, all the alignment algorithms we have considered have been 'correct', in the sense that they are guaranteed to find the optimal score according to the specified scoring scheme. In particular, the affine gap versions described in the last section are generally regarded as providing the most sensitive sequence matching methods available. However, they are not the fastest available sequence alignment methods, and in many cases speed is an issue. The dynamic programming algorithms described so far have time complexity of the order of $O(nm)$, the product of the sequence lengths. The current protein database contains of the order of 100 million residues, so for a sequence of length one thousand, approximately 10^{11} matrix cells must be evaluated to search the complete database. At ten million matrix cells a second, which is reasonable for a single workstation at the time this is being written, this would take 10^4 seconds, or around three hours. If we want to search with many different sequences, time rapidly becomes an important issue.

For this reason, there have been many attempts to produce faster algorithms than straight dynamic programming. The goal of these methods is to search as small a fraction as possible of the cells in the dynamic programming matrix, while still looking at all the high scoring alignments. In cases where sequences are very similar, there are a number of methods based on extending computer science exact match string searching algorithms to non-exact cases, that provably find the optimal match [Chang & Lawler 1990; Wu & Manber 1992; Myers 1994]. However, for the scoring matrices used to find distant matches, these exact methods become intractable, and we must use heuristic approaches that sacrifice some sensitivity, in that there are cases where they can miss the best scoring alignment. A number of heuristic techniques are available. We give here brief descriptions of two of the best-known algorithms, BLAST and FASTA, to illustrate the types of approaches and trade offs that can be made. However, a detailed analysis of heuristic algorithms is beyond the scope of this book.

BLAST

The BLAST package [Altschul *et al.* 1990] provides programs for finding high scoring local alignments between a query sequence and a target database, both of which can be either DNA or protein. The idea behind the BLAST algorithm is that true match alignments are very likely to contain somewhere within them a short stretch of identities, or very high scoring matches. We can therefore look initially for such short stretches and use them as 'seeds', from which to extend out in search of a good longer alignment. By keeping the seed segments short, it is possible to pre-process the query sequence to make a table of all possible seeds with their corresponding start points.

BLAST makes a list of all 'neighbourhood words' of a fixed length (by default 3 for protein sequences, and 11 for nucleic acids), that would match the query sequence somewhere with score higher than some threshold, typically around 2 bits per residue. It then scans through the database, and whenever it finds a word in this set, it starts a 'hit extension' process to extend the possible match as an ungapped alignment in both directions, stopping at the maximum scoring extension (in fact, because of the way this is done, there is a small chance that it will stop short of the true maximal extension).

The most widely used implementation of BLAST finds ungapped alignments only. Perhaps surprisingly, restricting to ungapped alignments misses only a small proportion of significant matches, in part because the expected best score of unrelated sequences drops, so partial ungapped scores can still be significant, and also because BLAST can find and report more than one high scoring match per sequence pair and can give significance values for combined scores [Karlin & Altschul 1993]. Nonetheless, new versions of BLAST have recently become available that give gapped alignments [Altschul & Gish 1996; Altschul *et al.* 1997].

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 short 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 *ktup* between the two sequences. For proteins, *ktup* 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 subregion 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 *ktup*: higher values of *ktup* 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 $ktup = 1$.

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 $u = \lfloor \frac{n}{2} \rfloor$, the integer part of $\frac{n}{2}$. 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 (i',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.

Exercises

2.11 Fill in the correct values of $c(i, j)$ for the global alignment of the example pair of sequences in Figure 2.5 for the first pass of the algorithm ($u = 5$).

2.12 Show that the time required by the linear space algorithm is only about twice that of the standard $O(nm)$ algorithm.

2.7 Significance of scores

Now that we know how to find an optimal alignment, how can we assess the significance of its score? That is, how do we decide if it is a biologically meaningful alignment giving evidence for a homology, or just the best alignment between two entirely unrelated sequences? There are two possible approaches. One is Bayesian in flavour, based on the comparison of different models. The other is based on the traditional statistical approach of calculating the chance of a match score greater than the observed value, assuming a null model, which in this case is that the underlying sequences were unrelated.

The Bayesian approach: model comparison

We gave the log-odds ratio on p. 15 as the relevant score without much motivation. We might argue that what is really wanted is the probability that the sequences are related as opposed to being unrelated, which would be $P(M|x, y)$, rather than the likelihood calculated above, $P(x, y|M)$. $P(M|x, y)$ can be calculated using Bayes' rule, once we state some more assumptions. First we must specify the *a priori* probabilities of the two models. These reflect our expectation that the sequences are related before we actually see them. We will write these as $P(M)$, the prior probability that the sequences are related, and hence that the match model is correct, and $P(R) = 1 - P(M)$, the prior probability that the random model is correct. Then once we have seen the data the posterior probability that the match model is correct, and hence that the sequences are related, is

$$\begin{aligned} P(M|x,y) &= \frac{P(x,y|M)P(M)}{P(x,y)} \\ &= \frac{P(x,y|M)P(M)}{P(x,y|M)P(M)+P(x,y|R)P(R)} \\ &= \frac{P(x,y|M)P(M)/P(x,y|R)P(R)}{1+P(x,y|M)P(M)/P(x,y|R)P(R)}. \end{aligned}$$

Let

$$S' = S + \log\left(\frac{P(M)}{P(R)}\right) \tag{2.17}$$

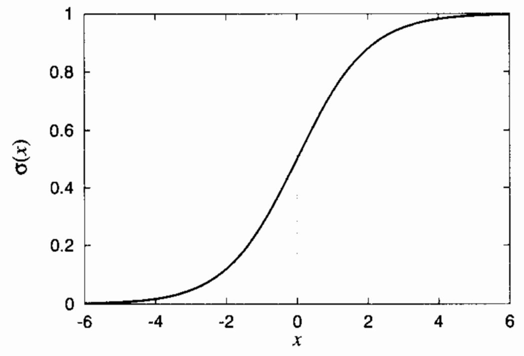


Figure 2.12 *The logistic function.*

where

$$S = \log\left(\frac{P(x,y|M)}{P(x,y|R)}\right)$$

is the log-odds score of the alignment. Then

$$P(M|x,y) = \sigma(S')$$

where

$$\sigma(x) = \frac{\mathrm{e}^x}{1+\mathrm{e}^x}.$$

$\sigma(x)$ is known as the *logistic* function. It is a sigmoid function, tending to 1 as x tends to infinity, to 0 as x tends to minus infinity, and with value $\frac{1}{2}$ at $x = 0$ (see Figure 2.12). The logistic function is widely used in neural network analysis to convert scores built from sums into probabilities – not entirely a coincidence.

From (2.17) we can see that we should add the prior log-odds ratio, $\log\left(\frac{P(M)}{P(R)}\right)$, to the standard score of the alignment. This corresponds to multiplying the likelihood ratio by the prior odds ratio, which makes intuitive sense. Once this has been done we can in principle compare the resulting value with 0 to indicate whether the sequences are related. For this to work, we have to be very careful that all the expressions we use really are probabilities, and in particular that when we sum them over all possible pairs of sequences that might have been given they sum to 1. When a scoring scheme is constructed in an *ad hoc* fashion this is unlikely to be true.

A particular example of where the prior odds ratio becomes important is when we are looking at a large number of different alignments for a possible significant match. This is the typical situation when searching a database. It is clear that if we have a fixed prior odds ratio, then even if all the database sequences are unrelated, as the number of sequences we try to match increases, the probability of one of the matches looking significant by chance will also increase. In fact, given a fixed prior odds ratio, the expected number of (falsely) significant observations will increase linearly. If we want it to stay fixed, then we must set the prior odds ratio in inverse proportion to the number of sequences in the database N. The effect of this is that to maintain a fixed number of false positives we should compare S with $\log N$, not 0. A conservative choice would be to choose a score that corresponds to an expected number of false positives of say 0.1 or 0.01. Of course, this type of approach is not necessarily appropriate. For example, we may believe that 1% of all proteins are kinases, in which case the prior odds should be $\frac{1}{100}$, and the expectation is that although false positives will increase as more sequences are looked at, so will true positives. On the other hand, if we believe that we will be looking for cases where one match in the whole database will be significant, then the $\log N$ comparison is more reasonable.

At this point we can turn to consider the statistical significance of a score obtained from the local match algorithm. In this case we have to correct for the fact that we are looking at the best of many possible different local matches between subsequences of the two sequences. A simple estimate of the number of start points of local matches is the product of the lengths of the sequences, nm. If all matches were constant length and all start points gave independent matches, this would result in a requirement to compare the best score S with $\log(nm)$. However, these assumptions are both clearly wrong (for instance, match segments at consecutive points along a diagonal are not independent), with the consequence that a further small correction factor should be added to S, dependent only on the scoring function s, but not on n and m. There is no analytical theory for this effect, but for scoring systems typically used when comparing protein sequences it seems that a multiplicative factor of around 0.1 is appropriate. Since what we care about is an additive term of the logarithm of this factor, the effect is comparatively small.

The classical approach: the extreme value distribution

There is an alternative way to consider significance in such situations, using a more classical statistical framework. We can look at the distribution of the maximum of N match scores to independent random sequences. If the probability of this maximum being greater than the observed best score is small, then the observation is considered significant.

In the simple case of a fixed ungapped alignment (2.2), the score of a match to a random sequence is the sum of many similar random variables, and so will be very well approximated by a normal distribution. The asymptotic distribution of the maximum M_N of a series of N independent normal random variables is known, and has the form

$$P(M_N \leq x) \simeq \exp(-KNe^{\lambda(x-\mu)}) \qquad (2.18)$$

for some constants K, λ. This form of limiting distribution is called the *extreme value distribution* or EVD (Chapter 11). We can use equation (2.18) to calculate the probability that the best match from a search of a large number N of unrelated sequences has score greater than our observed maximal score, S. If this is less than some small value, such as 0.05 or 0.01, then we can conclude that it is unlikely that the sequence giving rise to the observed maximal score is unrelated, i.e. it is likely that it is related.

It turns out that, even when the individual scores are not normally distributed, the extreme value distribution is still the correct limiting distribution for the maximum of a large number of separate scores (see Chapter 11). Because of this, the same type of significance test can be used for any search method that looks for the best score from a large set of equivalent possibilities. Indeed, for best local match scores from the local alignment algorithm, the best score between two (significantly long) sequences will itself be distributed according to the extreme value distribution, because in this case we are effectively comparing the outcomes of $O(nm)$ distinct random starts within the single matrix.

For local ungapped alignments, Karlin & Altschul [1990] derived the appropriate EVD distribution analytically, using results given more fully in Dembo & Karlin [1991]. We give this here in two steps. First, the number of unrelated matches with score greater than S is approximately Poisson distributed, with mean

$$E(S) = Kmn\mathrm{e}^{-\lambda S}, \tag{2.19}$$

where λ is the positive root of

$$\sum_{a,b} q_a q_b \mathrm{e}^{\lambda s(a,b)} = 1, \tag{2.20}$$

and K is a constant given by a geometrically convergent series also dependent only on the q_a and $s(a,b)$. This K corresponds directly to the multiplicative factor we described at the end of the previous section; it corrects for the non-independence of possible starting points for matches. The value λ is really a scale parameter, to convert the $s(a,b)$ into a natural scale. Note that if the $s(a,b)$ were initially derived as log likelihood quantities using equation (2.3) then $\lambda = 1$, because $\mathrm{e}^{\lambda s(a,b)} = p_{ab}/q_a q_b$.

The probability that there is a match of score greater than S is then

$$P(x > S) = 1 - \mathrm{e}^{-E(S)}. \tag{2.21}$$

It is easy to see that combining equations (2.19) and (2.21) gives a distribution of the same EVD form as (2.18), but without μ. In fact, it is common not to bother with calculating a probability, but just to use a requirement that $E(S)$ is significantly less than 1. This converts into a requirement that

$$S > T + \frac{\log mn}{\lambda} \tag{2.22}$$

for some fixed constant T. This corresponds to the Bayesian analysis in the

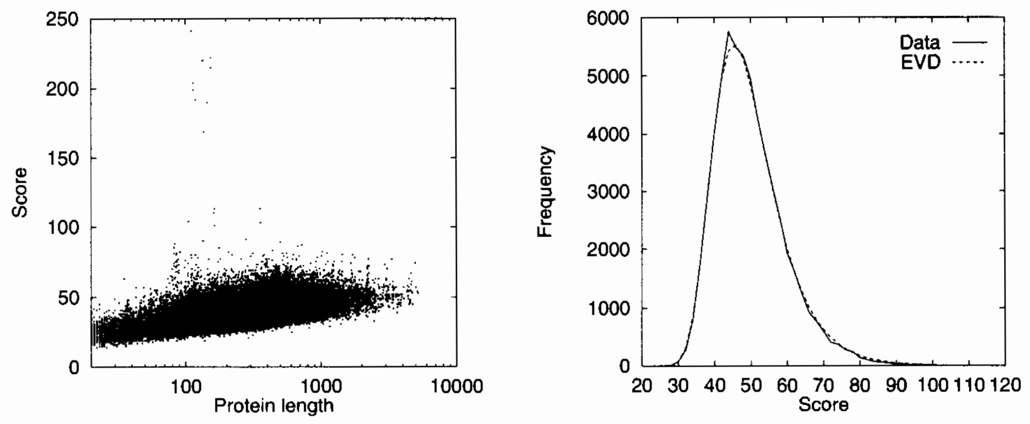


Figure 2.13 *Left, a scatter plot of the distribution of local match scores obtained from comparing human cytochrome C* (SWISS-PROT *accession code P000001) against the* SWISS-PROT34 *protein database with the Smith–Waterman implementation* SSEARCH *[Pearson 1996]. Right, the corresponding length-normalised distribution of scores, showing the fit to an EVD distribution.*

previous section suggesting that we should compare S with $\log mn$, but in this case we can assign a precise meaning to the value of T that we use.

Although no corresponding analytical theory has yet been derived for gapped alignments, Mott [1992] suggested that gapped alignment scores for random sequences follow the same form of extreme value distribution as ungapped scores, and there is now considerable empirical evidence to support this. Altschul & Gish [1996] have fit λ and K values for (2.19) for a range of standard protein alignment scoring schemes, using a large amount of randomly generated sample data.

Correcting for length

When searching a database of mixed length sequences, the best local matches to longer database sequences tend to have higher scores than the best local matches to shorter sequences, even when all the sequences are unrelated. An example is shown in Figure 2.13. This is not surprising: if our search sequence has length n and the database sequences have length m_i, then there are more possible start points in the nm_i matrix for larger m_i. However, if our prior expectation is that a match to any database entry should be equally likely, then we want random match scores to be comparable independent of length.

A theoretically justifiable correction for length dependence is that we should adjust the best score for each database entry by subtracting $\log(m_i)$. This follows from the expression for S' in the previous section. An alternative, which appears to perform slightly better in practice and is easily carried out when there are large numbers of sequences being searched, is to bin all the database entries by length, and then fit a linear function of the log sequence length [Pearson 1995] (the separation of 'background' from signal makes this a little tricky to implement).

Why use the alignment score as the test statistic?

So far in this section we have always assumed that we will use the same alignment score as a test statistic for the alignment's significance as was used to find the best match during the search phase. It might seem attractive to search for a match with one criterion, then evaluate it with another, uncorrelated one. This would seem to prevent the problem that the search phase increases the background level when testing. However, we need both the search and significance test to have as much discriminative power as possible. It is important to use the best available statistic for both. If we miss a genuinely related alignment in the search phase, then we obviously can't consider it when testing for significance.

A consequence of using the test statistic for searching is that the best match in unrelated sequences will tend to look qualitatively like a real match. As a striking example of this, Karlin & Altschul [1990] showed that when optimal local ungapped alignments are found between random sequences, the frequency of observing residue a aligned to residue b in these alignments will be $q_a q_b e^{\lambda s(a,b)}$, i.e. exactly the frequency p_{ab} with which we expect to observe a being aligned to b in our true, evolutionarily matched model. The only property we can use to discriminate true from false matches is the magnitude of the score, the expectation of which is proportional to the length of the match.

Of course, it may be that there are complex calculations involved in the most sensitive scoring scheme, which could not practically be implemented during the search stage. In this case, it may be necessary to search with a simpler score, but keep several alternative high scoring alignments, rather than simply the best one. We give methods for obtaining such suboptimal alignments in Chapter 4.

2.8 Deriving score parameters from alignment data

We finish this chapter by returning to the subject of the first section: how to determine the components of the scoring model, the substitution and gap scores. There we described how to derive scores for pairwise alignment algorithms from probabilities. However, this left open the issue of how to estimate the probabilities. It should be clear that the performance of our whole alignment system will depend on the values of these parameters, so considerable care has gone into their estimation.

A simple and obvious approach would be to count the frequencies of aligned residue pairs and of gaps in confirmed alignments, and to set the probabilities

p_{ab}, q_a and $f(g)$ to the normalised frequencies. (This corresponds to obtaining maximum likelihood estimates for the probabilities; see Chapter 11.)

There are two difficulties with this simple approach. The first is that of obtaining a good random sample of confirmed alignments. Alignments tend not to be independent from each other because protein sequences come in families. The second is more subtle. In truth, different pairs of sequences have diverged by different amounts. When two sequences have diverged from a common ancestor very recently, we expect many of their residues to be identical. The probability p_{ab} for $a \neq b$ should be small, and hence $s(a,b)$ should be strongly negative unless $a = b$. At the other extreme, when a long time has passed since two sequences diverged, we expect p_{ab} to tend to the background frequency $q_a q_b$, so $s(a,b)$ should be close to zero for all a,b. This suggests that we should use scores that are matched to the expected divergence of the sequences we wish to compare.

Dayhoff PAM matrices

Dayhoff, Schwartz & Orcutt [1978] took both these difficulties into consideration when defining their PAM matrices, which have been very widely used for practical protein sequence alignment. The basis of their approach is to obtain substitution data from alignments between very similar proteins, allowing for the evolutionary relationships of the proteins in families, and then extrapolate this information to longer evolutionary distances.

They started by constructing hypothetical phylogenetic trees relating the sequences in 71 families, where each pair of sequences differed by no more than 15% of their residues. To build the trees they used the parsimony method (Chapter 7), which provides a list of the residues that are most likely to have occurred at each position in each ancestral sequence. From this they could accumulate an array A_{ab} containing the frequencies of all pairings of residues a and b between sequences and their immediate ancestors on the tree. The evolutionary direction of this pairing was ignored, both A_{ab} and A_{ba} being incremented each time either an a in the ancestral sequence was replaced by a b in the descendant, or vice versa. Basing the counts on the tree avoided overcounting substitutions because of evolutionary relatedness.

Because they wanted to extrapolate to longer times, the primary value that they needed to estimate was not the joint probability p_{ab} of seeing a aligned to b, but instead the conditional probability $P(b|a,t)$ that residue a is substituted by b in time t. $P(b|a,t) = p_{ab}(t)/q_a$. We can calculate conditional probabilities for a long time interval by multiplying those for a short interval, as shown below. These conditional probabilities are known as *substitution probabilities*; they play an important part in phylogenetic tree building (see Chapter 8). The short time interval estimates for $P(b|a)$ can be derived from the A_{ab} matrix by setting $P(b|a) = B_{a,b} = A_{ab}/\sum_c A_{ac}$.

These values must next be adjusted to correct for divergence time t. The expected number of substitutions in a 'typical' protein, where the residue a occurs at the frequency q_a, is $\sum_{a,b} q_a q_b B_{ab}$. Dayhoff *et al.* defined a substitution matrix to be a 1 PAM matrix (an acronym for 'point accepted mutation') if the expected number of substitutions was 1%, i.e. if $\sum_{a,b} q_a q_b B_{ab} = 0.01$. To turn their B matrix into a 1 PAM matrix of substitution probabilities, they scaled the off-diagonal terms by a factor σ and adjusted the diagonal terms to keep the sum of a row equal to 1. More precisely, they defined $C_{ab} = \sigma B_{ab}$ for $a \neq b$, and $C_{aa} = \sigma B_{aa} + (1-\sigma)$, with σ chosen to make C into a 1 PAM matrix; we will denote this 1 PAM C by $S(1)$. Its entries can be regarded as the probability of substituting a with b in unit time, $P(b|a,t=1)$.

To generate substitution matrices appropriate to longer times, $S(1)$ is raised to a power n (multiplying the matrix by itself n times), giving $S(n) = S(1)^n$. For instance, $S(2)$, the matrix product of $S(1)$ with itself, has entries $P(a|b,t=2) = \sum_c P(a|c,t=1)P(c|b,t=1)$, which are the probabilities of the substitution of b by a occurring via some intermediate, c. For small n, the off-diagonal entries increase approximately linearly with n. Another way to view this is that the matrix $S(n)$ represents the result of n steps of a Markov chain with 20 states, corresponding to the 20 amino acids, each step having transition probabilities given by $S(1)$ (Markov chains will be introduced fully in Chapter 3).

Finally, a matrix of scores is obtained from $S(t)$. Since $P(b|a) = p_{ab}/q_a$, the entries of the score matrix for time t are given by

$$s(a,b|t) = \log \frac{P(b|a,t)}{q_b}.$$

These values are scaled and rounded to the nearest integer for computational convenience. The most widely used matrix is PAM250, which is scaled by $3/\log 2$ to give scores in third-bits.

BLOSUM matrices

The Dayhoff matrices have been one of the mainstays of sequence comparison techniques, but they do have their limitations. The entries in $S(1)$ arise mostly from short time interval substitutions, and raising $S(1)$ to a higher power, to give for instance a PAM250 matrix, does not capture the true difference between short time substitutions and long term ones [Gonnet, Cohen & Benner 1992]. The former are dominated by amino acid substitutions that arise from single base changes in codon triplets, for example L $\leftrightarrow$ I, L $\leftrightarrow$ V or Y $\leftrightarrow$ F, whereas the latter show all types of codon changes.

Since the PAM matrices were made, databases have been formed containing multiple alignments of more distantly related proteins, and these can be used to derive score matrices more directly. One such set of score matrices that is widely used is the BLOSUM matrix set [Henikoff & Henikoff 1992]. In detail, they were derived from a set of aligned, ungapped regions from protein families called the BLOCKS database [Henikoff & Henikoff 1991]. The sequences from each block were clustered, putting two sequences into the same cluster whenever their percentage of identical residues exceeded some level $L\%$. Henikoff & Henikoff then calculated the frequencies A_{ab} of observing residue a in one cluster aligned against residue b in another cluster, correcting for the sizes of the clusters by weighting each occurrence by $1/(n_1 n_2)$, where n_1 and n_2 are the respective cluster sizes.

From the A_{ab}, they estimated q_a and p_{ab} by $q_a = \sum_b A_{ab} / \sum_{cd} A_{cd}$, i.e. the fraction of pairings that include an a, and $p_{ab} = A_{ab} / \sum_{cd} A_{cd}$, i.e. the fraction of pairings between a and b out of all observed pairings. From these they derived the score matrix entries using the standard equation $s(a,b) = \log p_{ab}/q_a q_b$ (2.3). Again, the resulting log-odds score matrices were scaled and rounded to the nearest integer value. The matrices for $L = 62$ and $L = 50$ in particular are widely used for pairwise alignment and database searching, BLOSUM62 being standard for ungapped matching, and BLOSUM50 being perhaps better for alignment with gaps [Pearson 1996]. BLOSUM62 is scaled so that its values are in half-bits, i.e. the log-odds values were multiplied by $2/\log 2$, and BLOSUM50 is given in third-bits. Note that lower L values correspond to longer evolutionary time, and are applicable for more distant searches.

Estimating gap penalties

There is no similar standard set of time-dependent gap models. If there were a time-dependent gap score model, one reasonable assumption might be that the expected number of gaps would increase linearly with time, but their length distribution would stay constant. In an affine gap model, this corresponds to making the gap-open score d linear in $\log t$, while the gap-extend score e would remain constant. Gonnet, Cohen & Benner [1992] derive a similar distribution from empirical data. In fact, they suggest that a better fit is obtained by the form $\gamma(g) = A + B \log t + C \log g$, although there is some circularity in their approach because the data come from a complete comparison of the protein database against itself using sequence alignment algorithms.

In practice, people choose gap costs empirically once they have chosen their substitution scores. This is possible because there are only two affine gap parameters, whereas there are 210 substitution score parameters for proteins. A careful discussion of the factors involved in choosing gap penalties can be found in Vingron & Waterman [1994].

There is a final twist to be added once we have a combined substitution and gap model. Now that there is a possibility of a gap occurring in a sequence at a given position, it is no longer inevitable that there will be a match. It can be argued that we should include in our substitution score a term for the probability that a gap has not opened. The probability that there is a gap in a particular position in sequence x is $\sum_{i\geq 1} f(i)$, and likewise there is the same probability that there is a gap in y at that position. From this we can derive the probability that there is a no gap, i.e. that there is a match:

$$P(\text{no gap}) = 1 - 2\sum_{i\geq 1} f(i). \tag{2.23}$$

As a consequence, the substitution score, which corresponds to a match, should not be $s(a,b)$ but instead $s'(a,b) = s(a,b) + \log P(\text{no gap})$. The effect of this would be to reduce the pairwise scores as gaps become more likely, i.e. as gap penalties decrease. This correction is, however, small, and is not normally made when deriving a scoring system from alignment frequencies.

2.9 Further reading

Good reviews of dynamic programming methods for biological sequence comparison include Pearson [1996] and Pearson & Miller [1992]. The sensitivity of dynamic programming methods has been evaluated and compared to the fast heuristic methods BLAST and FASTA by Pearson [1995] and Shpaer *et al.* [1996].

Bucher & Hofmann [1996] have described a probabilistic version of the Smith–Waterman algorithm, which is related to the methods we will discuss in Chapter 4.

Interesting areas in pairwise dynamic programming alignment that we have not covered include fast 'banded' dynamic programming algorithms [Chao, Pearson & Miller 1992], the problem of aligning protein query sequences to DNA target sequences [Huang & Zhang 1996], and the problem of recovering not only the optimal alignment but also 'suboptimal' or 'near-optimal' alignments [Zuker 1991; Vingron 1996].