	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 $f(g) = ke^{-\lambda g}$.

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 $x_{1\ldots i}$ of x up to x_i and the initial segment $y_{1\ldots j}$ 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
```

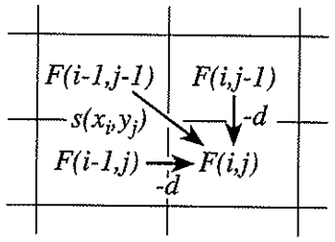
Figure 2.4 *The three ways an alignment can be extended up to (i,j): x_i aligned to y_j, x_i aligned to a gap, and y_j aligned to a gap.*

$F(i,j)$ 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 $x_{1\ldots n}$ to $y_{1\ldots m}$, 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:

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

		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 posi-

tions 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.

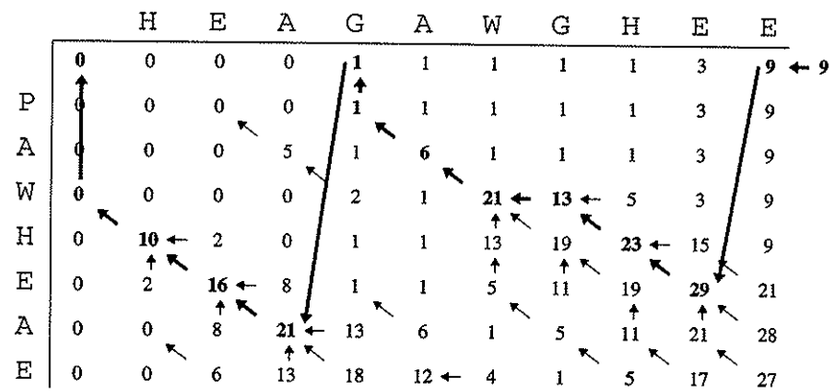


Figure 2.7 *Above, the repeat dynamic programming matrix for the example sequences, for $T = 20$. Below, the optimal alignment, with total score $9 = 29 - 20$. There are two separate match regions, with scores 1 and 8. Dots are used to indicate unmatched regions of x.*

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 $x_{1\ldots i}$, 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 $x_{1\ldots i}$, 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:

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

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

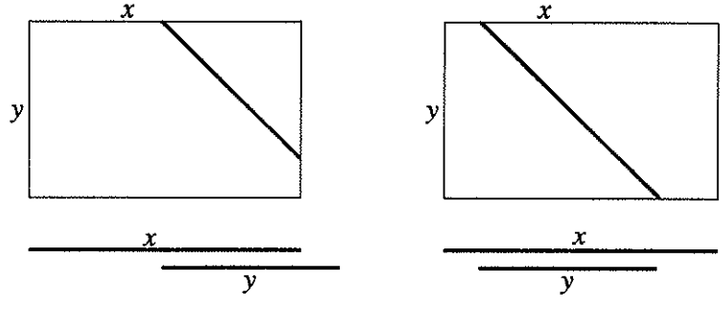
Equation (2.11) handles unmatched regions and ends of matches, only allowing matches to end when they have score at least T. Equation (2.12) handles starts of matches and extensions. The total score of all the matches is obtained by adding an extra cell to the matrix, $F(n+1,0)$, using (2.11). This score 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

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

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

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:

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

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

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

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

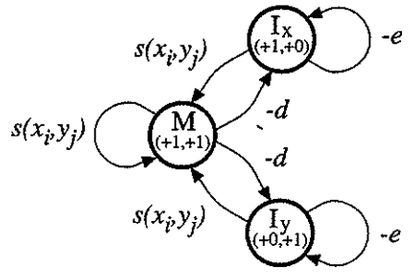


Figure 2.9 *A diagram of the relationships between the three states used for affine gap alignment.*

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

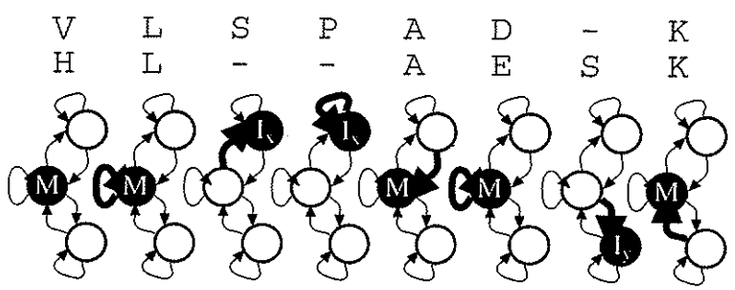


Figure 2.10 *An example of the state assignments for an alignment using the affine gap model.*

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

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

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

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-

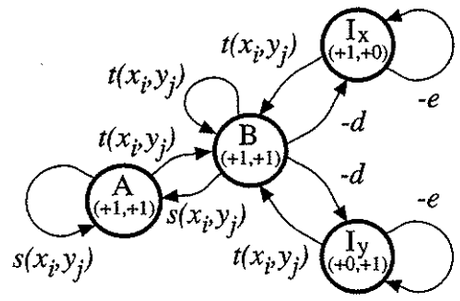


Figure 2.11 *The four-state finite state automaton with separate match states A and B for high and low fidelity regions. Note that this FSA emits on transitions with costs $s(x_i, y_j)$ and $t(x_i, y_j)$, rather than emitting on states, a distinction discussed earlier in the text.*

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].