5

THEORY CHAPTER

PAIRWISE SEQUENCE ALIGNMENT AND DATABASE SEARCHING

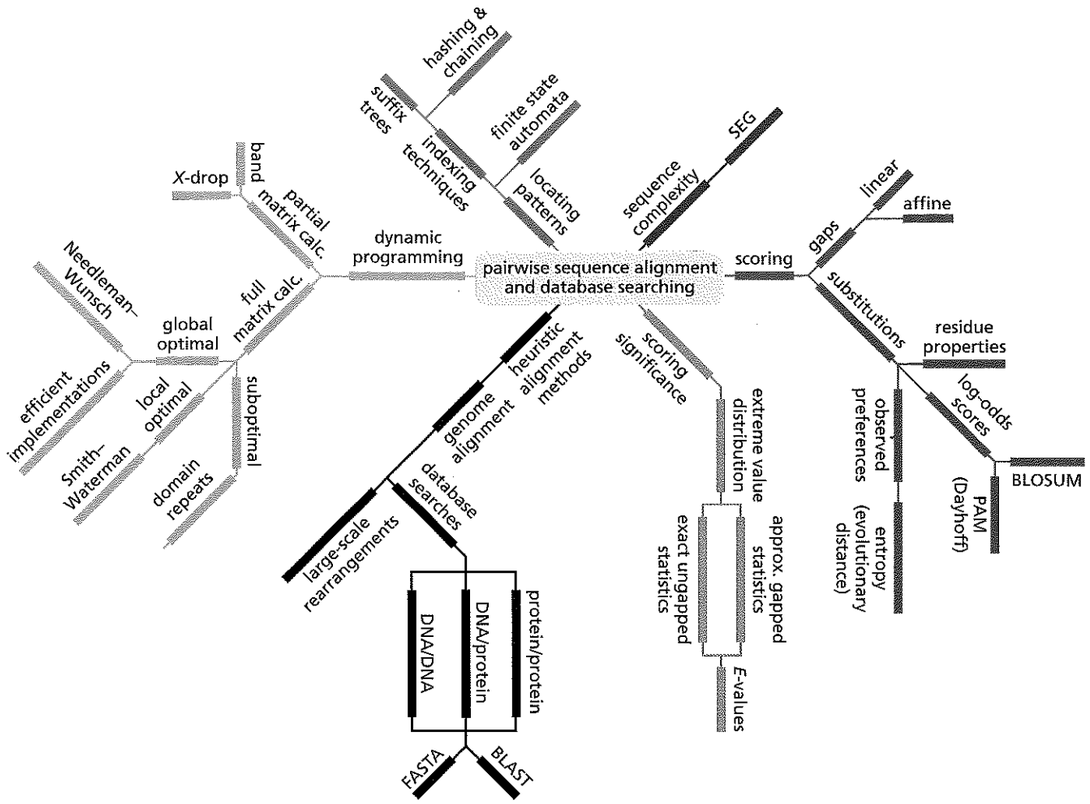
When you have read Chapter 5, you should be able to:

- Compare and contrast different scoring schemes.
- Summarize the techniques for obtaining the best-scoring alignments of a given type.
- Describe ways to reduce the computational resources required.
- Speed up database searches using index techniques.
- Evaluate approximations used in common database search programs.
- Summarize techniques for aligning DNA and protein sequences together.
- Identify sequences of low complexity.
- Identify significant alignments on the basis of their score.
- Summarize the techniques for alignments involving complete genome sequences.

The identification of homologous sequences and their optimal alignment is one of the most fundamental tasks in bioinformatics. As will be seen in many other chapters of this book, there are very few topics in bioinformatics which do not at some stage involve these techniques. A large part of Chapter 4 was devoted to an introduction to some practical aspects of sequence comparison and alignment. In this chapter we will focus in considerable detail on these methods and the science that lies behind them, but will restrict our attention solely to methods of aligning two sequences. The problems of obtaining multiple alignments and profiles and of identifying patterns will be discussed in Chapter 6.

Given two sequences, and allowing gaps to be inserted as was described in Section 4.4, it is possible to construct a very large number of alignments. Of these, there will be an optimal alignment, which in the ideal case perfectly identifies the true equivalences between the sequences. However, there will be many alternative alignments with varying degrees of error that could potentially be seriously misleading. Furthermore, the fact that an alignment can be constructed for any two sequences, even ones with no meaningful equivalences, has the potential to be even more misleading. Therefore, all useful methods of sequence alignment must not only generate alignments but also be able to compare them in a meaningful way and to provide an assessment of their significance.

Both the comparison of alignments and the assessment of their significance require a method of scoring. As discussed in Chapter 4, by considering the evolutionary processes that are responsible for sequence divergence it is possible to find ways of including their salient features in an alignment scoring system. If the scoring



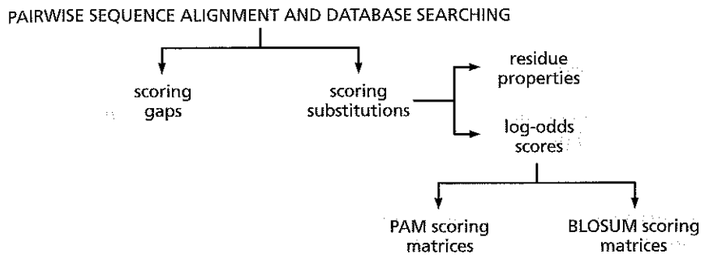
Mind Map 5.1
The theory of pairwise alignment and searching the database depicted in a mind map.

scheme is accurate and appropriate, all meaningful alignments should have optimal scores, i.e., they will have a better score when compared to any alternative. In Section 5.1 some of the best scoring schemes will be presented.

The number of alternative alignments is so great, however, that efficient methods are required to determine those with optimal scores. Fortunately, algorithms have been derived that can be guaranteed to identify the optimal alignment between two sequences for a given scoring scheme. These methods are described in detail in Section 5.2, although it should be noted that the emphasis is mostly on the scientific rather than the computer science aspects. As long as only single proteins or genes, or small segments of genomes, are aligned, these methods can be applied with ease on today's computers. When searching for alignments of a query sequence with a whole database of sequences it is usual practice to use more approximate methods that speed up the search. These are described in Section 5.3.

Finding the best-scoring alignment between two sequences does not guarantee the alignment has any scientific validity. Ways must be found to discriminate between fortuitously good alignments and those due to a real evolutionary relationship. Section 5.4 presents some of the concepts behind the theory of assessing the statistical significance of alignment scores.

The large number of complete genome sequences has led to increased interest in aligning very long sequences such as whole genomes and chromosomes. As described in Section 5.5, these applications require a number of approximations and techniques to increase the speed and reduce the storage requirements. In



Flow Diagram 5.1
The key concept introduced in this section is that if alignments of two sequences are assigned a quantitative score based on evolutionary principles then meaningful comparisons can be made. Several alternative approaches have been suggested, resulting in a number of different scoring schemes including those which account for insertions and deletions.

addition, the presence of large-scale rearrangements in these sequences has required the development of new algorithms.

5.1 Substitution Matrices and Scoring

As discussed in Chapter 4, the aim of an alignment score is to provide a scale to measure the degree of similarity (or difference) between two sequences and thus make it possible to quickly distinguish among the many subtly different alignments that can be generated for any two sequences. Scoring schemes contain two separate elements: the first assigns a value to a pair of aligned residues, while the second deals with the presence of insertions or deletions (indels) in one sequence relative to the other. For protein sequence alignments, reference substitution matrices (see Section 4.3) are used to give a score to each pair of aligned residues. Indels necessitate the introduction of gaps in the alignment, which also have to be scored. The total score S of an alignment is given by summing the scores for individual alignment positions. Special scoring techniques that are applicable only to multiple alignments will be dealt with in Sections 6.1 and 6.4.

One might think that a relatively straightforward way of assessing the probability of the replacement of one amino acid by another would be to use the minimum number of nucleotide base changes required to convert between the codons for the two residues. However, most evolutionary selection occurs at the level of protein function and thus gives rise to significant bias in the mutations that are accepted. Therefore the number of base changes required cannot be expected to be a good measure of the likelihood of substitution. Currently used reference substitution matrices are based on the frequency of particular amino acid substitutions observed in multiple alignments that represent the evolutionary divergence of a given protein. The substitution frequencies obtained thus automatically take account of evolutionary bias.

Two methods that have been used in deriving substitution matrices from multiple sequence alignments will be described here. These have provided two sets of matrices in common use: the PAM and the BLOSUM series. Both these matrices can be related to a probabilistic model, which will be covered first. The key concepts involved in deriving scoring schemes for sequence alignments are outlined in Flow Diagram 5.1.

Alignment scores attempt to measure the likelihood of a common evolutionary ancestor

The theoretical background of alignment scoring is based on a simple probabilistic approach. Two alternative mechanisms could give rise to differences in DNA or protein sequences: a random model and a nonrandom (evolutionary) model. By

generating the probability of occurrence of a particular alignment for each model, an assessment can be made about which mechanism is more likely to have given rise to that alignment.

In the random model, there is no such process as evolution; nor are there any structural or functional processes that place constraints on the sequence and thus cause similarities. All sequences can be assumed to be random selections from a given pool of residues, with every position in the sequence totally independent of every other. Thus for a protein sequence, if the proportion of amino acid type a in the pool is p_a, this fraction will be reproduced in the amino acid composition of the protein.

The nonrandom model, on the other hand, proposes that sequences are related—in other words, that some kind of evolutionary process has been at work—and hence that there is a high correlation between aligned residues. The probability of occurrence of particular residues thus depends not on the pool of available residues, but on the residue at the equivalent position in the sequence of the common ancestor; that is, the sequence from which both of the sequences being aligned have evolved. In this model the probability of finding a particular pair of residues of types a and b aligned is written $q_{a,b}$. The actual values of $q_{a,b}$ will depend on the properties of the evolutionary process.

Suppose that in one position in a sequence alignment, two residues, one type a and the other type b, are aligned. The random model would give the probability of this occurrence as $p_a p_b$, a product, as the two residues are seen as independent. The equivalent value for the nonrandom model would be $q_{a,b}$. These two models can be compared by taking the ratios of the probabilities, called the **odds ratio**, that is $q_{a,b}/p_a p_b$. If this ratio is greater than 1 (that is, $q_{a,b} > p_a p_b$) the nonrandom model is more likely to have produced the alignment of these residues. However, a single model is required that will explain the complete sequence alignment, so all the individual terms for the pairs of aligned residues must be combined.

In practice, there is often a correlation between adjacent residues in a sequence; for example, when they are in a hydrophobic stretch of a protein such as a transmembrane helix (see Chapter 2). This type of correlation is ignored in both of these models, so that each position in an alignment will be regarded as independent. In that case, the odds ratios for the different positions can be multiplied together to give the odds ratio for the entire alignment:

$$\prod_u \left(\frac{q_{a,b}}{p_a p_b} \right)_u \qquad \text{(EQ5.1)}$$

where the product is over all positions u of the alignment.

It is frequently more practical to deal with sums rather than products, especially when small numbers are involved. This can easily be arranged by taking logarithms of the odds ratio to give the **log-odds ratio**. This ratio can be summed over all positions of the alignment to give S, the score for the alignment:

$$S = \sum_u \log \left(\frac{q_{a,b}}{p_a p_b} \right)_u = \sum_u \left(s_{a,b} \right)_u \qquad \text{(EQ5.2)}$$

where $s_{a,b}$ is the score (that is, the **substitution matrix** element) associated with the alignment of residue types a and b. A positive value of $s_{a,b}$ means that the probability of those two residues being aligned is greater in the nonrandom than in the random model. The converse is true for negative $s_{a,b}$ values. S is a measure of the relative likelihood of the whole alignment arising due to the nonrandom model as compared with the random model. However, as discussed later in this chapter, a

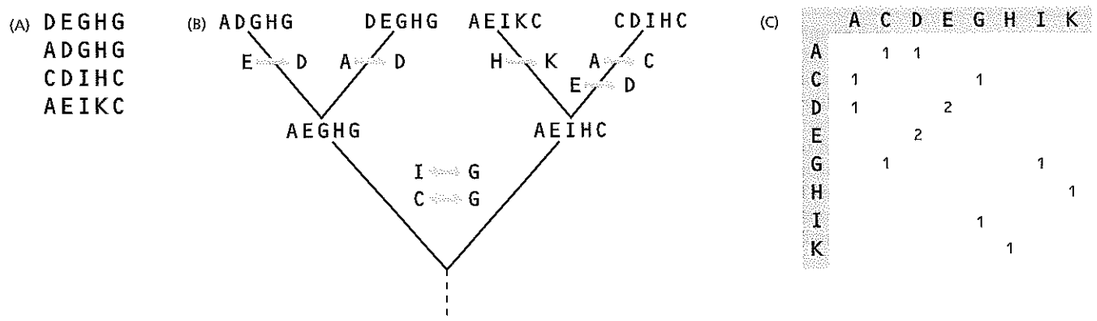


Figure 5.1
The identification of accepted point mutations in the derivation of the PAM/MDM amino acid substitution scoring matrices. (A) Part of an alignment that might have occurred in the protein sequence data used to obtain the Dayhoff PAM matrices. (B) The phylogenetic tree derived from this alignment. Note that this tree hypothesizes the smallest number of mutations required to explain the observed sequences, but does not necessarily represent the actual evolutionary history. See Chapters 7 and 8 for further details on phylogenetic trees. The direction of some mutations cannot be resolved due to lack of knowledge of the oldest ancestral sequence. This is also the case for the other ancestral sequences shown, which are specified here for clarity only. (C) The matrix of accepted substitutions derived from this tree. Note that all observed mutations have been assumed to be equally likely in either direction, and hence are counted in both directions.

positive S is not a sufficient test of the alignment's significance. There will be a distribution of values of S for a given set of sequences, which can be used to determine significant scores.

From this discussion one would expect there to be both positive and negative $s_{a,b}$ values. In practice this is not always the case, because each $q_{a,b}/p_a p_b$ term can be multiplied by a constant. Multiplication by a constant X will result in a term $\log X$ being added to each $s_{a,b}$. This could result in all $s_{a,b}$ values being positive, for example. The alignment score S is shifted by $L_{aln} \log X$ for an alignment of length L_{aln}. Similarly, all the $s_{a,b}$ values can be multiplied by a constant. In both cases, scores of alternative alignments of the same length retain the same relative relationship to each other. However, local alignments discussed below involve comparing alignments of different lengths, in which case adding a constant $\log X$ to each $s_{a,b}$ will have an effect on the relative scores.

Note that the link between substitution matrices and the log-odds ratio described by Equation EQ 5.2 may exist even if the matrix was derived without any reference to this theory. Firstly, note that the **expected score** for aligning two residues can be written

$$E\left(s_{a,b}\right) = \sum_{a,b} p_a p_b s_{a,b} \qquad \text{(EQ5.3)}$$

If this is negative, and there is at least one positive score, one can in principle perform the reverse procedure to obtain the $q_{a,b}$ given the $s_{a,b}$ and the p_a. The procedure is not entirely straightforward, and interested readers should refer to the Further Reading at the end of the chapter.

The PAM (MDM) substitution scoring matrices were designed to trace the evolutionary origins of proteins

As we saw in Section 4.3, one commonly used type of matrix for protein sequences is the point accepted mutations (PAM) matrix, also known as the mutation data matrix (MDM), derived by Margaret Dayhoff and colleagues from the analysis of multiple alignments of families of proteins, including cytochrome c, α- and β-globin (the protein chains of which hemoglobin is composed), insulin A and B chains, and ferredoxin.

These raw data are biased by the amino acid composition of the sequences, the differing rates of mutation in different protein families, and sequence length. The first step in an attempt to remove this bias is to calculate, for each alignment, the exposure of each type of amino acid residue to mutation. This is defined as the compositional fraction of this residue type in the alignment multiplied by the total number of mutations (of any kind) that have occurred per 100 alignment positions.

See below for a description of how the number of mutations is calculated. The value for each residue type is summed over all the alignments in the data set to give the total exposure of that residue type. The mutability of a specific residue type a is defined as the ratio of the total number of mutations in the data that involve residue a divided by the total exposure of residue a. Usually this is reported relative to alanine as a standard, and is referred to as the **relative mutability**, m_a. A few residues have higher mutability than alanine, but more notable are those that are less likely to change, especially cysteine and tryptophan. The mutability of these residues is approximately one-fifth that of alanine.

Phylogenetic trees are constructed for each alignment by a method that infers the most likely ancestral sequence at each internal node and hence postulates all the mutations that have occurred. The observed substitutions are tabulated in a matrix, ***A*** **(accepted point mutation matrix)** according to the residue types involved (see Figure 5.1). It is assumed that a mutation from residue type a to type b was as likely as a mutation from b to a, so all observed mutations are included in the count for both directions. Where there is uncertainty as to which mutations have occurred, all possibilities are treated as equally likely, and fractional mutations are added to the matrix of accepted substitutions. The matrix element values are written $A_{a,b}$. Dayhoff's 1978 dataset contained 1572 observed substitutions, but even so, 35 types of mutations had not been observed, for example tryptophan (W) → alanine (A). This was due to the relatively small dataset of highly similar sequences that was used.

From this information a **mutation probability matrix**, ***M***, can be defined. Each element $M_{a,b}$ gives the probability that a residue of type b will be replaced by one of type a after a given period of evolutionary time. The residue b has a likelihood of mutating that is proportional to m_b. The expected fraction of mutations of b into residue a can be obtained from the accepted point mutation matrix (see Figure 5.1C) with elements $A_{a,b}$. Thus the off-diagonal ($a \neq b$) terms of M are given by the formula

$$M_{a,b} = \frac{\Lambda m_b A_{a,b}}{\sum_a A_{a,b}} \qquad \text{(EQ5.4)}$$

where Λ is a constant that accounts in part for the proportionality constant of m_b and in part for the unspecified evolutionary time period. Note that matrix ***M*** is not symmetrical because of the residue relative mutability m_b. The diagonal terms ($a = b$) of this matrix, corresponding to no residue change, are

$$M_{b,b} = 1 - \Lambda m_b \qquad \text{(EQ5.5)}$$

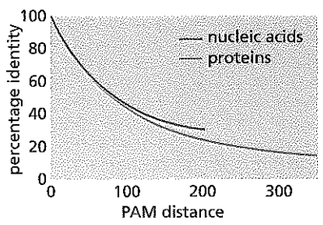
Note that the sum of all off-diagonal elements $M_{a,b}$ involving mutation from a given residue b to another type and the element $M_{b,b}$ equals 1, as required for ***M*** to be a probability matrix.

The percentage of amino acids unchanged in a sequence of average composition after the evolutionary change represented by the matrix ***M*** can be calculated as

$$100\sum_b f_b M_{b,b} = 100\sum_b f_b\left(1 - \Lambda m_b\right) \qquad \text{(EQ5.6)}$$

where f_b is approximately the frequency of residue type b in the average composition. In fact f_b is the total exposure of residue b normalized so that the sum of all values is 1, and is the residue composition weighted by the sequence mutation rate. The value of Λ is selected to determine this percentage of unchanged residues. If Λ is chosen to make this sum 99, the matrix represents an evolutionary change of 1 PAM (that is, one accepted mutation per 100 residues).

Figure 5.2
The relationship between the evolutionary distance in PAMs and the percentage identity between two sequences. This shows that the evolutionary model predicts considerable identity even between very distantly related sequences.



	C	S	T	P	A	G	N	D	E	Q	H	R	K	M	I	L	V	F	Y	W
C	11																			
S	1	2																		
T	-1	1	2																	
P	-2	1	1	6																
A	-1	1	2	1	2															
G	-1	1	-1	-1	1	5														
N	-1	1	1	-1	0	0	3													
D	-3	0	-1	-2	0	1	2	5												
E	-4	-1	-1	-2	-1	0	1	4	5											
Q	-3	-1	-1	0	-1	-1	0	1	2	5										
H	0	-1	-1	0	-2	-2	1	0	0	2	6									
R	-1	-1	-1	-1	-1	0	0	-1	0	2	2	5								
K	-3	-1	-1	-2	-1	-1	1	0	1	2	1	4	5							
M	-2	-1	0	-2	-1	-3	-2	-3	-3	-2	-2	-2	-2	6						
I	-2	-1	1	-2	0	-3	-2	-3	-3	-3	-3	-3	-3	3	4					
L	-3	-2	-1	0	-1	-4	-3	-4	-4	-2	-2	-3	-3	3	2	5				
V	-2	-1	0	-1	1	-2	-2	-2	-2	-3	-3	-3	-3	2	4	2	4			
F	0	-2	-2	-3	-3	-5	-3	-5	-5	-4	0	-4	-5	0	0	2	0	8		
Y	2	-1	-3	-3	-3	-4	-1	-2	-4	-2	4	-2	-3	-2	-2	-1	-3	5	9	
W	1	-3	-4	-4	-4	-2	-5	-5	-5	-3	-3	0	-3	-3	-4	-2	-3	-1	0	15

Figure 5.3
The PET91 version of the PAM250 substitution matrix. Scores that would be given to identical matched residues are in blue; positive scores for nonidentical matched residues are in red. The latter represent pairs of residues for which substitutions were observed relatively often in the aligned reference sequences.

According to the Dayhoff model of evolution, to obtain the probability matrices for higher percentages of accepted mutations, the 1-PAM matrix is multiplied by itself. This is because the model of evolution proposed was a Markov process (Markov models are described in detail in Section 10.3). If the 1-PAM matrix is squared, it gives a 2-PAM matrix; if cubed, a 3-PAM matrix; and so on. These correspond to evolutionary periods twice and three times as long as the period used to derive the 1-PAM matrix. Similarly, a 250-PAM matrix is obtained by raising the 1-PAM matrix to the 250th power.

These matrices tell us how many mutations have been accepted, but not the percentage of residues that have mutated: some may have mutated more than once, others not at all. For each of these matrices, evaluation of Equation EQ5.6 gives the actual percentage identity to the starting sequence expected after the period of evolution represented by the matrix. The percentage identity does not decrease linearly with time in this model (see Figure 5.2).

So far, a probability matrix has been obtained, but not a scoring matrix. As discussed earlier, such a score should involve the ratio of probabilities derived from nonrandom and random models. The matrix ***M*** gives the probability of residue b mutating into residue a if the two sequences are related, that is, if substitution is nonrandom. It already includes a term for the probability of occurrence of residue b in the total exposure term used to calculate m_b. Thus the only term needed for the random-model probability is f_a, the likelihood of residue a occurring by chance. Hence, the scoring matrix was originally derived from ***M*** by the formula

$$s_{a,b} = 10\log_{10}\left(\frac{M_{a,b}}{f_a}\right) \qquad \text{(EQ5.7)}$$

and is in fact a symmetrical matrix. The resultant scoring matrices s are usually named PAMn, where n is the number of accepted point mutations per 100 residues in the probability matrix $\boldsymbol{M}$ from which they are derived. The exact scaling factors and logarithm base are to some degree arbitrary, and are usually chosen to provide integer scores with a suitable number of significant figures. In Figure 4.4B the PAM120 matrix is shown, but a scaling factor of $2\log_2$ has been used instead of $10\log_{10}$, so that the values are in units of half bits (a bit is a measure of information) (see Appendix A).

There was a lack of sequence data available when the original work was done to derive the PAM matrices, and in 1991 the method was applied to a larger dataset, producing the PET91 matrix that is an updated version of the original PAM250 matrix (see Figure 5.3). This matrix shows considerable differences for aligning two tryptophan (W) residues, with a score of 15, as opposed to two alanine residues (A), which score only 2. About one-fifth of the scores for nonidentical residues have positive scores, considerably more than occurs in PAM120 (see Figure 4.4B), reflecting the longer evolutionary period represented by the matrix.

The BLOSUM matrices were designed to find conserved regions of proteins

The BLOCKS database containing large numbers of ungapped multiple local alignments of conserved regions of proteins, compiled by Steven and Jorja Henikoff, became available in 1991 (see Section 4.8). These alignments included distantly related sequences, in which multiple base substitutions at the same position could be observed. The BLOCKS database was soon recognized as a resource from which substitution preferences could be determined, leading to the BLOSUM substitution score matrices.

There are two contrasts with the data analysis used to obtain the PAM matrices. First, the BLOCKS alignments used to derive the BLOSUM matrices include sequences that are much less similar to each other than those used by Dayhoff, but whose evolutionary homology can be confirmed through intermediate sequences. In addition, these alignments are analyzed without creating a phylogenetic tree and are simply compared with each other.

A direct comparison of aligned residues does not model real substitutions, because in reality the sequences have evolved from a common ancestor and not from each other. Nevertheless, as the large sequence variation prevents accurate construction of a tree, there is no alternative. However, if the alignment is correct, then aligned residues will be related by their evolutionary history and therefore their alignment will contain useful information about substitution preferences. Another argument in favor of direct analysis of aligned sequence differences is that often the aim is not to recreate the evolutionary history, but simply to try to align sequences to test them for significant similarity. The intention in producing the BLOSUM matrices was to find scoring schemes that would identify conserved regions of protein sequences.

One of the key aspects of the analysis of the alignment blocks is to weight the sequences to try to reduce any bias in the data. This is necessary because the sequence databases are highly biased toward certain species and types of proteins, which means there are many very similar sequences present. The weighting involves clustering the most similar sequences together, and different matrices are produced according to the threshold C used for this clustering. Sequences are clustered together if they have $\geq C\%$ identity, and the substitution statistics are calculated only between clusters, not within them. Weights are determined according to the number of sequences in the cluster. For a cluster of N_{seq} sequences, each sequence is assigned a weight of $1/N_{seq}$. The weighting scheme was used to obtain a series of substitution matrices by varying the value of C, with the matrices named

(A)

	1	2	3	4	5
1	A	T	C	K	Q
2	A	T	C	R	N
3	A	S	C	K	N
4	S	S	C	R	N
5	S	D	C	E	Q
6	S	E	C	E	N
7	T	E	C	R	Q

(B)

	q_{QN}	q_{NN}	q_{QQ}	p_N	p_Q
C=62%	0.114	0.057	0.029	0.114	0.086
C=50%	0.117	0.025	0.058	0.084	0.117
C=40%	–	–	–	–	–

Figure 5.4
Derivation of the BLOSUM amino acid substitution scoring matrices. (A) An example ungapped alignment block with one fully conserved position (cysteine) colored red. In this case, sequences have been clustered if they are at least 50% identical to any other within the cluster. Thus for example, although sequences 1 and 4 are only 20% identical to each other, they are in the same cluster, as they are both 60% identical to sequence 2. Similar clustering of sequence data was used to derive the BLOSUM-50 matrix. If the same data were used to derive the BLOSUM-62 matrix (that is, C = 62%) none of these sequences would cluster together, as no two of them share more than 60% identical residues, leading to very different sequence weights. (B) Values of $q_{a,b}$ and p_a for asparagine (N) and glutamine (Q) residues for three different values of C. For C = 40% all the sequences belong in the same cluster, and since $q_{a,b}$ and p_a measure intercluster alignment statistics, they cannot be calculated in this case.

as BLOSUM-62, for example, in the case where C = 62%. The sequences of all the alignments used to obtain the Dayhoff matrices fall in a single cluster for $C \leq 85\%$, indicating that they were much more similar to each other than those used for BLOSUM.

The derivation of substitution data will be illustrated using the example alignment in Figure 5.4A and the case of C = 50%, which would lead to the BLOSUM-50 matrix. There are three sequence clusters, with four, two, and one sequences, giving weights to their constituent sequences of $^1/_4$, $^1/_2$, and 1, respectively. These weights are applied to the counts of observed aligned residues to produce the weighted frequencies $f_{a,b}$. The observed probability ($q_{a,b}$) of aligning residues of types a and b is given by

$$q_{a,b} = \frac{f_{a,b}}{\sum_{1 \leq b \leq a}^{20} f_{a,b}} \qquad \text{(EQ5.8)}$$

Note that this ignores which sequence the residue a or b has come from, so that $f_{a,b}$ and $f_{b,a}$, and hence $q_{a,b}$ and $q_{b,a}$, are equal.

Consider the calculation of $q_{a,b}$ for asparagine (N) and glutamine (Q) residues, which occur only in column 5 in Figure 5.4A. If C = 62%, then all sequence clusters will contain just one sequence and each sequence will have a weight of 1. (No pair of sequences share more than 60% identical residues.) In this case there are 21 ($7 \times 6/2$) distinct cluster pairs and thus 21 pairs of aligned residues in any single alignment column. Counting these, there are 12 QN pairs (= $f_{Q,N}$), 3 QQ (= $f_{Q,Q}$), and 6 NN (= $f_{N,N}$), making a total of 21 pairs. As all sequence weights are 1, they play no real part in this calculation. As there are five alignment columns, the total number of aligned pairs (the denominator of Equation EQ5.8) is 105. From these data the $q_{a,b}$ can be obtained, as listed in Figure 5.4B. Note that if other columns had contained N and Q, they would also have needed to be included in the calculation.

Considering the case of C = 50%, the sequences separate into three clusters, so the total number of cluster pairs at position 5 will be 3 ($3 \times 2/2$). The top, middle, and bottom clusters can be regarded as being {$^1/_4$Q, $^3/_4$N}, {$^1/_2$Q, $^1/_2$N}, and {Q}, respectively, at this position. Remembering to consider only pairs between clusters and not within them, the weighted number of QN aligned pairs is calculated as

$$f_{Q,N} = \left(\frac{1}{4} \times \frac{1}{2}\right) + \left(\frac{3}{4} \times \frac{1}{2}\right) + \left(\frac{3}{4} \times 1\right) + \left(\frac{1}{2} \times 1\right) = \frac{14}{8} \qquad \text{(EQ5.9)}$$

where the first term is for Q residues of the top cluster and N residues of the second cluster; the second term is for N residues of the top cluster and Q residues of the

Figure 5.5
The BLOSUM-62 substitution matrix scores in half bits. Scores that would be given to identical matched residues are in blue; positive scores for nonidentical matched residues are in red. The latter represent pairs of residues for which substitutions were observed relatively often in the aligned reference sequences.

	C	S	T	P	A	G	N	D	E	Q	H	R	K	M	I	L	V	F	Y	W
C	9																			
S	-1	4																		
T	-1	1	5																	
P	-3	-1	-1	7																
A	0	1	0	-1	4															
G	-3	0	-2	-2	0	6														
N	-3	1	0	-2	-2	0	6													
D	-3	0	-1	-1	-2	-1	1	6												
E	-4	0	-1	-1	-1	-2	0	2	5											
Q	-3	0	-1	-1	-1	-2	0	0	2	5										
H	-3	-1	-2	-2	-2	-2	1	-1	0	0	8									
R	-3	-1	-1	-2	-1	-2	0	-2	0	1	0	5								
K	-3	0	-1	-1	-1	-2	0	-1	1	1	-1	2	5							
M	-1	-1	-1	-2	-1	-3	-2	-3	-2	0	-2	-1	-1	5						
I	-1	-2	-1	-3	-1	-4	-3	-3	-3	-3	-3	-3	-3	1	4					
L	-1	-2	-1	-3	-1	-4	-3	-4	-3	-2	-3	-2	-2	2	2	4				
V	-1	-2	0	-2	0	-3	-3	-3	-2	-2	-3	-3	-2	1	3	1	4			
F	-2	-2	-2	-4	-2	-3	-3	-3	-3	-3	-1	-3	-3	0	0	0	-1	6		
Y	-2	-2	-2	-3	-2	-3	-2	-3	-2	-1	2	-2	-2	-1	-1	-1	-1	3	7	
W	-2	-3	-2	-4	-3	-2	-4	-4	-3	-2	-2	-3	-3	-1	-3	-2	-3	1	2	11

second cluster; the third term is for the Q residue of the third cluster; and the fourth term is for the N residues of each of the other clusters. The equivalent values for $f_{N,N}$ and $f_{Q,Q}$ are 3/8 and 7/8, respectively. Dividing these $f_{a,b}$ values by 15 (the weighted total number of aligned pairs in the data, three for each alignment position) gives the $q_{a,b}$ values shown in Figure 5.4B. The values for $f_{Q,Q}$ and $f_{N,N}$ differ between C = 62% and C = 50%, because in the latter case most of the N residues are in one cluster. Note that in the case of C = 40% there is only one cluster, as each sequence is at least 40% identical to at least one other, so that no intercluster residue pairing exists to be counted! This example shows how the value of C can affect the derived substitution matrix in a complicated manner.

The scores for the residue pairs are obtained using the log-odds approach described earlier. The estimate $e_{a,b}$ of the probability of observing two residues of type a and b aligned by chance is given (as in deriving PAM matrices) by a weighted composition of the sequences, with p_a being approximately the fraction of all residues that is type a. The background residue frequencies p_a are defined by

$$p_a = q_{a,a} + \sum_{a \neq b} \frac{q_{a,b}}{2} \qquad \text{(EQ5.10)}$$

Figure 5.6
Three nucleotide substitution scoring matrices, derived by Chiaromonte and co-workers. Each matrix was obtained by analysis of alignments of distinct regions of the human and mouse genomes with different G+C content. (A) was derived from the CFTR region which is 37% G+C; (B) was derived from the HOXD region which is 47% G+C; (C) was derived from the hum16pter region which is 53% G+C. Each matrix was scaled to give a maximum score of 100.

(A)

	A	C	G	T
A	67	-96	-20	-117
C	-96	100	-79	-20
G	-20	-79	100	-96
T	-117	-20	-96	67

(B)

	A	C	G	T
A	91	-114	-31	-123
C	-114	100	-125	-31
G	-31	-125	100	-114
T	-123	-31	-114	91

(C)

	A	C	G	T
A	100	-123	-28	-109
C	-123	91	-140	-28
G	-28	-140	91	-123
T	-109	-28	-123	100

Figure 5.4B shows how these are affected by the choice of C. For identical residues, $e_{a,a}$ is p_a^2, whereas for different residues, $e_{a,b}$ is $2p_ap_b$, because there are two ways in which one can select two different residues. The BLOSUM matrices are defined using information measured in bits, so the formula for a general term is

$$s_{a,b} = \log_2\left(\frac{q_{a,b}}{e_{a,b}}\right) \qquad \text{(EQ5.11)}$$

One of the most commonly used of these matrices is BLOSUM-62, which is shown in Figure 5.5 in units of half bits, obtained by multiplying the $s_{a,b}$ in Equation EQ5.11 by 2.

Scoring matrices for nucleotide sequence alignment can be derived in similar ways

The methods described above for deriving scoring matrices have been illustrated with examples from protein sequences. The same techniques can be applied to nucleotides, although often simple scoring schemes such as +5 for a match and –4 for a mismatch are used. With certain exceptions, such as 16S rRNA and repeat sequences, until quite recently almost all sequences studied were for protein-coding regions, in which case it is usually advantageous to align the protein sequences. This has changed with the sequencing of many genomes and the alignment of long multigene segments or even whole genomes.

Matrices have been reported that are derived from alignments of human and mouse genomic segments (see Figure 5.6). The different matrices were derived from sequence regions with different G+C content, as it is thought that this influences the substitution preferences. This is to be contrasted with the different PAM and BLOSUM matrices, which are based on evolutionary distance. The matrices were derived using a similar approach to that described for the BLOSUM series. However, there is a difference worth noting when dealing with DNA sequences. Any alignment of DNA sequences implies a second alignment of the complementary strand. Thus, every observation of a T/C aligned pair implies in addition an aligned A/G pair.

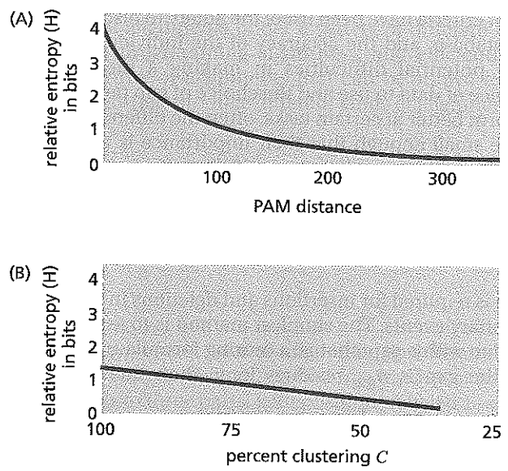


Figure 5.7
Plots of the relative entropy H in bits for two different series of substitution matrices. (A) Plot for the PAM matrices according to PAM distance (Data from Altschul, 1991.) (B) Plot for the BLOSUM matrices according to the percentage cut-off C used in clustering alignment blocks. (Data from Henikoff and Henikoff, 1992.)

The substitution scoring matrix used must be appropriate to the specific alignment problem

Many other substitution scoring matrices have been derived. Some are based on alternative ways of analyzing the alignment data, while others differ in the dataset used. Some are intended for specialized use, such as aligning transmembrane sequences. The matrices can be compared in three ways, focusing on (1) the relative patterns of scores for different residue types, (2) the actual score values, or (3) the practical application of the matrices.

Cluster analysis of the scores can be used to see if matrices distinguish between the amino acids in different ways. For example, one matrix may be strongly dominated by residue size, another by polarity. This may improve our understanding of the evolutionary driving forces or, alternatively, highlight shortcomings in the sequence data used to derive the matrix, but probably will not assist in determining the relative usefulness of matrices for constructing alignments.

In common with most proposed amino acid substitution matrices, the PAM and BLOSUM matrix series include both positive and negative scores, with the average score being negative. The actual score values can be summarized in two measures that have some practical use. The **relative entropy** (H) of the nonrandom model with respect to the random model is defined as

$$H = \sum_{a,b} q_{a,b} s_{a,b} = \sum_{a,b} q_{a,b} \log\left(\frac{q_{a,b}}{p_a p_b}\right) \qquad \text{(EQ5.12)}$$

The scores $s_{a,b}$ are summed, weighted by $q_{a,b}$, and H is a measure of the average information available at each alignment position to distinguish between the nonrandom and random models, and is always positive. (See Appendix A for further discussion of this measure.) Figure 5.7 shows the variation of H for different PAM and BLOSUM matrices. The shortest local alignment that can have a significant score is in part dependent on the relative entropy of the scoring matrix used, as discussed later in the chapter. The other measure of score values is the expected score, defined in Equation EQ5.3, which is usually—but not necessarily—negative, for example –0.52 for BLOSUM-62. This measure has been found to influence the variation of alignment scores with alignment length.

Perhaps the best way to compare matrices is to see how well they perform with real data. Two different criteria have been used: the ability to discover related sequences in searching a database, and the accuracy of the individual alignments derived. There are many potential difficulties in making a meaningful comparison, including the choice of data to use and the choice of gap penalties. Although some matrices do perform better at certain tasks, often the differences for the commonly used matrices are small enough that their importance is unclear, especially as a poor choice of gap penalties can have a significant effect.

Gaps are scored in a much more heuristic way than substitutions

A scoring scheme is required for insertions and deletions in alignments, as they are common evolutionary events. The simplest method is to assign a gap penalty g on aligning any residue with a gap; that is, a scoring formula $g = -E$, where E is a positive number. If the gap is n_{gap} residues long, then this **linear gap penalty** is defined as

$$g(n_{\text{gap}}) = -n_{\text{gap}} E \qquad \text{(EQ5.13)}$$

Usually, no account is taken of the type of residue aligned with the gap, although making the value of E vary with residue type would easily do this. The observed preference for fewer and longer gaps can be modeled by using a higher penalty to initiate a gap [the **gap opening penalty** (GOP), designated I] and then a lower penalty to extend an existing gap [the **gap extension penalty** (GEP), designated E]. This leads to the **affine gap penalty** formula

$$g(n_{\text{gap}}) = -I - (n_{\text{gap}} - 1)E \qquad \text{(EQ5.14)}$$

for a gap of n_{gap} residues. Note that an alternative definition can give rise to the formula

$$g(n_{\text{gap}}) = -I - n_{\text{gap}} E \qquad \text{(EQ5.15)}$$

Again, residue preferences can easily be added to this scheme by varying the value of E.

The values of the gap parameters need to be carefully chosen for the specific substitution scoring matrix used. Failure to optimize these parameters can significantly degrade the performance of the overall scoring scheme. This is illustrated by the worked example later in the chapter. Some matrices seem less sensitive to gap parameterization than others. In practice, as optimization is a lengthy process, most workers use previously reported optimal combinations. Typical ranges of the parameters for protein alignment are 7–15 for I and 0.5–2 for E.

5.2 Dynamic Programming Algorithms

For any given pair of sequences, if gaps are allowed there is a large number of possibilities to consider in determining the best-scoring alignment. For example, two sequences of length 1000 have approximately 10^{600} different alignments, vastly more than there are particles in the universe! Given the number and length of known sequences it would seem impossible to explore all these possibilities. Nevertheless, a class of algorithms has been introduced that is able to efficiently explore the full range of alignments under a variety of different constraints. They are known as dynamic programming algorithms, and efficiently avoid needless exploration of the majority of alignments that can be shown to be nonoptimal. There are several variants that produce different kinds of alignments, as outlined in Flow Diagram 5.2.

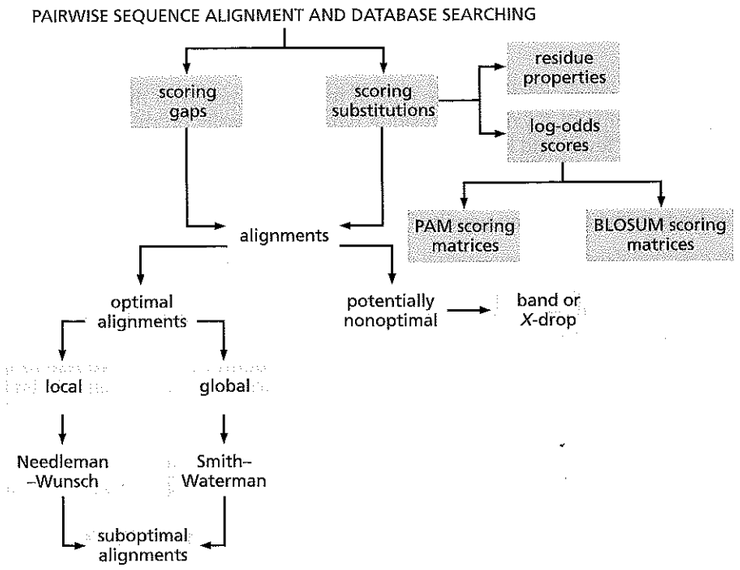
The key property of dynamic programming is that the problem can be divided into many smaller parts. Consider the following alignment:

$$X_1 \cdots X_u \; X_{u+1} \cdots X_v \; X_{v+1} \cdots X_L$$

$$Y_1 \cdots Y_u \; Y_{u+1} \cdots Y_v \; Y_{v+1} \cdots Y_L$$

in which the subscripts u, v, etc. refer to alignment positions rather than residue types, so that X_u, Y_v, and so on each correspond to a residue or to a gap. The alignment has been divided into three parts, with positions labeled $1 \rightarrow u$, $u+1 \rightarrow v$, and

Flow Diagram 5.2
The key concept introduced in this section is that the method of dynamic programming can be applied with minor modifications to solve several related problems of determining optimal and near-optimal pairwise sequence alignments.



$v + 1 \rightarrow L$. Because scores for the individual positions are added together, the score of the whole alignment is the sum of the scores of the three parts; that is, their contributions to the score are independent. Thus, the optimal global alignment can be reduced to the problem of determining the optimal alignments of smaller sections. A corollary to this is that the global optimal alignment will not contain parts that are not themselves optimal. While affine gap penalties require a slightly more sophisticated argument, essentially the same property holds true for them as well.

Starting with sufficiently short sub-sequences, for example the first residue of each sequence, the optimal alignment can easily be determined, allowing for all possible gaps. Subsequently, further residues can be added to this, at most one from each sequence at any step. At each stage, the previously determined optimal subsequence alignment can be assumed to persist, so only the score for adding the next residue needs to be investigated. A worked example later in the section will make this clear. In this way the optimal global alignment can be grown from one end of the sequence. As an alignment of two sequences will consist of pairs of aligned residues, a rectangular matrix can conveniently represent these, with rows corresponding to the residues of one sequence, and columns to those of the other.

Until the global optimal alignment has been obtained, it is not known which actual residues are aligned. All possibilities must be considered or the optimal alignment could be missed. This is not as impossible as it might seem.

Saul Needleman and Christian Wunsch published the original dynamic programming application in this field in 1970, since then many variations and improvements have been made, some of which will be described here. There have been three different motivations for developing these modifications. Firstly, global and local alignments require slightly different algorithms. Secondly, but less commonly, certain gap-penalty functions and the desire to optimize scoring parameters have resulted in further new schemes. Lastly, especially in the past, the computational requirements of the algorithms prevented some general applications. For example, the basic technique in a standard implementation requires computer memory proportional to the product mn for two sequences of length m and n. Some algorithms have been proposed that reduce this demand considerably.

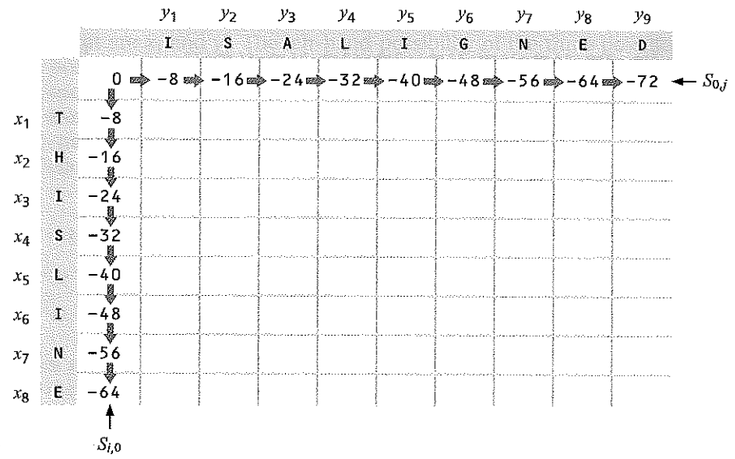


Figure 5.8
The initial stage of filling in the dynamic programming matrix to find the optimal global alignment of the two sequences THISLINE and ISALIGNED. The initial stage of filling in the matrix depends only on the linear gap penalty, defined in Equation EQ5.13 with E set to –8. The arrows indicate the cell(s) to which each cell value contributes.

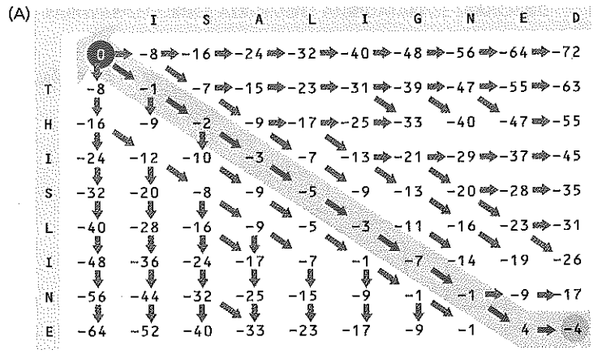
Optimal global alignments are produced using efficient variations of the Needleman–Wunsch algorithm

We will introduce dynamic programming methods by describing their use to find optimal global alignments. Needleman and Wunsch were the first to propose this method, but the algorithm given here is not their original one, because significantly faster methods of achieving the same goal have since been developed. The problem is to align sequence x ($x_1x_2x_3...x_m$) and sequence y ($y_1y_2y_3...y_n$), finding the best-scoring alignment in which all residues of both sequences are included. The score will be assumed to be a measure of similarity, so that the highest score is desired. Alternatively, the score could be an evolutionary distance (see Section 4.2), in which case the smallest score would be sought, and all uses of "max" in the algorithm would be replaced by "min."

The key concept in all these algorithms is the matrix $\boldsymbol{S}$ of optimal scores of subsequence alignments. The matrix has $(m + 1)$ rows labeled $0 \rightarrow m$ and $(n + 1)$ columns labeled $0 \rightarrow n$. The rows correspond to the residues of sequence $\boldsymbol{x}$, and the columns to those of sequence $\boldsymbol{y}$ (see Figure 5.8). We shall use as a working example the alignment of the sequences $\boldsymbol{x}$ = THISLINE and $\boldsymbol{y}$ = ISALIGNED, with the BLOSUM-62 substitution matrix as the scoring matrix (see Figure 5.5). Because the sequences are small they can be aligned manually, and so we can see that the optimal alignment is:

```
TH I S - L I - NE -
   |  |  |  |  |  |
-- I S A L I GNE D
```

This alignment might not produce the optimal score if the gap penalty were set very high relative to the substitution matrix values, but in this case it could be argued that the scoring parameters would then not be appropriate for the problem. In the matrix in Figure 5.8, the element $S_{i,j}$ is used to store the score for the optimal alignment of all residues up to x_i of sequence $\boldsymbol{x}$ with all residues up to y_j of sequence $\boldsymbol{y}$. The sequences $(x_1x_2x_3...x_i)$ and $(y_1y_2y_3...y_j)$ with $i < m$ and $j < n$ are called subsequences. Column $S_{i,0}$ and row $S_{0,j}$ correspond to the alignment of the first i or j residues with the same number of gaps. Thus, element $S_{0,3}$ is the score for aligning sub-sequence $y_1y_2y_3$ with a gap of length 3.



(B) THISLINE-
||
ISALIGNED

Figure 5.9
The dynamic programming matrix (started in Figure 5.8) used to find the optimal global alignment of the two sequences THISLINE and ISALIGNED. (A) The completed matrix using the BLOSUM-62 scoring matrix and a linear gap penalty, defined in Equation EQ5.13 with E set to –8. (See text for details of how this was done.) The red arrows indicate steps used in the traceback of the optimal alignment. (B) The optimal alignment returned by these calculations, which has a score of –4.

To fill in this matrix, one starts by aligning the beginning of each sequence; that is, in the extreme upper left-hand corner. We could equally well start at the end of the sequences (extreme bottom right-hand corner), but then the matrix should be labeled $1 \rightarrow m+1$ and $1 \rightarrow n+1$. The elements $S_{i,0}$ and $S_{0,j}$ are easy to fill in, because there is only one possible alignment available. $S_{i,0}$ represents the alignment

$$\begin{array}{cccccc} a_1 & a_2 & a_3 & \cdots & a_i \\ - & - & - & \cdots & - \end{array}$$

We will start by considering a linear gap penalty g of $-8n_{gap}$ for a gap of n_{gap} residues, giving the scores of $S_{i,0}$ and $S_{0,j}$ as $-8i$ and $-8j$, respectively. This starting point with numerical values inserted into the matrix is illustrated in Figure 5.8.

The other matrix elements are filled in according to simple rules that can be understood by considering a process of adding one position at a time to the alignment. There are only three options for any given position, namely, a pairing of residues from both sequences, and the two possibilities of a residue from one sequence aligning with a gap in the other. These three options can be written as:

$$\begin{array}{ccc} \cdots x_i & \cdots - & \cdots x_i \\ \cdots y_j & \cdots y_j & \cdots - \end{array}$$

The scores associated with these are $s(x_i,y_j)$, g, and g, respectively. The value of $s(x_i,y_j)$ is given by the element $s_{a,b}$ of the substitution score matrix, where a is the residue type of x_i and b is the residue type of y_j. The change in notation is solely to improve the clarity of the following equations.

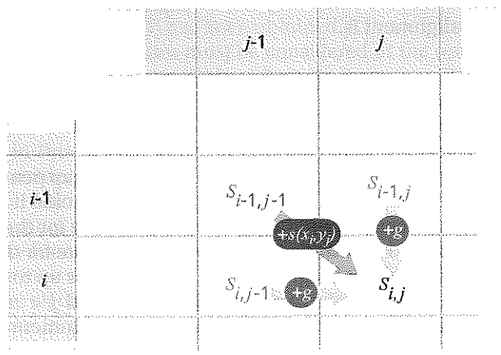


Figure 5.10
Illustration of the application of Equation EQ5.17 to calculate an element of the dynamic programming matrix. Only a small part of the matrix is shown, as only this part contributes directly to the calculation of the element.

Consider the evaluation of element $S_{1,1}$, so that the only residues that appear in the alignment are x_1 and y_1. The left-hand possibility of the three possibilities could only occur starting from $S_{0,0}$, as all other alignments will already contain at least one of these two residues. The middle possibility can only occur from $S_{1,0}$ because it requires an alignment that contains x_1 but not y_1. Similar reasoning shows that the right-hand possibility can only occur from $S_{0,1}$. The three possible alignments have the following scores:

$$\begin{array}{rcccccc} S_{0,0}+s(x_1,y_1) & = & 0+s(\mathrm{I,T}) & = & 0-1 & = & -1 \\ S_{1,0}+g & = & -8-8 & = & -16 & & \\ S_{0,1}+g & = & -8-8 & = & -16 & & \end{array} \qquad \text{(EQ5.16)}$$

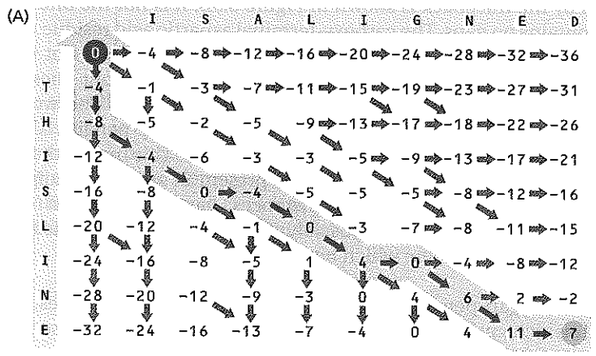
where s(I,T) has been obtained from Figure 5.5. Of these alternatives, the optimal one is clearly the first. Hence in Figure 5.9, $S_{1,1} = -1$. Because $S_{1,1}$ has been derived from $S_{0,0}$ an arrow has been drawn linking them in the figure.

An identical argument can be made to construct any element of the matrix from three others, using the formula

$$S_{i,j} = \max \begin{cases} S_{i-1,j-1} + s(x_i,y_1) \\ S_{i-1,j} + g \\ S_{i,j-1} + g \end{cases} \qquad \text{(EQ5.17)}$$

The maximum ("max") implies that we are using a similarity score. Figure 5.10 illustrates this formula in the layout of the matrix. Note that it is possible for more than one of the three alternatives to give the same optimal score, in which case arrows are drawn for all optimal alternatives. The completed matrix for the example sequences is given in Figure 5.9A. Note that the number of steps in this algorithm is proportional to the product of the sequence lengths.

We now have a matrix of scores for optimal alignments of many sub-sequences, together with the global sequence alignment score. This is given by the value of $S_{m,n}$, which in this case is $S_{8,9} = -4$. Note that this is not necessarily the highest score in the matrix, which in this case is $S_{8,8} = 4$, but only $S_{m,n}$ includes the information from both complete sequences. For each matrix element we know the element(s) from which it was directly derived. In the figures in this chapter, arrows are used to indicate this information.



(B) THIS-LI-NE-
|| || ||
--ISALIGNED

Figure 5.11
Optimal global alignment of two sequences, identical to Figure 5.9, except for a change in gap scoring. The linear gap penalty, defined in Equation EQ5.13 using a value of –4 for the parameter E. (A) The completed matrix using the BLOSUM-62 scoring matrix. (B) The optimal alignment, which has a score of 7.

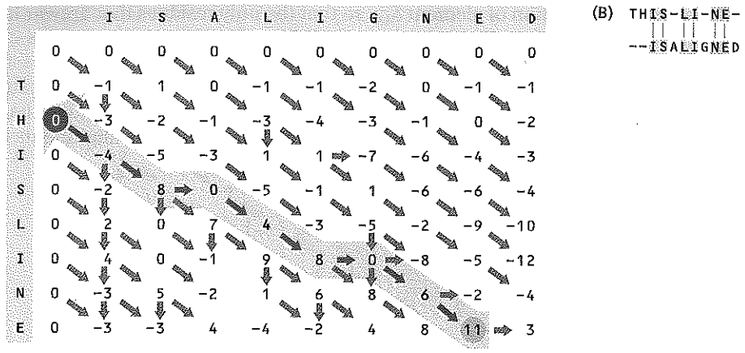


Figure 5.12
Dynamic programming matrix for semiglobal alignment of the same sequences as in Figure 5.9. In this case, end gaps are not penalized. (A) The completed matrix for determining the optimal global alignment of THISLINE and ISALIGNED using the BLOSUM-62 scoring matrix with a linear gap penalty, defined in Equation EQ5.13 with E set to –8 and with end gaps not penalized. (B) The optimal alignment, which has a score of 11.

We can use the information on the derivation of each element to obtain the actual global alignment that produced this optimal score by a process called **traceback**. Beginning at $S_{m,n}$ we follow the arrows back through the matrix to the start ($S_{0,0}$). Thus, having filled the matrix elements from the beginning of the sequences, we determine the alignment from the end of the sequences. At each step we can determine which of the three alternatives given in Equation EQ5.17 has been applied, and add it to our alignment. If $S_{i,j}$ has a diagonal arrow from $S_{i-1,j-1}$, that implies the alignment will contain x_i aligned with y_j. Vertical arrows imply a gap in sequence x aligning with a residue in sequence y, and vice versa for horizontal arrows. The traceback arrows involved in the optimal global alignment in Figure 5.9A are shown in red. When tracing back by hand, care must be taken, as it is easy to make mistakes, especially by applying the results to residues x_{i-1} and y_{j-1} instead of x_i and y_j.

The traceback information is often stored efficiently in computer programs, for example using three bits to represent the possible origins of each matrix element. If a bit is set to zero, that path was not used, with a value of one indicating the direction. Such schemes allow all this information to be easily stored and analyzed to obtain the alignment paths.

Note that there may be more than one optimal alignment if at some point along the path during traceback an element is encountered that was derived from more than one of the three possible alternatives. The algorithm does not distinguish between these possible alignments, although there may be reasons for preferring one to the others. Such preference would normally be justified by knowledge of the molecular structure or function. Most programs will arbitrarily report just one single alignment.

The alignment given by the traceback is shown in Figure 5.9B. It is not the one we expected, in that it contains no gaps. The carboxy-terminal aspartic acid residue (D) in sequence y is aligned with a gap only because the two sequences are not the same length. We can readily understand this outcome if we consider our chosen gap penalty of 8 in the light of the BLOSUM-62 substitution matrix. The worst substitution score in this matrix is –4, significantly less than the gap penalty. Also, many of the scores for aligning identical residues are only 4 or 5. This means that if we set such a high gap penalty, a gap is unlikely to be present in an optimal alignment using this scoring matrix. In these circumstances, gaps will occur if the sequences are of different length and also possibly in the presence of particular residues such as tryptophan or cysteine which have higher scores.

If instead we use a linear gap penalty $g(n_{gap}) = -4n_{gap}$, the situation changes, as shown in Figure 5.11, which gives the optimal alignment we expected. Because the

gap penalty is less severe, gaps are more likely to be introduced, resulting in a different alignment and a different score. In this particular case, four additional gaps occur, two of which occur within the sequences. The overall alignment score is 7, but this alignment would have scored –13 with the original gap penalty of 8.

This example illustrates the need to match the gap penalty to the substitution matrix used. However, care must be taken in matching these parameters, as the performance also depends on the properties of the sequences being aligned. Different parameters may be optimal when looking at long or short sequences, and depending on the expected sequence similarity (see Figure 4.5 for a practical example).

A simple modification of the algorithm allows the sequences to overlap each other at both ends of the alignment without penalty, often called **semiglobal alignment**. In this way, better global alignments can be obtained for sequences that are not the same length. Instead of applying the gap penalty scores to matrix elements $S_{i,0}$ and $S_{0,j}$, we set these to zero. The remaining elements are calculated as before (Equation EQ5.17). However, instead of traceback beginning at $S_{m,n}$, it starts at the highest-scoring element in the bottom row or right-most column. This is illustrated in Figure 5.12 for the gap penalty $g(n_{gap}) = -8n_{gap}$. Note that now the expected alignment is obtained, despite the gap penalty being so high. This modified algorithm gives the same optimal alignment with a gap penalty of $g(n_{gap}) = -4n_{gap}$.

The methods presented above are for use when scoring gaps with a linear penalty of the form $g(n_{gap}) = -n_{gap}E$. If we wish to differentiate between penalties for starting and extending a gap, using a scoring scheme such as $g(n_{gap}) = -I - (n_{gap} - 1)E$, a slightly different algorithm is required. The problem is not simply one of ensuring that we know if a gap is just being started or is being extended. In the previous algorithm, the decision for $S_{i,j}$ could be made without knowing the details of the alignment for any of $S_{i-1,j}$, $S_{i-1,j-1}$, or $S_{i,j-1}$. Now we need to know if these alignments ended with gaps, which means knowing details of their derivation, particularly involving $S_{i-2,j}$ and $S_{i,j-2}$.

Consider the general case of using a gap penalty, which is simply written as a function of the gap length n_{gap}; that is, $g(n_{gap})$. Now, for any matrix element $S_{i,j}$, one must consider the possibility of arriving at that element directly via insertion of a gap of length up to i in sequence $\boldsymbol{x}$ or j in sequence $\boldsymbol{y}$. Some of these routes are illustrated in Figure 5.13. The algorithm now has to be modified to

$$S_{i,j} = \max \begin{cases} S_{i-1,j-1} + s(x_i, y_j) \\ [S_{i-n_{gap1},j} + g(n_{gap1})]_{1 \le n_{gap1} \le i} \\ [S_{i,j-n_{gap2}} + g(n_{gap2})]_{1 \le n_{gap2} \le j} \end{cases} \quad \text{(EQ5.18)}$$

The algorithm now has a number of steps proportional to mn^2, where m and n are the sequence lengths with $n > m$. This is a significant increase in requirements over the original, because there are approximately n terms involving gaps used to evaluate each matrix element. However, when we use the specific affine gap penalty formula of Equation EQ5.14 it is possible to reformulate things and obtain the full matrix in mn steps again. Let us define $V_{i,j}$ as

$$V_{i,j} = \max \left\{ S_{i-n_{gap1},j} + g(n_{gap1}) \right\}_{1 \le n_{gap1} \le i} \quad \text{(EQ5.19)}$$

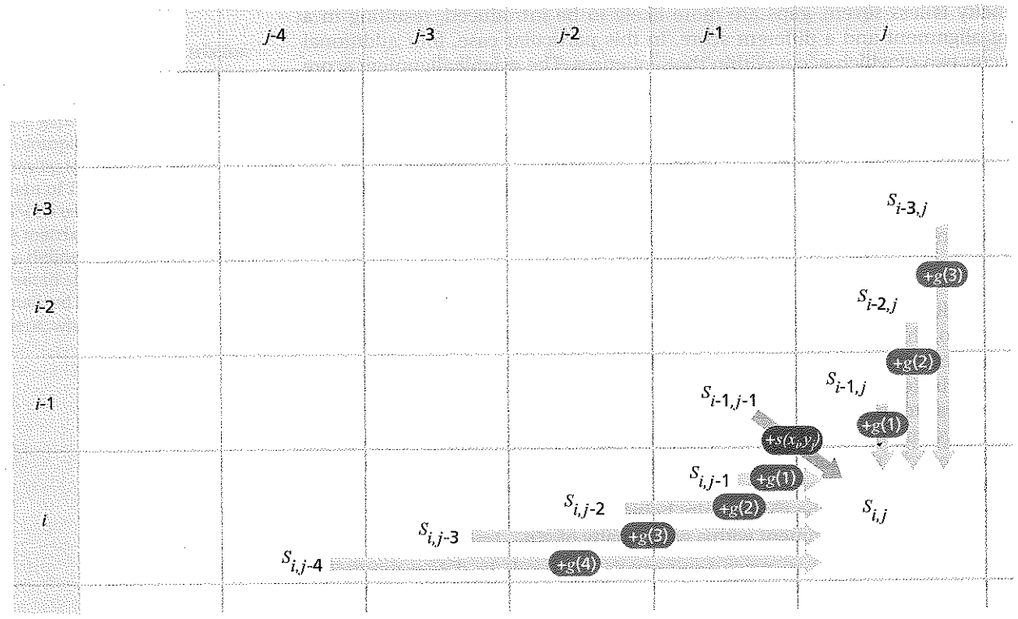


Figure 5.13
Illustration of some of the possible paths that could be involved in calculating the alignment score matrix for a general gap penalty $g(n_{gap})$. All possible gap sizes must be explored, with $S_{i,j}$ being chosen as the maximum of all these possibilities.

from which

$$V_{i,j} = \max\begin{cases} S_{i-1,j} + g(1) \\ [S_{i-n_{gap1},j} + g(n_{gap1})]_{2\le n_{gap1}\le i} \end{cases} \quad \text{(EQ5.20)}$$

$$= \max\begin{cases} S_{i-1,j} + g(1) \\ [S_{i-n_{gap1}-1,j} + g(n_{gap1}+1)]_{1\le n_{gap1}\le i-1} \end{cases} \quad \text{(EQ5.21)}$$

$$= \max\begin{cases} S_{i-1,j} - 1 \\ [S_{i-n_{gap1}-1,j} + g(n_{gap1}) - E]_{1\le n_{gap1}\le i-1} \end{cases} \quad \text{(EQ5.22)}$$

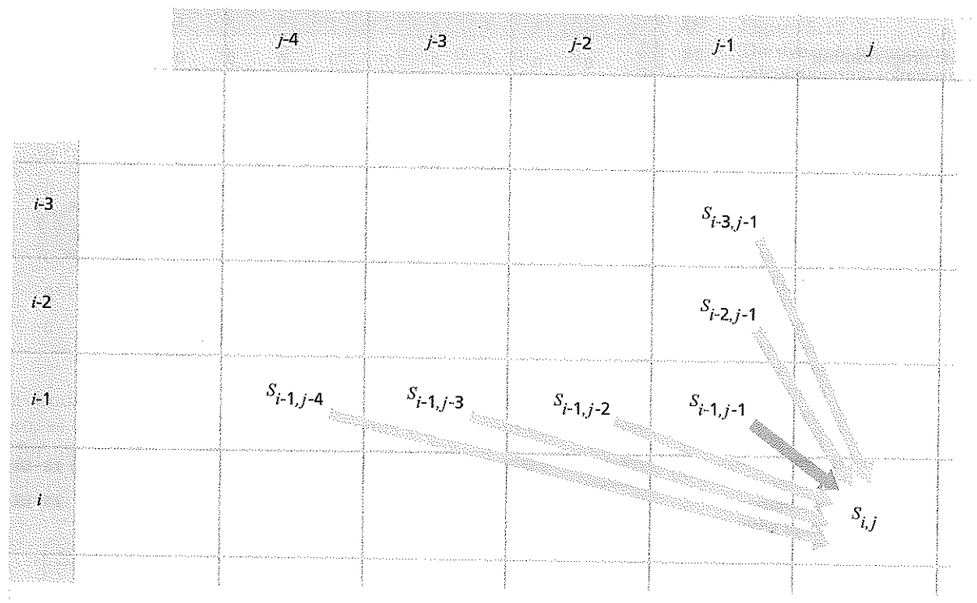
But from Equation EQ5.19 substituting $i-1$ for i, we can see that

$$V_{i-1,j} = \max\left\{S_{i-n_{gap1}-1,j} + g(n_{gap1})\right\}_{1\le n_{gap1}\le i-1} \quad \text{(EQ5.23)}$$

so that

$$V_{i,j} = \max\begin{cases} S_{i-1,j} - I \\ V_{i-1,j} - E \end{cases} \quad \text{(EQ5.24)}$$

Thus we have a recursive equation for the elements $V_{i,j}$, involving aligning residues in sequence x with gaps in y. This can readily be evaluated from a starting point of



$V_{1,j} = S_{0,j} - I$ (Equation EQ5.19 with $i = 1$). In a similar manner, we can define $W_{i,j}$, involving aligning residues in sequence y with gaps in x as

$$W_{i,j} = \max\begin{cases} S_{i,j-1} - I \\ W_{i,j-1} - E \end{cases} \quad \text{(EQ5.25)}$$

This can readily be evaluated from a starting point of $W_{i,1} = S_{i,0} - I$. These two recursive formulae can be substituted into Equation EQ5.18 to give the faster (nm steps) algorithm

$$S_{i,j} = \max\begin{cases} S_{i-1,j-1} + s(x_i, y_i) \\ V_{i,j} \\ W_{i,j} \end{cases} \quad \text{(EQ5.26)}$$

It should be noted that the original algorithm of Needleman and Wunsch differs in some key details from those described here. Their method is slower (requiring more computing steps) and is rarely used today. The types of path involved in calculating the matrix elements with their algorithm are shown in Figure 5.14. Note that the interpretation of the paths through the matrix differs from that presented above (see the figure legend for details).

Figure 5.14
Illustration of some of the possible paths that could be involved in calculating the alignment score matrix with the original Needleman and Wunsch algorithm. All possible gap sizes must be explored, with $S_{i,j}$ being chosen as the maximum of all these possibilities. The interpretation differs from the previous matrices in that if a path stops at an element $S_{i,j}$ it indicates that residues x_i and y_j are aligned with each other. Thus the path $S_{i-1,j-2} \rightarrow S_{i,j}$ represents the alignment of x_{i-1}–x_i with $y_{j-2}y_{j-1}y_j$; that is, an insertion in sequence x aligned with y_{j-1}.

Local and suboptimal alignments can be produced by making small modifications to the dynamic programming algorithm

Often we do not expect the whole of one sequence to align well with the other. For example, the proteins may have just one domain in common, in which case we

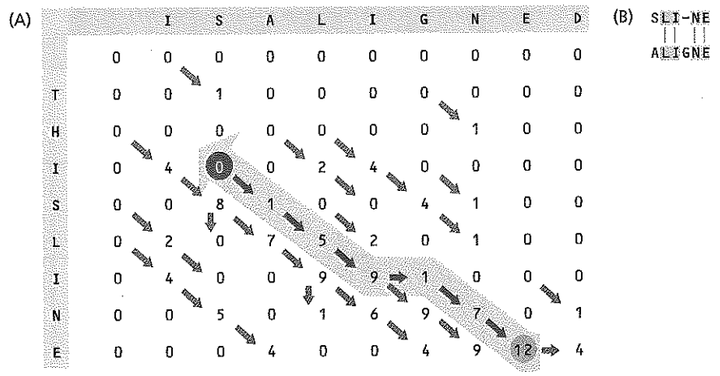


Figure 5.15
The dynamic programming calculation for determining the optimal local alignment of the two sequences THISLINE and ISALIGNED. (A) The completed matrix using the BLOSUM-62 scoring matrix with a linear gap penalty, defined in Equation EQ5.13 with E set to –8. (B) The optimal alignment, determined by the highest-scoring element, which has a score of 12.

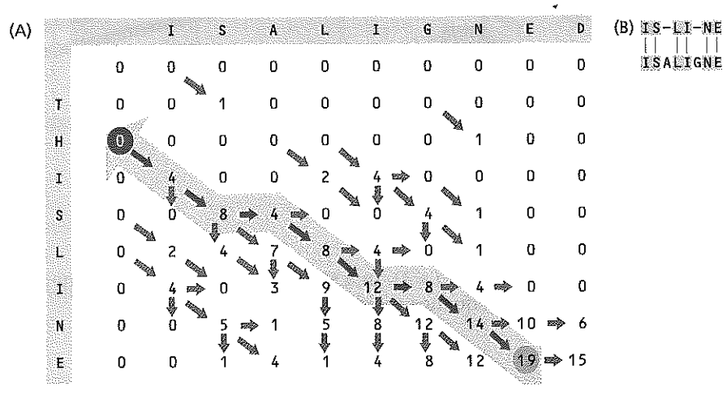


Figure 5.16
Optimal local alignment calculation identical to Figure 5.15, except with a linear gap penalty with E set to –4. (A) The completed matrix for determining the optimal local alignment of THISLINE and ISALIGNED using the BLOSUM-62 scoring matrix. (B) The optimal alignment, identified by the highest-scoring element in the entire matrix, which has a score of 19.

want to find this high-scoring zone, referred to as a local alignment (see Section 4.5). In a global alignment, those regions of the sequences that differ substantially will often obscure the good agreement over a limited stretch. The local alignment will identify these stretches while ignoring the weaker alignment scores elsewhere.

It turns out that a very similar dynamic programming algorithm to that described above for global alignments can obtain a local alignment. Smith and Waterman first proposed this method. However, it should be noted that the method presented here requires a similarity-scoring scheme that has an expected negative value for random alignments and positive value for highly similar sequences. Most of the commonly used substitution matrices fulfill this condition. Note that the global alignment schemes have no such restriction, and can have all substitution matrix scores positive. Under such a scheme, scores will grow steadily larger as the alignment gets larger, regardless of the degree of similarity, so that long random alignments will ultimately be indistinguishable by score alone from short significant ones.

The key difference in the local alignment algorithm from the global alignment algorithm set out above is that whenever the score of the optimal sub-sequence alignment is less than zero it is rejected, and that matrix element is set to zero. The scoring scheme must give a positive score for aligning (at least some) identical residues. We would expect to be able to find at least one such match in any alignment worth considering, so that we can be sure that there should be some positive alignment scores. Another algorithmic difference is that we now start traceback from the highest-scoring matrix element wherever it occurs.

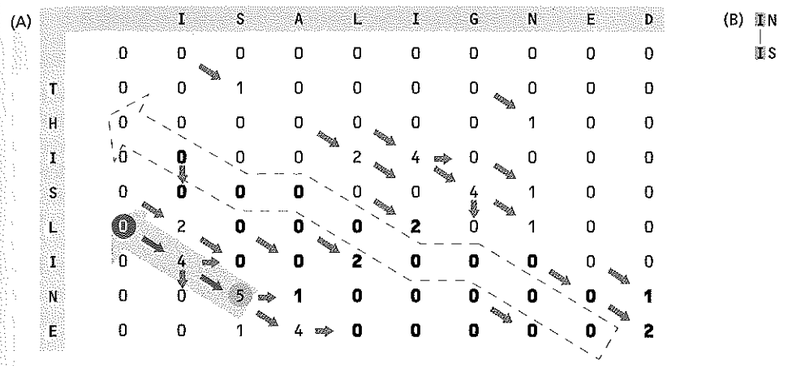


Figure 5.17
The dynamic programming calculation that follows on from the calculation shown in Figure 5.16 to find the best suboptimal local alignment. (A) The completed matrix for determining the best suboptimal local alignment of THISLINE and ISALIGNED using the BLOSUM-62 scoring matrix with a linear gap penalty with E set to –4. The matrix elements that were involved in the optimal local alignment have been set to zero and are shown with bold font. The matrix elements that have changed value from Figure 5.16(A) are also shown with bold font, and extend below and to the right of the optimal alignment. (B) The best suboptimal alignment, identified by the highest-scoring element in the entire matrix, has a score of 5.

The extra condition on the matrix elements means that the values of $S_{i,0}$ and $S_{0,j}$ are set to zero, as was the case for global alignments without end gap penalties. The formula for the general matrix element $S_{i,j}$ with a general gap penalty function $g(n_{\text{gap}})$ is

$$S_{i,j} = \max \begin{cases} S_{i-1,j-1} + s(x_i, y_i) \\ [S_{i-n_{\text{gap}1},j} + g(n_{\text{gap}1})]_{1 \le n_{\text{gap}1} \le i} \\ [S_{i,j-n_{\text{gap}2}} + g(n_{\text{gap}2})]_{1 \le n_{\text{gap}2} \le j} \\ 0 \end{cases} \qquad \text{(EQ5.27)}$$

which only differs from Equation EQ5.18 by the inclusion of the zero. The same modifications as above can be applied for the cases of linear gap penalty given in Equation EQ5.13 and affine gap penalty given in Equation EQ5.14.

Figures 5.15 and 5.16 show the optimal local alignments for our usual example in the two cases of linear gap penalties $g(n_{\text{gap}}) = -8n_{\text{gap}}$ and $-4n_{\text{gap}}$, respectively. Both result in removal of the differing ends of the sequences. In the first case, the higher gap penalty forces an alignment of serine (S) and alanine (A) in preference to adding a gap to reach the identical IS sub-sequence. Lowering the gap penalty in this instance improves the result to give the local alignment we would expect.

Sometimes it is of interest to find other high-scoring local alignments. A common instance would be the presence of repeats in a sequence. There will usually be a number of alternative local alignments in the vicinity of the optimal one, with only slightly lower scores. These will have a high degree of overlap with the optimal alignment, however, and contain little, if any, extra information beyond that given by the optimal local alignment. Of more interest are those suboptimal local alignments that are quite distinct from the optimal one. Usually their distinctness is defined as not sharing any aligned residue pairs.

An efficient method has been proposed for finding distinct suboptimal local alignments. These are alignments in which no aligned pair of residues is also found aligned in the optimal or other suboptimal alignments. They can be very useful in a variety of situations such as aligning multidomain proteins. Sometimes a pair of proteins has two or more domains in common but other regions with no similarity. In such cases it is useful to obtain separate local alignments for each domain, but only one of these will give the optimal score, the others being suboptimal alignments. The method starts as before by calculating the optimal local alignment. Then, to ensure that any new alignment found does not share any aligned residues

Box 5.1 Saving space by throwing away the intermediate calculations

In this chapter the steps required to identify optimal sequence alignments have been specified, but there has been no discussion of the precise coding used in a program. Such details are in general beyond the scope of this book, although important for the production of efficient and practical tools. The computer science and other specialist texts cited in Further Reading at the end of the chapter should be consulted for details of efficient coding techniques. However, we will briefly examine one algorithmic trick that can substantially increase the range of alignment problems feasible with limited computer resources.

All the methods presented so far calculate the whole of matrix ***S***, and by default it might be assumed that the entire matrix was stored for traceback analysis. If the sequences to be compared have lengths of up to a few thousand residues, the matrix can be stored quite easily on current computers. However, particularly in the case of nucleotide sequences, we might wish to align much longer sequences, for example mitochondrial DNA and even whole genomes. Storage of the whole matrix ***S*** is often not possible for such long sequences, sometimes requiring gigabytes of memory. Without using an alternative, this memory problem could prevent the use of these dynamic programming methods on such data. However, a neat solution is available by modifying the algorithm to store just two rows of the matrix. Note that further calculation is then needed to recover the actual alignment, as the basic method presented here only provides the optimal alignment score.

The key to this algorithm is to notice that the calculation of any element only requires knowledge of the results for the current and previous row (see Figure 5.10 and Equations EQ5.17 and EQ5.26); we could have chosen to work with columns instead. The two rows will be labeled $R_{0,j}$ and $R_{1,j}$. The scoring scheme used here will involve the linear gap penalty of Equation EQ5.13, but the affine gap penalty scheme can also be used. Similarly, the different variations in the treatment of end gaps can be incorporated.

The steps can be summarized as follows. Firstly, initialize row 0 according to the specific algorithm. For standard Needleman–Wunsch with a linear gap penalty, this means setting $R_{0,j} = -jE$ for $j = 0 \rightarrow$ m. For the 0th column, $R_{1,0}$ is then set to $-E$ or whatever other boundary conditions are required. This is equivalent to the initialization of the full matrix method above. We now step through each residue x_i of sequence x, from x_1 to x_n, calculating all the elements of row $R_{1,j}$, which at each step is the equivalent of the ith row in the full matrix. These elements are labeled with the letter j, and correspond to residue y_j of sequence y. Thus the elements of the $R_{0,j}$ row correspond to the matrix elements $S_{i-1,j}$, and those of the $R_{1,j}$ row to $S_{i,j}$. The other elements of $R_{1,j}$ are assigned a value according to the equation (equivalent to Equation EQ15.17)

$$R_{1,j} = \max \begin{cases} R_{0,j-1} + s(x_i, y_j) \\ R_{0,j} - E \\ R_{1,j-1} - E \end{cases} \qquad \text{(BEQ5.1)}$$

Once all the elements of the $R_{1,j}$ row have been calculated, the value of each matrix element of $R_{1,j}$ is transferred to $R_{0,j}$, which now represents matrix row 1. In practical programming, pointers would be used, so that this step would take virtually no time. For the 0th column, $R_{1,0}$ is then set to $-2E$ or whatever other boundary conditions are required, with the $R_{1,j}$ elements now representing matrix row 2. For the other columns, $R_{1,j}$ is assigned a value as before. In this way results keep being overwritten, but sufficient are kept to continue the calculation.

Certain scores must be saved, the details differing according to the precise type of alignment sought. Thus for the basic Needleman–Wunsch scheme, only $R_{m,n}$ need be stored, while Smith–Waterman requires the highest-scoring element and the associated values of i and j. The storage requirements are twice the length of the shorter sequence. Because this technique only stores two rows of the matrix, it makes it possible to use dynamic programming to align complete bacterial genomes.

However, the saving in memory required comes at a price, and the traceback procedure is much more complicated than that which can be used if the full matrix calculation has been stored. The traceback now involves considerably more calculation and so production of the overall alignment will require longer computing times. Versions of this technique exist for all the types of alignment mentioned in this chapter. Over the past few years much effort has been devoted to optimizing alignment programs for use with whole genome sequences. See Section 5.5 for more practical methods concerning whole genome sequence alignment. In contrast to the methods above, the methods discussed in that section all involve approximations, but a good argument can be made that such techniques are more appropriate for that type of problem.

with the optimal alignment, it is necessary to set all the matrix elements on this initial path to zero (see Figure 5.17). Subsequently, the matrix needs to be recalculated to include the effect of these zeroed elements. If we recalculate the matrix, only a relatively small number of elements near the optimal local alignment will have new values. The key to the efficiency of the technique is the recognition that these modified elements can only affect elements to their right and below them (see Figure 5.13). Working along a matrix row from the zeroed elements, the new values are monitored until an element is found whose value is unchanged from the original calculation. Further elements on this row will also be unchanged, so calculation can now move to the next row down. In this way the new matrix is obtained with a minimum of work. The suboptimal local alignment is identified by locating the highest matrix element. Further suboptimal alignments can be found by repeating the procedure, zeroing every element involved in a previous alignment. For affine gap penalties the alignment elements in the matrices ***V*** and ***W*** are also forced to zero and must also be unchanged before a row calculation can stop.

In the example shown in Figure 5.17, the first suboptimal local alignment is found for the linear gap penalty $g(n_{gap}) = -4n_{gap}$; that is, following on from Figure 5.16. All the matrix elements of the optimal alignment are in bold font, as are those elements that have been recalculated. Only 27 elements needed to be recalculated, all below or to the left of the elements of the optimal alignment. If we had not monitored for unchanged elements we would have had to recalculate 54 elements.

Time can be saved with a loss of rigor by not calculating the whole matrix

High-scoring local alignments indicating significant sequence similarity usually do not have many insertions and deletions. Global alignments may contain large insertions, for example if there is a whole domain inserted in one sequence, but such situations are only rarely encountered. In terms of the matrix, this means that these alignments generally follow a diagonal quite closely. This has led people to try to save time in deriving alignments by only calculating matrix elements within a

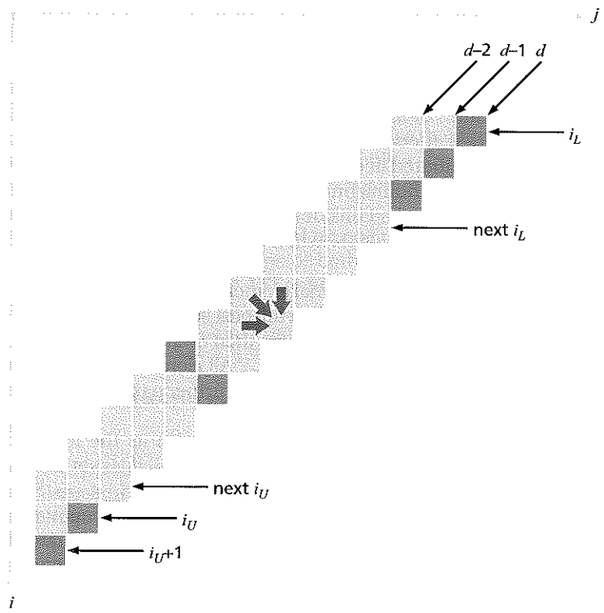


Figure 5.18
The *X*-drop method, proceeding by antidiagonals indexed by $d = i + j$. The boxes shaded in blue are matrix elements which have a score that falls X below the current best score, and therefore have been assigned a value of $-\infty$. The normal three possible paths used to determine the score are shown in one case by red arrows. (Adapted from Z. Zhang et al., A greedy algorithm for aligning DNA sequences, *J. Comput. Biol.* 7 (1–2):203–214, 2000.)

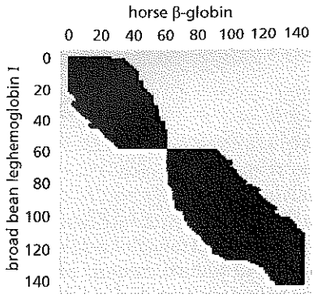


Figure 5.19
The matrix elements calculated in the BLAST program using the *X*-drop method for the example of aligning broad bean leghemoglobin I and horse β-globin. The value of X used was 40, with BLOSUM-62 substitution matrix and an affine gap penalty of $g(n_{gap}) = -10 - n_{gap}$. The alignment starts at $S_{60,62}$, alanine in both sequences, which is the location determined by the process illustrated in Figure 5.24. Calculated elements are shown black. (From S.F. Altschul et al., Gapped BLAST and PSI-BLAST: a new generation of protein database search programs, *N.A.R.* 25 (17):3389–3402, 1997, by permission of Oxford University Press.)

short range of a specified diagonal. Note that such techniques are not guaranteed to find the optimal alignment!

There are two ways of assigning the diagonal around which the alignment will be sought. One can assume that both sequences are similar at a global level, and use the diagonal $S_{i,i}$ of the matrix. Alternatively, especially in the case of database searches, one can use the diagonal of a high-scoring ungapped local alignment found in an initial alignment search, as discussed later.

There are two ways of restricting the coverage of the matrix away from the central diagonal. If a limit is imposed for the maximum difference M_{ins} between the number of insertions in each of the two sequences, the algorithm can simply be set to include only M_{ins} diagonals on either side of the central one; that is $(2M_{\text{ins}} + 1)$ diagonals in total. The database search program FASTA (see Section 4.6 and below) uses this technique, especially for nucleotide sequences, when M_{ins} is frequently set to 15. A second method of restricting the matrix elements examined is to limit the search according to how far the scores fall below the current best score. In this ***X*-drop method**, used in the database searching program BLAST (see Section 4.6 and below), the current best score is monitored. When calculating elements along a row, calculation stops when an element scores a preset value X below this best score, and calculation restarts on the next row. A large value of X results in more of the matrix being evaluated, in which case the true optimal alignment is more likely to be determined.

The X-drop algorithm that follows aligns two sequences $\boldsymbol{x}$ and $\boldsymbol{y}$ of length m and n, respectively. The matrix elements are processed for each antidiagonal in turn. The dth antidiagonal is defined by $d = i + j$. Only a restricted region of the dth antidiagonal is evaluated, defined by variables i_L and i_U, as illustrated in Figure 5.18. The current best alignment score is stored in the variable S_{best}.

In some cases, such as occur in the database search methods described in Section 5.3, the algorithm is started at element S_{i_0,j_0}, which has been identified by a preliminary local alignment step. In this instance S_{best} is initially set to the value of element S_{i_0,j_0}, and the other variables are initialized to $d = i_0 + j_0$ and $i_L = i_U = i_0$. Alternatively, the algorithm can be started at element $S_{0,0}$, which has the value zero, so that S_{best} is initially set to zero, as are d, i_L, and i_U. With these initial values, the algorithm proceeds as follows.

Each antidiagonal is evaluated in turn. We will describe the method as proceeding from smaller to larger values of d. The changes required to proceed to decreasing values of d will be discussed afterwards. If there are elements of the antidiagonal to calculate, residues x_i are evaluated in order of increasing i, from i_L to $i_U + 1$. For each value of i the relevant j for this antidiagonal can be obtained using $j = d - i$. This identifies the matrix element $S_{i,d-i}$ under consideration. Initial evaluation is as described previously, for example, Equation EQ5.17 when a linear gap penalty is used.

After this initial evaluation of the antidiagonal elements, a check is first made to see if the score of any of these elements improves on the best score so far, in which case S_{best} is set to this value. The antidiagonal elements are then evaluated to identify any that fall more than X below this current best score, S_{best}. All elements that have such low scores are assigned the value $-\infty$, so that they play no further part in later calculations.

In the next step, i_L and i_U are redetermined. i_L is chosen as the lowest i in the selected region of the current antidiagonal such that $S_{i,j} \neq -\infty$. i_U is set to the highest i in the selected region of the current antidiagonal such that $S_{i,j} \neq -\infty$. Sometimes this will result in a smaller region being defined for the next antidiagonal, the situation illustrated in Figure 5.18. When the elements at the edge of the antidiagonal region do not have the value $-\infty$ the region must be extended. This extension might be based on the calculation of scores for further elements of the antidiagonal, or

may be limited to a prespecified number of extra elements. If $i_U + 1 \leq i_L$ there are no matrix elements in the selected region, and the calculation is finished. Otherwise, the next antidiagonal, that is, the $(d + 1)$th, is now evaluated.

If the calculation started at an element S_{i_0,j_0}, that is not $S_{0,0}$, it must be carried out in reverse as well to trace the alignment toward the start of the sequences. Note that there is no difference in principle between using dynamic programming to find optimal alignments forward or backward. Some indices need to be changed, however, as for example, element $S_{i,j}$ now depends on the values of $S_{i+1,j+1}$, $S_{i+1,j}$, and $S_{i,j+1}$. The score of such an element $S_{i,j}$ now relates to an alignment starting with residues x_i and y_j and ending at x_u and y_v. The full alignment is found by two tracebacks, one from the forward and one from the backward region of the matrix, both of which end at S_{i_0,j_0}. Figure 5.19 shows an example of the matrix elements calculated for a real alignment by an algorithm like this.

5.3 Indexing Techniques and Algorithmic Approximations

The huge increase in the size of the sequence databases and their daily updating has obliged the general research community to access these databases through central facilities. This makes it easier for people to be confident of searching all known data, but serves to concentrate the demands for similarity searches on a few machines. Even though the power of computers has greatly increased over the years, the methods of full-matrix dynamic programming described above are too demanding for general use in database searches.

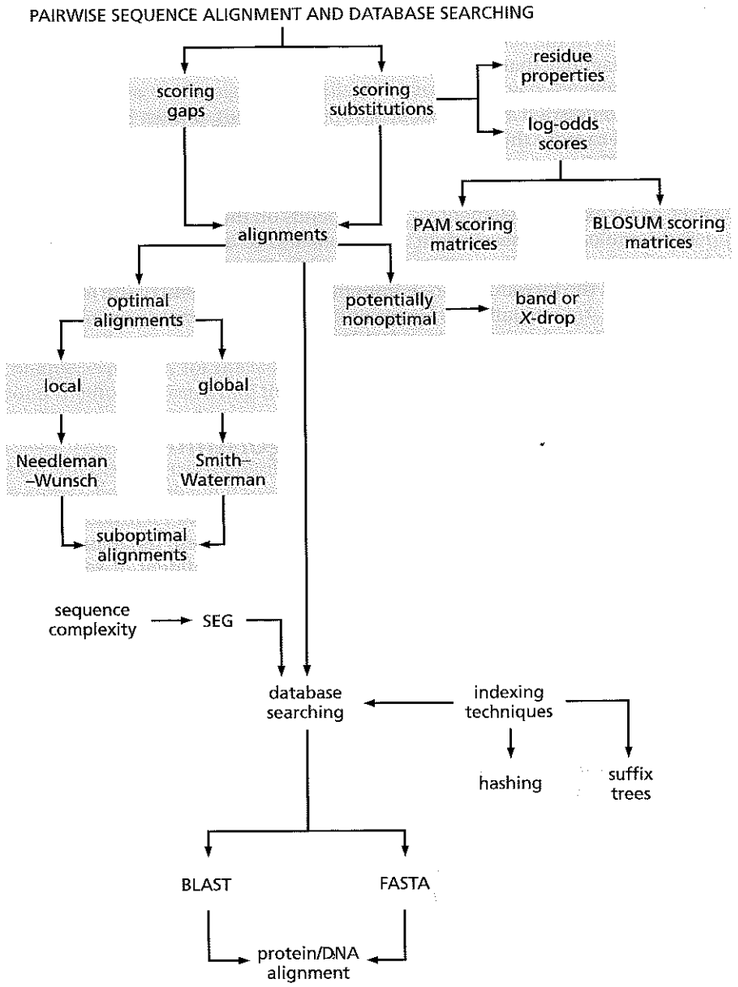
A number of alternative procedures have been developed that are considerably faster, although there is a penalty to pay in that they do not guarantee to find the highest-scoring alignment. The key to their success is the use of **indexing techniques** to locate possible high-scoring short local alignments. These initial local alignments are then extended, subject to certain constraints, to provide scores that are used to rank the database sequences by similarity. In most implementations, modified dynamic programming algorithms are used to examine the best-scoring sequences and to produce final scores and alignments.

In the following sections we will examine in detail the two major methods implemented in the two program suites in widest use today: BLAST and FASTA. These both start by considering very short segments of sequence that they call "words," "k-tuples," or "k-mers." A k-tuple (as used in FASTA) is simply a stretch of k residues in the query sequence. A k-mer (as used in BLAST) is a stretch of k residues, which when aligned with all k residue stretches of the query sequence will score above some threshold value (T) at least once. The term "word" is used more generally, meaning simply any short sub-sequence. The initial steps of both methods find high-scoring ungapped local alignments, referred to in BLAST as **high-scoring segment pairs** (**HSPs**), of which the highest-scoring one for a given pair of sequences is the **maximal segment pair** (**MSP**). Flow Diagram 5.3 gives an outline of the steps covered in this section.

Suffix trees locate the positions of repeats and unique sequences

One method of indexing uses a device known as the **suffix tree.** A variant of this technique is used in the BLAST programs. The example given here will be for a nucleotide sequence because an example using a realistic protein sequence would be too complex to illustrate the method. Consider a short segment of a nucleotide sequence ATCCGAGGATATCGA$, where the $ is used to identify the end of the sequence. This has a number of short repeats, such as AT. A suffix tree is a way of

Flow Diagram 5.3
The key concept introduced in this section is that database sequence similarity searches require refinements in the pairwise alignment methods to make them more efficient, although at the cost of decreasing the chance of finding the best-scoring alignment.



representing the sequence that uses these repeats to reduce the space needed for a full description. In addition, the form of the tree makes it very easy to find specific sequences and sequence repeats.

A **suffix** is defined as the shortest sub-sequence starting at a particular position that is unique in the complete sequence and can therefore be used to clearly identify that position. For example, the third base in the sequence above is C. As there are several Cs in the whole sequence, the sub-sequence C is not sufficiently unique to label position 3. However, by including the next base, as in CC, we have found a unique sub-sequence, and the suffix at position 3 is CC.

The suffix tree for the example sequence is given in Figure 5.20. Looking at it you can easily see that the longest repeats are the triplets ATC and CGA, which occur at positions (1,11) and (4,13), respectively. The efficiency of this technique may not be so apparent with this example because the sequence is rather short. Although longer sequences will tend to have longer suffixes, they will also tend to have more repeats, resulting in a more efficient tree.

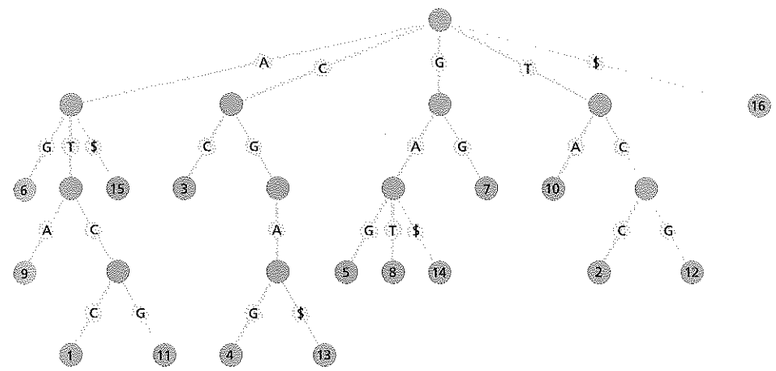


Figure 5.20
The suffix tree for the sequence ATCCGAGGATATCGA$. There are alternative ways of showing and labeling suffix trees, and often they are labeled with the whole suffix, but this form has been used because parallels can then be seen readily with finite-state automata (Figure 5.23). The numbered terminal nodes (leaves) correspond with the numbered positions in the sequence. Thus 6 refers to the sixth position, at which the base is A. The suffix for position 6 is AG. All suffix positions are clustered according to the 5′-terminal part of their sequence (or amino terminal if dealing with a protein sequence), grouping together all suffixes starting with a G for example.

Constructing the tree is straightforward. First, all positions in the sequence are grouped according to their base type; for a protein sequence this would be amino acid residue type. These groups correspond to the first row of nodes from the root. Each of these groups is then regrouped according to the following base to give the second row of nodes. This procedure is continued, stopping for a group when it only contains one sequence position. For a sequence of length L, this method requires a number of steps proportional to L log L. Faster methods, requiring a number of steps proportional to L, are known but are more involved.

Hashing is an indexing technique that lists the starting positions of all k-tuples

The basic aim of hashing is to construct a list of the starting positions of all k-tuples that occur in a query sequence. If we subsequently want to find where a particular k-tuple occurs in the sequence we simply look up the list. This procedure is used in FASTA.

Before construction of the list can begin we need to create a code for each k-tuple. Suppose we are dealing with nucleotide sequences, and k-tuples of length $k = 3$. Because there are four possible bases, there are 4^3 (= 64) possible k-tuples, trinucleotides in this case. We can easily create a number code for these. First, assign each base a number from 0 to 3, for example A = 0, C = 1, G = 2, and T = 3. If we label this variable e, any 3-tuple $x_i x_{i+1} x_{i+2}$ can be assigned a number (c_i) according to the formula

$$c_i = e(x_i)4^2 + e(x_{i+1})4^1 + e(x_{i+2})4^0 \qquad \text{(EQ5.28)}$$

For example, the trinucleotides AAA, CAA, ACA, and AAC would be assigned the numbers 0, 16, 4, and 1, respectively. The c_i will vary from 0 to 63 (= $4^3 - 1$). As an example, the sequence TAAAACTCTAAC has 10 trinucleotides with c_1 to c_{10} given by 48, 0, 0, 1, 7, 31, 55, 28, 48, and 1, respectively. In the case of protein sequences, the amino acids would be numbered from 0 to 19, and instead of using powers of 4, powers of 20 would be used.

If the sequence in question is of length L, there will be $(L - k + 1)$ k-tuples that will be coded into the values of c_1 to c_{n-k+1}. Because we ultimately want to search for particular k-tuples, which means finding particular values of the c_i, we need to sort the c_i into numerical order. For the example given above, the ordering will be c_2, c_3, c_4, c_{10}, c_5, c_8, c_6, c_1, c_9, c_7. There are many textbooks on numerical algorithms that give details of efficient sorting methods, so they will not be discussed here (see Further Reading).

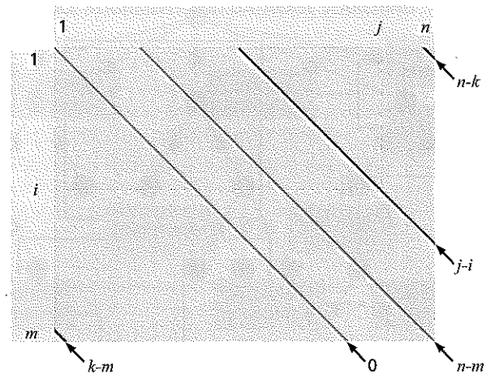


Figure 5.21
Definition of the labeling of matrix diagonals d_{j-i}. A word length of k is used, so the last $(k-1)$ elements of each row and column are not filled. This is why the diagonal labels range from $k-m$ to $n-k$.

For each different k-tuple we need to know whether it occurs, and if so at which base(s) it starts. There are several ways of doing this, of which a particularly efficient one is **chaining**. For sequence of length L, two arrays are required: one, $\boldsymbol{a}$, as long as the number of different possible k-tuples (4^k for nucleotides) and one, $\boldsymbol{b}$, of length $(L-k+1)$. The element a_i contains a pointer to the first base of the first occurrence of the ith k-tuple if it exists in the sequence, or else a value that signifies its absence. Suppose that this pointer is to base x_j. In that case, b_j contains a pointer to x_k, the first base of the second instance of the ith k-tuple, or else a value that signifies there is no second instance. If there is an x_k in b_j, then b_k will contain information about the presence of a further instance of the ith k-tuple.

The first eight elements of $\boldsymbol{a}$ for the example sequence TAAAACTCTAAC will be (2, 4, 0, 0, 0, 0, 0, 5), where a value 0 indicates the absence of that particular 3-tuple. The elements of $\boldsymbol{b}$ are (9, 3, 0, 10, 0, 0, 0, 0, 0, 0). This contains three pointers because three 3-tuples are present more than once in the sequence. It can be seen that no 3-tuple occurs more than twice, because these pointers in $\boldsymbol{b}$ are to elements that contain 0. Hashing and chaining techniques are used to great effect in the FASTA programs, as we discuss next.

The FASTA algorithm uses hashing and chaining for fast database searching

A series of programs have been written by William Pearson and colleagues that allow fast and accurate searching of both protein and nucleic acid databases with both protein and nucleotide query sequences. They are based on a very fast heuristic algorithm that has four distinct steps, which are applied to each database sequence independently. In the first step, local ungapped alignments of k-tuples are located. These are then scored using a standard scoring scheme, and only the highest-scoring aligned regions are retained. Still retaining the ungapped regions, an attempt is then made to join these into a single crude alignment for the pair of sequences, resulting in an initial alignment score. This score is used to rank the database sequences. In the final step, the highest-scoring sequences are aligned using dynamic programming. Depending on the program parameters selected, this may use only a band of the full matrix including the region containing the crude alignment. We will now describe some of these steps in more detail.

FASTA starts by hashing and chaining the query sequence. A parameter, ktup, gives the size of the k-tuples to be hashed, and is usually set to 2 amino acids for proteins and 6 bases for nucleotides. Note that these values produce 400 and 4096 different possible k-tuples, respectively. As the program will search for k-tuples shared by both sequences, using smaller values of the ktup parameter makes the search more

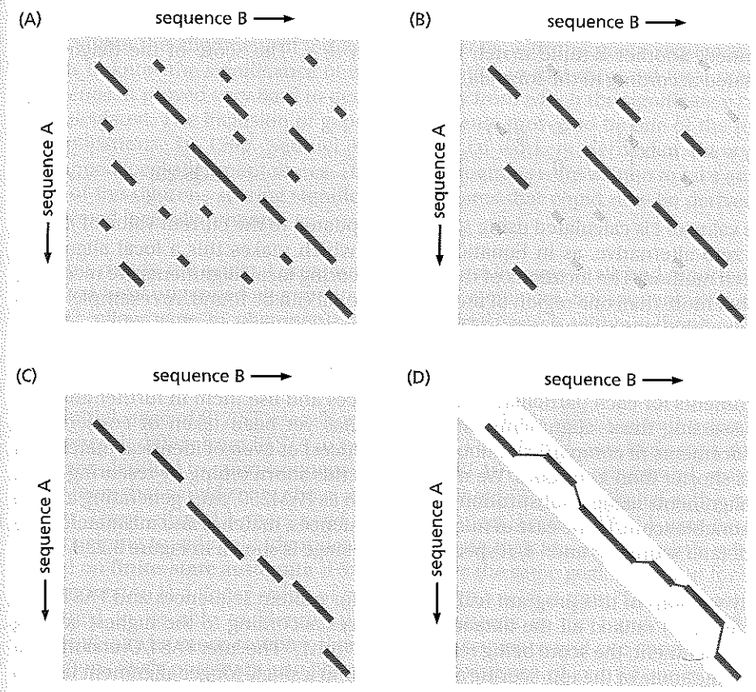


Figure 5.22
The four steps in the FASTA method for database searching. (A) For a given database sequence, all ungapped local alignments (against the query sequence) of suitably high score are shown. (B) The ten highest-scoring alignments are rescored with the PAM250 matrix, the highest having score init1. (C) Attempts are made to join together some of these ungapped alignments, allowing some gaps, to obtain score initn. (D) Dynamic programming is used to extend the alignment and give the final score opt.

sensitive. However, this is usually only beneficial for distantly related sequences, and searches based on family profiles (see Sections 4.6 and 6.1) would be expected to be even more sensitive.

In what follows, we will assume that the chaining arrays a and b have been calculated as described in the previous section. For each database entry, successive k-tuples of the sequence are searched against these arrays to identify where identical k-tuples align in the two sequences. As these alignments are ungapped, they lie on diagonals in an alignment matrix (such as the one in Figure 5.11). We want to find those diagonals that have many such k-tuples aligned along them.

Before giving the details of the search algorithm, a few comments about labeling diagonals in an alignment matrix will be helpful. As in the alignment matrices shown in Section 5.2, the alignment diagonals descend to the right. Suppose the two sequences have m and n residues, respectively. Remembering that there are $(m-k)$ k-tuples in an m residue sequence, there will be $(n+m-2k+1)$ diagonals, which can be labeled d_{k-m} to d_{n-k} (see Figure 5.21).

With this diagonal labeling, if the same k-tuple is located at position i of the query sequence and at position j of the database sequence, the alignment of these k-tuples will lie on the diagonal d_{j-i}. The chaining arrays make it simple to determine all the identical k-tuples, and on which diagonals they lie. All such aligned k-tuples are given the same (positive) score s. We just need to account for residues between these aligned k-tuples to complete the scoring of diagonals. These can be scored with a penalty $g(l)$ proportional to their length l. Using these scores, a straightforward algorithm finds the highest-scoring local regions of the diagonals. This is related to the Smith–Waterman algorithm, although as there are only matching residues it is much simpler (and quicker). A score is maintained for each diagonal

of the alignment matrix. Let this score be s_{j-i} for diagonal d_{j-i}, and suppose the last k-tuple match on this diagonal was at location j_0. As the database sequence is scanned, another k-tuple match is found on d_{j-i}. The score of the diagonal is updated according to the formula

$$s_{j-i} = s + \max \begin{cases} s_{j-i} & + \quad g(l) \\ & 0 \end{cases} \qquad \text{(EQ5.29)}$$

The value of l is calculated using j_0, and j_0 is updated to the current value of j. Note the zero alternative, as in Equation EQ5.27, which makes this a local alignment search method. The locations of the highest-scoring local alignments are recorded, so that more than one region of the same diagonal can be found. An example of the result of this step can be seen in Figure 5.22A.

Current implementations of this scheme identify the 10 highest-scoring ungapped alignments for each particular pair of sequences and use them in further analysis. In assessing these alignments, no account has yet been taken of conservative replacements (if comparing amino acid sequences) or even of identical matches in runs shorter than k residues. We now address this shortcoming by rescoring these 10 alignments using a substitution matrix such as PAM250, thus generating a more reasonable score. In the case of nucleotide sequences, matches and mismatches are by default scored +5 and –4, respectively. This stage is shown in Figure 5.22B.

Early versions of this program (called FASTP for protein sequences and FASTN for nucleotides) ranked all the database sequences according to the highest-scoring local alignment, the score being referred to as "init1." The later FASTA versions first combine some of the top-scoring alignments into a single longer alignment using a simple dynamic programming technique. The scoring scheme for this technique is based on the scores of the individual alignments and a joining penalty to score the regions between them. The resultant alignment score is called "initn," and is used to make a preliminary ranking of the database sequences. This stage is shown in Figure 5.22C.

In the last step, suitably high-scoring database sequences are further investigated with a Smith–Waterman local alignment procedure to produce the final alignment and score. This will introduce gaps to give the best alignment. In most cases the older versions of the program used a banded version of the algorithm, centered on the initial approximate alignment. In general, a band of 16-residue width was used, except in the case of protein sequence alignments with 1-tuple indexing, when a 32-residue band was employed. This stage is shown in Figure 5.22D. In later versions, such as FASTA3, all alignments of protein sequences use the full-matrix Smith–Waterman method by default. The default for nucleotide sequences is still to use the banded version, as the full-matrix method is very time consuming for the longer sequences.

The resultant alignment score, opt, is used for the final sequence ranking, but the ranking itself is according to the estimated significance of the score. This is based on theories such as the extreme-value distribution discussed in the final part of the chapter. The significance estimates involve the sequence lengths as well as the score, and thus the ranking reported can differ from that based directly on opt. By default, the alignments reported by FASTA are those given by the Smith–Waterman method, and therefore may contain gaps.

By restricting the initial search to ungapped perfect matches, joining these together in a simple way, and using the resulting score to filter out very dissimilar database sequences, FASTA can achieve a considerable speed-up in database searches over a straightforward application of the Smith–Waterman method. Furthermore this has been achieved for a very small loss of sensitivity.

The BLAST algorithm makes use of finite-state automata

To evaluate the alignment of a database sequence with the query sequence, one needs to know the significance of its score relative to that expected for a random sequence. It proved very hard to derive a theory from which the significance could be calculated. The inclusion of gaps in alignments proved to be one of the major complications. In 1990, Samuel Karlin and co-workers derived a theory for ungapped local alignment scores. The BLAST programs were written to take advantage of the rigorous scoring significance estimates that could now be derived for ungapped local alignments.

For this reason the original program did not consider gaps at all, and reported one or more local ungapped alignments per pair of sequences. Subsequent versions of BLAST allow gaps in alignments after an initial search without them, so that currently BLAST will produce a final local gapped alignment. In this respect the latest versions of BLAST are similar to FASTA.

The initial stage of BLAST uses short words to search for identities in the database sequence. It differs from FASTA in two respects. First, FASTA only looks for k-tuples that are identical to query-sequence k-tuples. BLAST, on the other hand, searches for k-mers that would score above a given threshold (T) when aligned with a query k-mer. These aligned k-mers need not be identical. Second, FASTA uses hashing and chaining to aid rapid identification of k-tuple matches. BLAST uses a scheme based on **finite-state automata** (**FSA**) to achieve the same goal. The input for such an automaton is a linear string of symbols taken one at a time. The automaton is designed to be able to identify particular patterns in the input string. These patterns may be more complex than a specific sequence and can cover a range of variation. Such automata usually report the existence of one or more of these patterns, possibly including location information, by emitting data under specific circumstances.

Each state of an automaton has well-defined responses to any possible input, responses that can include both the transition to a new state and the emission of symbols. A key response to a new symbol is not to accept it, meaning that the input string is rejected. Some automata always start reading a new string from a particular state, in which case rejection can also be written as transition back to this state. Figure 5.23 shows an example of such a finite-state automaton, in which rejection (denoted by ¶$_1$ and ¶$_2$) results in transition back to state 0.

As **hidden Markov models** (**HMMs**) are discussed in detail in further chapters (especially Sections 6.2 and 9.3) it is useful to note here that in contrast to HMMs, the transitions and emissions in FSA models are not probabilistic. Each input symbol leads to a deterministic outcome.

Unlike FASTA, at all stages BLAST calculates scores using a realistic substitution matrix such as BLOSUM-62 (that is, it does not use a simple sum of identities such

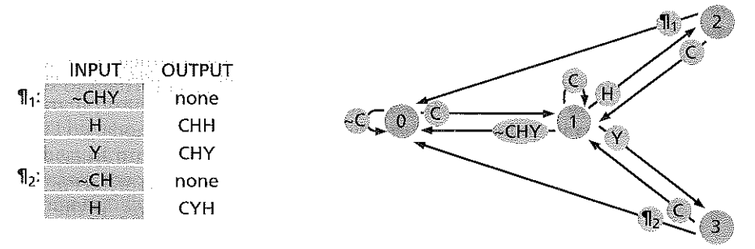


Figure 5.23
The deterministic finite-state automaton that can be used to find instances of the 3-mers CHH, CHY, and CYH in an input sequence. Input starts at state 0. The input that causes a particular transition is given near the start of the relevant arrow. Thus the transition from state 2 to state 1 is triggered by input C. The symbol ~ means "not," so ~CHY means any input except C, H or Y. The transitions 2 → 0 and 3 → 0 are triggered by several different inputs as listed, some of which also result in an output from the automaton. No other transitions produce an output. The output in this example is simply the 3-mer matched in the input, but it could be the residue number of the start of the matching 3-mers, or any other useful information.

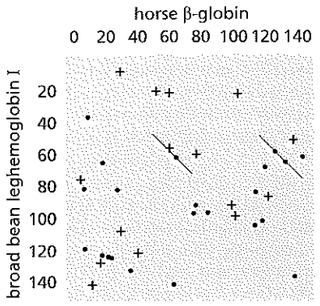


Figure 5.24
Illustration of BLAST word hits for a comparison of horse β-globin and broad bean leghemoglobin I.
The + symbols indicate the 15 hits with $T = 13$, as used in the original BLAST algorithm. All 15 would be extended to give ungapped HSPs. The • symbols show a further 22 hits with $T = 11$, the setting of the more recent gapped BLAST program. From this total of 37 hits, there are two pairs on the same diagonal within 40 residues. Only these two are extended, as shown by the lines. The left-hand one gives the higher ungapped HSP score, and is subsequently extended into a gapped alignment as shown in Figure 5.19. (Redrawn from S.F. Altschul et al., Gapped BLAST and PSI-BLAST: a new generation of protein database search programs, *N.A.R.* 25 (17):3389–3402, 1997, by permission of Oxford University Press.)

as Equation EQ5.29); as in FASTA, nucleotide sequences are by default scored +5 and –4 for matches and mismatches, respectively. Typically, a k-mer of length 3 is used for protein sequences, and of length 11 for nucleotide sequences. In the following discussion, unless explicitly stated, we will assume we are dealing with protein sequences.

For each 3-mer in the query sequence all possible 3-mers that have an alignment score greater than T are generated. This list is then used to scan the database sequence to find all possible similar regions. A typical value for T was 14 in early versions of BLAST when used with the BLOSUM-62 matrix, meaning an average alignment score per residue of over 4. This meant that 3-mers composed only of alanine, isoleucine, leucine, serine, or valine could not reach the threshold (see Figure 5.5), as they all score a maximum of 4. (Some implementations of the algorithm allow the exact 3-mer, i.e., the identical tripeptide, in such a case.) In fact, the large number of negative scores in the BLOSUM-62 matrix, and the maximum possible score of 11 (for aligning two tryptophan residues), mean that all the possible 3-mers are highly similar, if not identical, to the query 3-mer. If T is raised, the number of possible 3-mers is reduced, and they will be even closer to the query 3-mer. This will be the case even for 3-mers including residues that have high scores, such as cysteine and histidine. For example, if $T = 19$, only CHH, CHY, and CYH will score sufficiently highly against the query sequence 3-mer CHH with BLOSUM-62.

For protein sequences there are a total of $20^3 = 8000$ possible different 3-mers, and an n-residue query protein will have $(n - 2)$ 3-mers. Each possible 3-mer must be associated with the position of the query sequence 3-mer. The default choice of parameters can result in about 50 words, making a long list of potentially tens of thousands of k-mers for even an average-length query sequence.

For nucleotide sequences, a word size of 11 is often used, and instead of allowing nonidentical matches, only exact matches to these query words are allowed. In this case, construction of the word list used for searching resembles that in FASTA.

Returning to the case where we are matching nonidentical k-mers, given the list of k-mers, a deterministic finite-state automaton can be constructed such that if the database sequences are input to the automaton, all possible matches will be output in an efficient way. Finite-state automata are best explained with an example, and Figure 5.23 shows one for the CHH 3-mer mentioned above. The figure assumes that only the three 3-mers CHH, CHY, and CYH are to be searched for.

The automaton is started in state 0. The database sequence is input to the automaton one residue at a time. From state 0, only state 1 can be reached, requiring a C, because all desired 3-mers start with a C. Note, however, that all inputs to state 0 that are not C (~C) are regarded as causing a transition from 0 to itself (shown by the curled arrow on the left in Figure 5,23). In fact the results of any input to any state are defined by an unambiguous transition, which is why this is called a **deterministic finite-state automaton.**

From state 1, an H input leads to state 2, which can only be reached this way, requiring an input sequence CH. Similarly, state 3 can only be reached with the input sequence CY. Thus we are gradually building up the desired 3-mers, in a way analogous to the suffix tree (see Figure 5.20). If the input at state 1 is C there is a transition back to state 1, because this second C can be regarded as a second attempt to start one of the 3-mers. All other inputs give a transition back to state 0, from where the search starts all over again with the next input.

From states 2 and 3, unless the input is C, the transition is to state 0. Certain inputs result in the output of a 3-mer name, and these transitions can only occur if the input sequence contains the relevant 3-mer at that point. Thus, if the input

sequence is CHCYHC, the states visited will be 0121301 and the transition to state 0 will be accompanied by the output CYH. Thus the 3-mer has been identified. In the BLAST algorithm, instead of storing the 3-mer sequence, as in the first stage of FASTA the positions of the 3-mers in the two sequences are kept for further analysis in the next stage of the algorithm. An example of the results at this stage is shown in Figure 5.24.

In a real case, with several thousand k-mers to find, a diagram of the automaton is extremely large, with many crossing transitions. Whenever input is such as to require restarting the search with a new initial residue, transition is made to the state that can reuse as much as possible of the failed k-mer. For example, if a state was reached via input ABCD but no desired k-mer has sequence ABCDE, then input E will cause a transition to a state that can only be reached via input of BCDE, CDE, DE, or E, or else will return to state 0. These transitions will be considered in the order given, to try to retain as much information as possible. These transition choices depend only on the k-mer list, and so can be constructed with fixed transitions as soon as the list is available. Using this approach, all suitably high-scoring k-mer matches between the query and database sequences can be listed.

In the original version of BLAST, all k-mer matches scoring above T are extended in both directions without using gaps. Such extended ungapped local alignments are called high-scoring segment pairs (HSPs). The score is monitored and the extension is stopped when the score falls by some set amount X_u from the maximum found so far for this match. This procedure tries to allow an HSP to contain a region of lower similarity. For protein sequences, a typical value X_u for the permitted drop in score is 20. It is possible that the best-scoring HSP has a sufficiently poor-scoring internal region that the extension will stop prematurely, but the parameters are set to try to minimize the likelihood of this occurring. The highest-scoring region (MSP, maximal segment pair) for this pair of sequences is used with some statistical measures discussed in the final part of this chapter to estimate the significance of the alignment. In some cases, more than one HSP may score above some threshold value, in which case two or more HSP scores may be used in determining the significance. The alignment reported by BLAST can be a relatively short stretch of the whole sequence, and no attempt is made to extend it further using gaps.

Newer versions of BLAST take a different approach to generating alignments from the initial hits, called the two-hit method. This starts from the premise that any significant alignment is likely to have at least two high-scoring k-tuple matches on the same diagonal of the alignment matrix. Thus the initial matches are searched to find pairs on the same diagonal within a given distance (typically 40 residues) of each other. The scoring threshold T for k-mers is then lowered, for example to 11 for protein sequences scored with BLOSUM-62, producing more k-mers, and thus more initial hits (see Figure 5.24). However, only a few of these will pair up suitably close to each other on the same diagonal. Only the second hit of such pairs is extended, initially ungapped, as for the older BLAST version. The X_u parameter is used in obtaining the extension, as described above. This ungapped extension must produce an HSP with a score greater than a threshold value S_g if it is not to be discarded. S_g is set such that approximately 2% of database sequences will have an HSP of greater score.

Alignments with scores exceeding S_g are used to seed a dynamic programming calculation of a gapped alignment. This is started from the center of the highest-scoring 11-mer of the HSP. The matrix is filled both forward and backward, as described in Section 5.2, with elements calculated until the score falls a set amount X_g below the current highest-scoring alignment. In this way the amount of calculation is minimized without restricting the alignment to a predetermined band of the matrix. Only one gapped alignment is generated for a database sequence, and its score is used to determine the significance, as discussed in Section 5.4.

The parameters of the gapped BLAST program are set to make it approximately three times as fast as the ungapped version, yet more sensitive. This is achieved by severely reducing the number of extensions attempted. A gapped extension takes approximately 500 times as long as an ungapped one.

Comparing a nucleotide sequence directly with a protein sequence requires special modifications to the BLAST and FASTA algorithms

There are several situations where a comparison between a nucleotide sequence and a protein sequence is necessary. Most protein databases tend to be nonredundant and only contain highly reliable sequences (that is, minor variants and many hypothetical genes are excluded). When analyzing new sequences, a much stronger signal for significant similarity can be obtained by comparing against protein sequences. Thus, new nucleotide sequences are often compared with the protein sequence databases. Conversely, there are occasions when one wants to search for homologous proteins in large genomic sequences.

Two new problems arise in these circumstances. First, a nucleotide sequence can be translated into protein in six different reading frames (three on each strand), so that there are six different potential protein sequences to be examined for each nucleotide sequence. Second, insertion or deletion errors can be present in the nucleotide sequence. This can result in the actual protein sequence being in different reading frames for different parts of the sequence, a situation called **frameshift**. The algorithms described earlier for comparing protein with protein or nucleotide with nucleotide sequences require modifications to allow for these new factors.

In the BLAST program suite the two programs blastx and tblastx allow searches with different reading frames, but neither allows for frameshifts. In addition to the 20 possible amino acids, there may be some stop codons present, and by default, the score for aligning a stop codon with an amino acid is taken as the most negative score in the substitution matrix, although other scoring schemes are possible. Both these programs convert the nucleotide sequence into protein sequence and then work exactly as a standard protein BLAST search. The only difference is that the query-sequence reading frame used in each alignment is reported.

The FASTA program suite contains four programs relevant to this problem. Two of these (tfastx and tfasty) are for searching nucleotide databases with protein sequences, and the others (fastx and fasty) are for nucleotide query-sequence searches of protein databases. All these programs can account for frameshifts to some degree, as well as the alternative reading-frames, and thus are, in principle, more powerful than blastx at generating suitable alignments. However, in practice blastx is still very useful, and all these programs have their place in database searching.

The fastx and tfastx programs use an algorithm that only allows for nucleotide insertions and deletions. The possibility of a base being incorrectly given, for example an A where there should have been a T, is not considered. The nucleotide sequence is translated in all six frames, each set of three being considered in a single run of the alignment program.

Each of the three reading frames is analyzed in a separate matrix. However, at each step, the possibility of moving between matrices is considered. Thus, the alignment could arrive at matrix element $S_{i,j}$ from $S_{i-1,j-1}$ in the same frame, or from $T_{i-1,j-1}$ or $U_{i-1,j-1}$ in the alternative frames. The other $(i-1,j$ and $i,j-1)$ elements are also possible sources of the new (i,j) element. Any move between matrices incurs an extra penalty — the frameshift penalty. This can be set higher when the sequences are expected to be more accurate, to prevent excessive numbers of frameshifts in the alignments, and is often set a little higher than the gap-opening penalty. The full details of the algorithm will not be given here, but a sequence example is given to illustrate the effects of frameshifts.

Box 5.2 Sometimes things just aren't complex enough

Many protein and nucleotide sequences contain regions that can be described as being of low compositional complexity: low-complexity regions or **simple sequences**. Examples include stretches of identical amino acids in proteins, repeated short sequences, and longer DNA repeats (see Box 1.1). It is estimated that roughly half of all database sequences contain at least one such region. If alignments are made with these regions present, many spurious similarities will be reported because many unrelated sequences contain similar low-complexity regions. These stretches also cause problems in database searches because they are nonrandom, violating the assumptions on which the calculations of statistical significance are based (Section 5.4). It is important to be able to define these stretches and mask them to prevent their biasing the database search results. Many databases have such sequences masked when they are made available for sequence searches, so that often there is only a need to find these regions in the query sequence.

Three different properties of these simple sequences can be distinguished. Precisely repeating sequences are referred to as patterns. These can be very short, for example ATT, and are not necessarily in a single contiguous block, for example ATTCATTGCATTATT. If there is a clear period of repeat—for example, ATTATTATTATT has a repeat of three—this is called a periodicity. The periodicity need not necessarily be an integer, as can occur, for example, in an amino acid repeat relating to an α-helix. Furthermore, patterns and periodicities can both involve some degree of error, in that the repeat need not be exact. The final property that we will discuss is the compositional complexity, which is a measure of bias in the sequence composition.

In general terms, a sequence of length L is made up from N_{type} possible different components (i.e., N_{type} is 20 for proteins, 4 for nucleic acids) in a composition defined by the ath component occurring n_a times. The general complexity-state vector is defined as a list of these integers n_a in numerical order, disregarding the specific components. Thus both nucleotide sequences ATA and CAC are represented by the same vector {2, 1, 0, 0}. The number of distinct sequences of length L with this composition is given by

$$\frac{L!}{\prod_{a=1}^{N_{\text{type}}} n_a!} \qquad \text{(BEQ5.2)}$$

(The term $L!$ is called "L factorial", and means the product $L \times (L-1) \times (L-2) \ldots 3 \times 2 \times 1$, so $4! = 4 \times 3 \times 2 \times 1 = 24$. By definition, $0! = 1$.) The n_a can vary in value from 0 to L, and will always sum to L. The number of these n_a that have the value c is written r_c, and the r_c will sum to N_{type}. The number of different compositions that will give rise to the same complexity-state vector is given by

$$\frac{N_{\text{type}}!}{\prod_{c=0}^{L} r_c!} \qquad \text{(BEQ5.3)}$$

Thus the total number of distinct sequences corresponding to a particular complexity-state vector is

$$\frac{L!}{\prod_{a=1}^{N_{\text{type}}} n_a!} \times \frac{N_{\text{type}}}{\prod_{c=0}^{L} r_c!} \qquad \text{(BEQ5.4)}$$

The more distinct sequences available to a complexity state, the more complex the sequence.

As an example consider the five-nucleotide sequence ATTAT. The composition of this sequence can be represented as $\{T_3, A_2, C_0, G_0\}$ where the bases have been ordered according to their abundance. There are 10 possible sequences with the same composition: AATTT, ATATT, ATTAT, ATTTA, TAATT, TATAT, TATTA, TTAAT, TTATA, and TTTAA. Note that this is 5!/(3!2!). Now consider how many different compositions are possible by switching the bases around but maintaining the proportion of bases at 3:2:0:0. There are 12 of these: A_3C_2, A_3G_2, A_3T_2, C_3A_2, C_3G_2, C_3T_2, G_3A_2, G_3C_2, G_3T_2, T_3A_2, T_3C_2, and T_3G_2. Note that this is 4!/(2!1!1!). Therefore for the general complexity state represented by the vector {3, 2, 0, 0} there are a total of $10 \times 12 = 120$ distinct DNA sequences. For comparison, the complexity state represented by vector {5, 0, 0, 0} has only four unique DNA sequences [(5!/5!) × 4!/(3!1!)], compared to 360 for the state {2, 2, 1, 0} [5!/(2!2!1!) × 4!/(2!1!1!)].

Several programs are available that attempt to distinguish between simple and other regions of a sequence. We will only discuss one, the program SEG, which uses compositional complexity as a measure to determine regions of simple sequences. SEG works in two steps to determine regions that satisfy certain complexity constraints. The compositional complexity I_{SEG}, which is a measure of the information required per sequence position to specify a particular sequence, given the composition, is defined by

$$I_{\text{SEG}} = \frac{1}{L} \log_{N_{\text{type}}} \left(\frac{L!}{\prod_{a=1}^{N_{\text{type}}} n_a!} \right) \qquad \text{(BEQ5.5)}$$

Box 5.2 Sometimes things just aren't complex enough (continued)

For example, for a five-nucleotide sequence, I_{SEG} can vary from 0 for {5, 0, 0, 0} to ~1.92 for {2, 1, 1, 1}. In the first step of the SEG method, a more computationally efficient approximation is used to search the sequence. Windows of length L are identified for which the value of

$$I'_{SEG} = -\sum_{a=1}^{N_{type}} \frac{n_a}{L}\left(\log_2 \frac{n_a}{L}\right) \qquad \text{(BEQ5.6)}$$

is less than a given threshold I_{SEG1}. Note that I'_{SEG} approximates I_{SEG} for large L. These initial low-complexity regions are augmented by any overlapping windows that have a value of I'_{SEG} that is exceeded by a less-strict threshold I_{SEG2} (that is, $I_{SEG2} > I_{SEG1}$).

In the second step, SEG determines the sub-sequence of each initial low-complexity region whose composition has the least probability of occurrence, based on a model with all residues equally likely. The probability is calculated using the formula

$$P_{SEG} = \frac{1}{N_{type}^{L}}\left(\frac{L!}{\prod_{a=1}^{N_{type}} n_a!}\right)\left(\frac{N_{type}!}{\prod_{c=0}^{L} r_c!}\right) \qquad \text{(BEQ5.7)}$$

By inclusion of the first term, this value can be compared for different window sizes.

Versions of SEG are available that are designed to search for specific periodicities, which can be useful in some instances. For example, a number of proteins have regions of low complexity, often short repeats, which can indicate nonglobular structure. Since these sequences tend not to be exact repeats, SEG can be a powerful tool for identifying these regions. DUST is an equivalent program for DNA sequences. There are other programs designed to search for specific known repeat sequences such as the DNA repeats found in many genomes, usually defined in a small repeat database.

Consider the following short stretch of sequence and three forward translations:

A C C A G A G C C A A C T

Frame 1	T	R	A	N
Frame 2	P	E	P	T
Frame 3	Q	S	Q	

The translations have been placed under the central base of each codon, so that T is placed under the first C of ACC. Consider all the possible moves, including frameshifts, from a matrix element at position 2 in the translated sequence to one at position 3. This means that the alignment of TR, PE, and QS has already been considered, and we are now considering adding A, P, or Q, referring to frame 1, 2, or 3, respectively. When translated back into nucleotides, the possible interpretations of the sequence are:

1 → 2 (AGA)*G*(CCA)

1 → 3 (AGA)*GC*(CAA)

2 → 1 (GA(**G**)CC) or (GAG)*CC*(AAC)

2 → 3 (GAG)*C*(CAA)

3 → 1 (A(**GC**)C) or (AGC)*C*(AAC)

3 → 2 (AG(**C**)CA) or (AGC)*CA*(ACT)

where codons are in parentheses. Bases that are ignored in translation (deletions) are in italic, while those used in two successive codons (insertions) are in bold. Thus the sequence written (A(**GC**)C) is interpreted as (AGC)(GCC). Similarly (AGC)*CA*(ACT) is interpreted as (AGC)(ACT). Note that deletions and insertions only occur at the boundaries of codons; that is, the sequence ACGT is only interpreted as ACG or CGT, never ACT or AGT.

A greater variety of errors in the nucleotide sequence can be allowed for by fasty and tfasty. Any two, three, or four consecutive nucleotides can be interpreted as a codon, of which only the middle alternative does not involve an indel. In each case, base errors are considered. A penalty scheme for modifying the codons is combined with a BLOSUM-50 scoring of the aligned amino acids to find the best codon for that alignment; if indels are involved they also incur a penalty. Note that in this case, as well as allowing all four interpretations of ACGT mentioned above, base-pair changes are also considered. If the base change and frameshift penalties are not sufficiently punitive, almost any pair of sequences could be aligned with a high score. Both of these are commonly set to up to twice the gap opening penalty.

5.4 Alignment Score Significance

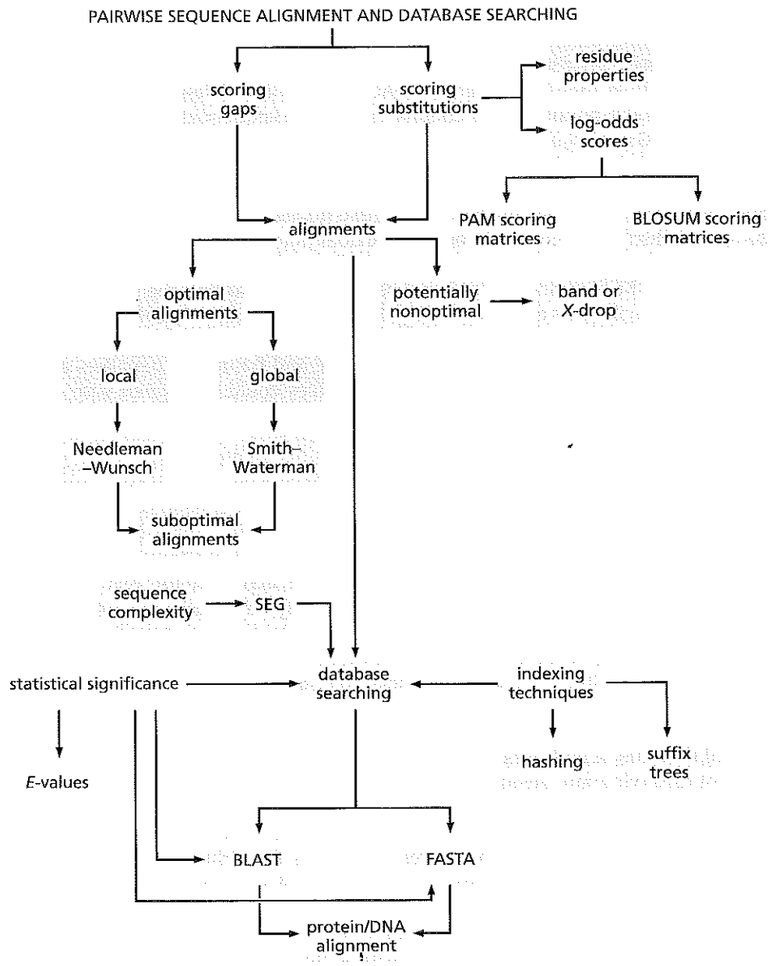
In this section we will examine how to determine whether the score of an optimal alignment is significantly higher than would be expected for two unrelated sequences (see Flow Diagram 5.4). This is not the only way of trying to assess significance, but it is probably the most sensitive currently available. The simpler technique of observing the percentage of identical or similar residues in the alignment (see Section 4.2), although useful, is far less precise.

If the alignment scores were normally distributed—that is, Gaussian—then knowledge of their mean and standard deviation would allow us to calculate the probability of observing any given score using standard tables. The situation is not so simple, however, because the score that is reported in a database search is that of the optimal alignment; that is, it is the best possible score for that particular pair of sequences. This means that the scores are always from the extreme end of the distribution of all alignment scores.

The statistics of optimal alignment scores have only been rigorously derived in the case of ungapped local alignments, for which the scores follow an extreme-value distribution. Using this theory, precise evaluation of score significance for ungapped local alignments is readily calculable. For gapped local alignments, only approximations are available for the statistical score distribution. Both gapped and ungapped local alignments have been found to have very similar distributions, but differ in their parameterization.

Before discussing the details of the practical application of these score distributions, two general points about scores have to be made. First, one can ask how much information is required to define the position of an alignment in two sequences. Information is usually measured as bits, which can be regarded as yes or no answers. For example, distinguishing the numbers 0–255 requires eight digits in a binary (i.e., 0 or 1) number representation, which is $\log_2 256$. For a sequence of length m, we have to distinguish among the m possible alignment starting positions, which in general requires $\log_2 m$ bits of information. If this sequence is aligned to an n-residue sequence, positioning the start of the alignment on this second sequence will require a further $\log_2 n$ bits of information. Therefore the alignment requires a total of $\log_2 m + \log_2 n = \log_2(mn)$ bits of information to define its start position on both sequences.

Flow Diagram 5.4
The key concept introduced in this section is that when comparing optimal alignments between a query sequence and a database of sequences, care must be taken to evaluate the true significance of the alignment score, since even the best scoring database sequence cannot be assumed to show significant similarity.



In performing a database search, if m is the length of the query sequence, then n is the total length of the database entries. Considering a protein sequence database of 100 million residues and an average-length query sequence of 250 residues, a score of approximately 35 bits would be required to define the location of the best alignment between the sequences.

The second general aspect of optimal alignment scores relates to how they vary with the length of the alignment. Waterman has shown that for global alignments with gaps, the score grows linearly with sequence length. For local alignments the situation is more complex, and depends on a property of the substitution matrix used. If the expected score of a random sequence with a given substitution matrix, given by Equation EQ5.3 is positive, the local alignment score grows linearly with sequence length regardless of the gap penalties. However, if the expected score is negative, which is the case for most amino acid substitution matrices in common use, then unless gap penalties are very low, the optimal local alignment score will grow logarithmically; that is, as log n where n is the number of residues. In what follows, it is assumed we are using scoring parameters such that the scores grow logarithmically with sequence length. The alignment scores need to be corrected for length as part of the process of determining their significance.

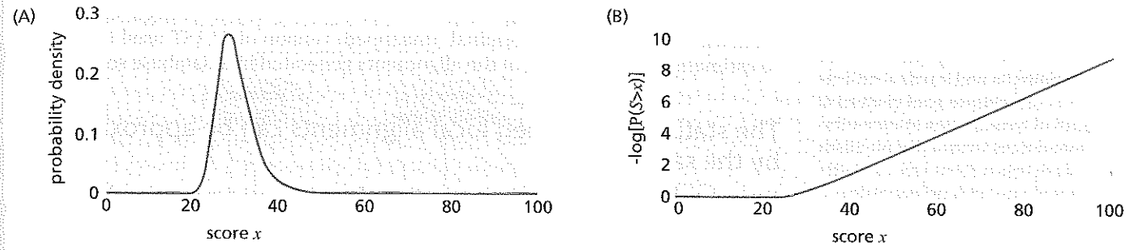


Figure 5.25
The extreme-value distribution of optimal alignment scores. (A) The distribution for λ and K of 0.286 and 0.055, respectively, for two sequences both of length 245 residues. (These values are from Altschul and Gish (1996) for a BLOSUM-62 substitution matrix, with affine gap opening and extension penalties –12 and –1, respectively.) (B) The probability, given the distribution in part A of an observed score S being at least x, plotted so that a value y on the vertical axis represents a probability 10^{-y}.

The distribution of alignment scores for optimal ungapped local alignments has been rigorously derived from first principles. A database search only carries out further analysis of the highest-scoring alignment of each database entry with the query sequence. Omitting the considerable amount of analysis required, it can be proved that the optimal ungapped local alignment score follows the Gumbel extreme-value distribution. With m and n defined as above, this distribution peaks at a value

$$U = \frac{\ln(Kmn)}{\lambda} \qquad \text{(EQ5.30)}$$

where K and λ are constants that depend on the scoring matrix used and the sequence composition. λ is a scaling parameter of the substitution matrix, and is the unique positive solution of the equation

$$\sum_{a,b} p_a p_b e^{s_{a,b}\lambda} = 1 \qquad \text{(EQ5.31)}$$

where the summation is over all residue types a and b, the p_a and p_b are frequencies of occurrence of the residues as defined in Section 5.1, and $s_{a,b}$ are the substitution scores.

The probability of the alignment score S being less than x is given by the cumulative distribution function

$$P(S < x) = \exp\left(-e^{-\lambda(x-U)}\right) \qquad \text{(EQ5.32)}$$

from which the complementary probability of the score being at least x is

$$P(S \geq x) = 1 - \exp\left(-e^{-\lambda(x-U)}\right) \qquad \text{(EQ5.33)}$$

Substituting for U we arrive at the formula for the probability of obtaining an alignment of score S greater than a value x:

$$P(S \geq x) = 1 - \exp\left(-Kmne^{-\lambda x}\right) \qquad \text{(EQ5.34)}$$

The extreme-value distribution is shown in Figure 5.25. The key feature of interest is the tail for high values. Notice that this decays much more slowly than the low-value tail; that is, the distribution is asymmetric with a bias to high values. It is important that we have the correct distribution if we are to estimate the significance of any given score accurately. In general, if $P(S \geq x)$ is less than 0.01, the alignment is significant at the 1% level. The level chosen as cut-off depends on the particular problem, and is discussed in more detail in Section 4.7.

Formulae exist for calculating K and λ for a given substitution matrix and sequence database composition. The original (ungapped) version of BLAST used this theory to estimate the significance of the alignments generated in a database search.

The statistics of gapped local alignments can be approximated by the same theory

For gapped local alignments, examination of actual database searches has shown that the scores of optimal alignments also fit an extreme-value distribution. However, in this case there is no rigorous theory to provide the parameters λ and K. These can be estimated by studying sample database searches for particular values of the scoring schemes (including gap penalties). Note, however, that they will depend on the composition of the database sequences, so that the parameterization should really be done according to the actual database to be searched. Both BLAST and FASTA use such methods to derive score significance.

The programs in the BLAST and FASTA suites report E-values, which are related to the probability $P(S \geq x)$. The value of P is calculated for the particular lengths of the query and database sequences, and varies from 0 to 1. The E-values are obtained from this by multiplying by the number of sequences in the database. Thus, if there are D database sequences, the E-values range from 0 to D.

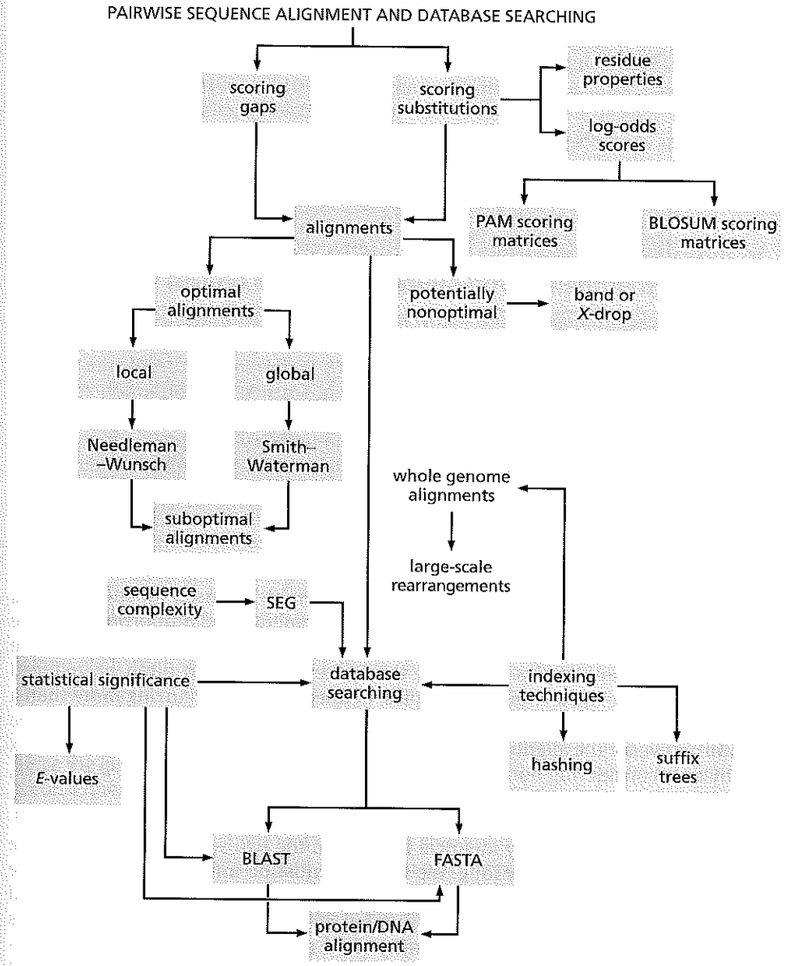
The E-value is the number of database sequences not related to the query sequence that are expected to have alignment scores greater than the observed score. Thus an E-value of 1 is not significant, as one unrelated database sequence would be expected to have such a score. The E-value deemed to indicate a significant similarity as shown by the alignment score is a practical problem discussed in Section 4.7.

5.5 Aligning Complete Genome Sequences

As more genome sequences have become available, of different species and also of different bacterial strains, there are several reasons for wanting to align entire genomes to each other. Such alignments can assist the prediction of genes because if they are conserved between species this will show up clearly in the alignment. In addition, other regions of high conservation between the genomes can indicate other functional sequences. The study of genome evolution needs global alignments to identify the large-scale rearrangements that may have occurred. Although in principle the dynamic programming methods presented in Section 5.2 could be used, the lengths of the sequences involved makes the demands on computer time and disk space prohibitive. As in the case of database searches, other techniques must be used that are not guaranteed to find the optimal scoring alignment.

When aligning two complete genome sequences, problems arise that are quite distinct from those discussed above in relation to aligning two related proteins or making simple database searches. In the case of complete genomes, the alignment problem is more complex because the intergene regions may be subject to higher mutation rates; in addition, large-scale rearrangements may have occurred. Many discrete, locally similar segments may exist, separated by dissimilar regions. Unlike in the BLAST and FASTA methods, therefore, it is insufficient to determine a single location from which to extend the alignment. The solution is to modify the indexing techniques for genome alignments to locate a series of anchors. The relationship of this subject to the other topics discussed in this chapter is shown in Flow Diagram 5.5.

A second problem is the linking together of the anchors to form a scaffold for the alignment, and doing this in such a way as to identify the large-scale rearrangements.



Flow Diagram 5.5
The key concept introduced in this section is that when aligning extremely long sequences, refinements are required in order to make the alignment search more efficient. The section also deals with the observation that long sequences often show large-scale rearrangements, a feature that needs to be included in the alignment methods.

Both of these problems require complex solutions that are only briefly outlined here. This area is still undergoing rapid development, and new techniques may yet emerge that are a significant advance in the field.

Another application involving a complete genome sequence is the alignment to it of many smaller nucleotide sequences. These could be from a variety of sources, including sequence data from related species at various stages of assembly. The latter task, in particular, can give rise to the problems noted above.

Indexing and scanning whole genome sequences efficiently is crucial for the sequence alignment of higher organisms

Both FASTA and BLAST use indexing techniques to speed up database searches, and both index the query sequence. For genome alignments it is now practical to index the complete genome sequence, as long as the computer used has sufficient memory to store the whole index. In contrast to usual database searches with single gene sequences, this requires careful planning of the data storage. If the index is larger than the available computer memory, the time required for the alignment

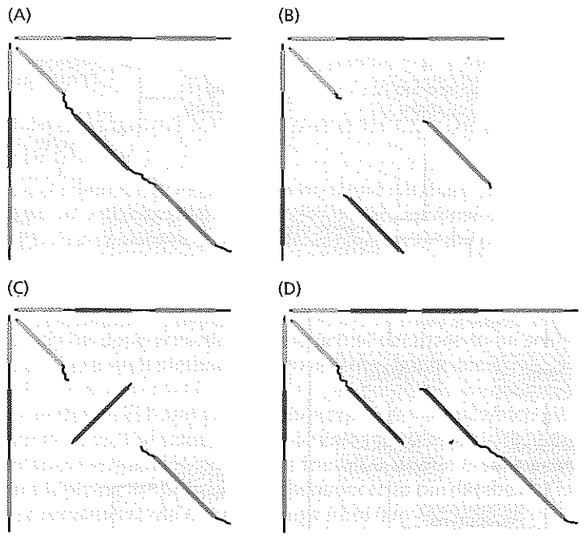


Figure 5.26
Four examples of possible relationships between two long nucleotide sequences. (A) Both sequences are similar along their whole length, with three particularly similar segments identified. No rearrangements during evolution from the most recent common ancestor need be proposed. This is the only case in which a global alignment can be given. (B) A translocation has changed the order of the red and blue segments for one of the sequences. (C) The red segment has been inverted in the horizontal sequence. (D) The red segment has been duplicated in the horizontal sequence. The black lines indicate regions of alignment outside the main segments. (Adapted from M. Brudno et al., Global alignment: finding rearrangement during alignment, *Bioinformatics* 19, Supplement 1:i54–i62, 2003.)

can increase by a factor of four or more. If suffix trees are used for the indexing, then with efficient techniques the space requirements are proportional to the sequence length. Hashing methods require storage according to the lengths of the sequence and the k-tuples. However, the space required can often be reduced significantly if the most common k-tuples, defined as those occurring more than a threshold number of times, are omitted on the grounds that they are likely to be uninformative for the alignment. The hashing methods often also reduce the index size by only considering nonoverlapping k-tuples; that is, bases $1 \rightarrow k$, $k+1 \rightarrow 2k$, and so on.

The index is used to identify identical k-tuples in the two sequences. Because the sequences involved are very long, the k-tuples must be much longer than in the example of database searching discussed above, or many matches will be found with random sequence. The expected number of matches depends on the lengths of the k-tuples and the percentage identity of the aligned sequences, and was thoroughly analyzed during development of the BLAT (BLAST-like Alignment Tool) package. k-tuples of 10–15 bases are used by the Sequence Search and Alignment by Hashing Algorithms (SSAHA) program, and lengths of 20–50 bases have been used in suffix tree methods.

One technique that has proved to be particularly effective is the spaced seed method. When searching using the long k-tuples mentioned above, all k positions of the k-tuples are compared. It has been found that these are not the most efficient

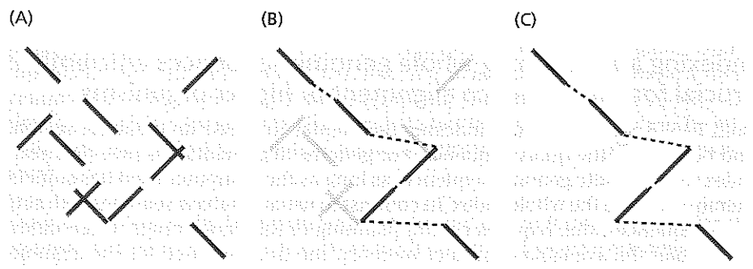


Figure 5.27
An example of the application of the SLAGAN method. (A) The set of identified potential anchors for the alignment of two long nucleotide sequences. (B) The path of anchors identified. The anchors are ordered progressively along the vertical sequence but not the horizontal one because of large-scale rearrangements. (C) The regions identified as locally consistent. Individual alignments are determined for each of the three regions. (Adapted from M. Brudno et al., Global alignment: finding rearrangement during alignment, *Bioinformatics* 19, Supplement 1:i54–i62, 2003.)

probes for identifying matches that are useful seeds for genome alignment. In the development of the PatternHunter program it was found that searching for a specific pattern of 12 bases in a stretch of 19 was more effective. The 19mer can be represented as 1110100110010101111, where 1 indicates a position whose match is sought, and 0 indicates a position whose base is disregarded. The later versions of the BLASTZ program made a further improvement by allowing any one of the 12 positions to align bases A and G or bases C and T, so long as the other 11 sites were perfect matches.

The complex evolutionary relationships between the genomes of even closely related organisms require novel alignment algorithms

Many anchor points must be identified between the two genome sequences, each initially as matched k-tuples which are usually extended in a gapless way as in FASTA and BLAST. The original MUMmer approach was to include only unique k-tuples common between the sequences; that is, k-tuples present just once on each sequence. Such matches can be expected with confidence to identify truly homologous segments. Most methods (including MUMmer 2) are not so restrictive, and include k-tuples that occur several times on a sequence. As a consequence, many matches are recorded, some of which are expected to be incorrect.

In contrast to the database search methods, which only need to select a single anchor for the alignment that is then extended using dynamic programming, for genome alignment a set of anchors is needed that spans the lengths of the genomes, identifying the true homologous segments. This step is usually similar to the third step of FASTA (see Figure 5.22C) and involves dynamic programming where each anchor and potential gap is given a score. Anchors are not allowed to overlap and must be arranged sequentially along the sequences.

In addition, a number of large-scale sequence rearrangements may have occurred since the last common ancestor of the species whose sequences are being aligned. As a consequence, the correct alignment will not have the straightforward character of the alignments discussed so far. Some examples of common rearrangements are shown in Figure 5.26, and a real example is shown for aligning equivalent chromosomes in mouse and rat in Figure 10.21. Most existing programs cannot automatically recognize large-scale rearrangements and thus cannot correctly assign segments to the overall alignment. An exception is the Shuffle-Limited Area Global Alignment of Nucleotides (SLAGAN) program, which attempts to identify rearrangements. The FASTA-type step referred to above requires each successive anchor to occur in succession along both sequences. However, any rearrangements that produce sequence reversal of a segment (such as in Figure 5.26C) will result in the successive anchors occurring progressively but in opposite directions along the two sequences. SLAGAN only requires the anchors to occur in succession along one of the sequences. A further step then identifies those regions that contain anchors that are locally consistent with a sequence reversal segment. An alignment is obtained for each of these regions. An example of a successful application of this method is shown in Figure 5.27.

In all the methods, an attempt is made to extend identified homologous segments, often using standard alignment techniques but trying to identify the limits beyond which the sequences are not recognizably similar. The alignments derived by these methods are often fragments separated by unaligned regions.

Summary

In the first part of this chapter, we looked at the derivation of the specialized algorithms that are used in automated methods to determine the optimal alignment between two sequences. Such algorithms are needed because if the insertion of

gaps is allowed there are a very large number of possible different alignments of two sequences. Before an algorithm can be applied, a scoring scheme has to be devised so that the resulting alignments can be ranked in order of quality of matching. Scoring schemes involve two distinct components: a score for each pair of aligned residues, which assesses the likelihood of such a substitution having occurred during evolution, and a penalty score for adding gaps to account for the insertions or deletions that have also occurred during evolution. The first score has its foundations in the concept of log-odds, and is assigned on the basis of reference substitution matrices that have been derived from the analysis of real data in the form of multiple alignments. The penalty scores given to gaps have a more ad hoc basis, and have been assigned on the basis of the ability of combined (substitution and gap) scoring schemes to reproduce reference sequence alignments. (Confidence in these reference alignments is usually based on a combination of structural information and regions of sufficiently high similarity.)

The automated construction and assessment of gapped alignments only became possible with the development of dynamic programming techniques, which provide a rigorous means of exploring all the possible alignments of two sequences. The essence of dynamic programming is that optimal alignments are built up from optimal partial alignments, residue by residue, such that nonoptimal partial alignments are efficiently identified and discarded. The technique ensures that the optimal alignment(s) will be identified. A number of different schemes have been developed, which treat gaps in different ways. Only minor variations in the dynamic programming algorithms are required to alter the type of alignment produced from global to local, or to identify repeat or suboptimal alignments. Thus the same technique can be used to answer several different problems of sequence similarity and relatedness.

The problem with dynamic programming methods is that despite their efficiency they can place heavy demands on computer memory and take a long time to run. The speed of calculation is no longer as serious a barrier as it has been in the past, but the problem of insufficient computer memory persists, particularly as there are now many very long sequences, including those of whole genomes, available for comparison and analysis. Some modifications of the basic dynamic programming algorithm have been made that reduce the memory and time demands. One way of reducing memory requirements is by storing not the complete matrix but only the two rows required for calculations. However, to recover the alignment from such a calculation takes longer than if all the traceback information has been saved. By only calculating a limited region of the matrix, commonly a diagonal band, both time and space saving can be made, although at the risk of not identifying the correct optimal alignment.

Often the first step in a sequence analysis is to search databases to retrieve all related sequences. Such searches depend on making pairwise alignments of the query sequence against all the sequences in the databases, but because of the scale of this task, fast approximate methods are usually used to make such searches more practicable. The algorithms for two commonly used search programs—BLAST and FASTA—make use of indexing techniques such as suffix trees and hashing to locate short stretches of database sequences highly similar or identical to parts of the query sequence. Attempts are then made to extend these to longer, ungapped local alignments which are scored, the scores being used to identify database sequences that are likely to be significantly similar. This process is considerably faster than applying full-matrix dynamic programming to each database sequence. At this point, both techniques revert to the more accurate methods to examine the highest-scoring sequences, in order to determine the optimal local alignment and score, but this is only done for a tiny fraction of the database entries.

There are instances where it is necessary to align a protein sequence with a nucleotide sequence. In such cases one solution is simply to translate the

nucleotide sequence in all possible reading frames and then use protein–protein alignments. However, this approach is rather bad at dealing with errors in the sequence that give rise to frameshifts. The dynamic programming method can be modified to overcome this problem by using three matrices, one for each possible reading frame in the given strand direction, with a set of rules for moving from one matrix to another. In this way, an optimal alignment can be generated that uses more than one reading frame, simultaneously proposing sequence errors.

Another problem that arises in database searches is the occurrence of regions of low sequence complexity, which can cause spuriously high alignment scores for unrelated sequences. Such regions can be defined either by identifying a repeating pattern or on the basis of their composition, and can then be omitted from the comparison.

To be sure that two sequences are indeed homologous, it is important to know when the alignment score reported is statistically significant. In the case of sequence alignments, the statistical analysis is very difficult; in fact the theory has not yet been fully developed for alignments including gaps. Part of the reason for this difficulty is that the alignments reported are always those with the best scores. These will not be expected to obey a normal distribution, but rather an extreme-value distribution. Furthermore, the scores are also dependent on the scoring scheme used, the sequence composition and length, and the size of the database searched. For alignments without gaps, formulae have been derived that allow the score to be converted to a probability from which the significance can be gauged.

The last section of the chapter deals with the alignment of very long sequences, for example those of whole genomes. The straightforward application of dynamic programming is often not feasible because of the lack of computer resources. This difficulty can be overcome by using similar indexing methods to those in the database search programs. However, there are significant differences, which lead to the indexing being applied to the genome sequence rather than the query sequence, and to searching for much longer identical segments than in database searches. Nevertheless, there is often an additional problem because frequently there are large-scale genome rearrangements even over relatively short periods of evolutionary time. To overcome this, local alignments must be identified that can then be joined together into a global alignment by a specific dynamic programming technique that allows for translations and inversions of segments of the sequence.

Further Reading

5.1 Substitution Matrices and Scoring

The PAM (MDM) substitution scoring matrices were designed to trace the evolutionary origins of proteins

Altschul SF (1991) Amino acid substitution matrices from an information theoretic perspective. *J. Mol. Biol.* 219, 555–565.

Dayhoff MO, Schwartz RM & Orcutt BC (1978) A model of evolutionary change in proteins. In Atlas of Protein Sequence and Structure, vol 5 suppl 3 (MO Dayhoff ed.), pp 345–352. Washington, DC: National Biomedical Research Foundation.

Jones DT, Taylor WR & Thornton JM (1992) The rapid generation of mutation data matrices from protein sequences. *Comput. Appl. Biosci.* 8, 275–282. *(The PET91 version of PAM matrices.)*

Yu Y-K, Wootton JC & Altshul SF (2003) The compositional adjustment of amino acid substitution matrices. *Proc. Natl Acad. Sci. USA* 100, 15688–15693.

See appendix for deriving target frequencies.

The BLOSUM matrices were designed to find conserved regions of proteins

Henikoff S & Henikoff JG (1992) Amino acid substitution matrices from protein blocks. *Proc. Natl Acad. Sci. USA* 89, 10915–10919.

Scoring matrices for nucleotide sequence alignment can be derived in similar ways

Chiaromonte F, Yap VB & Miller W (2002) Scoring pairwise genomic sequence alignments. *Pac. Symp. Biocomput.* 7, 115–126.

The substitution scoring matrix used must be appropriate to the specific alignment problem

Altschul SF (1991) Amino acid substitution matrices from an information theoretic perspective. *J. Mol. Biol.* 219, 555–565.

May ACW (1999) Towards more meaningful hierarchical classification of amino acid scoring matrices. *Protein. Eng.* 12, 707–712. *(A comparative study of substitution matrices in terms of the way they group amino acids.)*

Yu Y-K & Altschul SF (2005) The construction of amino acid substitution matrices for the comparison of proteins with non-standard compositions. *Bioinformatics* 21, 902–911.

Yu Y-K, Wootton JC & Atschul SF (2003) The compositional adjustments of amnio acid substitution matrices. *Proc. Natl Acad. Sci USA* 100, 15688–15693.

Gaps are scored in a much more heuristic way than substitutions

Goonesekere NCW & Lee B (2004) Frequency of gaps observed in a structurally aligned protein pair database suggests a simple gap penalty function. *Nucleic Acids Res.* 32, 2838–2843. *(This recent paper on scoring gaps for protein sequences includes a good listing of older work.)*

5.2 Dynamic Programming Algorithms

Waterman MS (1995) Introduction to Computational Biology: Maps, Sequences and Genomes, chapter 9. London: Chapman & Hall. *(A more computational presentation of dynamic programming alignment algorithms.)*

Optimal global alignments are produced using efficient variations of the Needleman–Wunsch algorithm

Gotoh O (1982) An improved algorithm for matching biological sequences. *J. Mol. Biol.* 162, 705–708.

Needleman SB & Wunsch CD (1970) A general method applicable to the search for similarities in the amino acid sequence of two proteins. *J. Mol. Biol.* 48, 443–453.

Local and suboptimal alignments can be produced by making small modifications to the dynamic programming algorithm

Smith TF & Waterman MS (1981) Identification of common molecular subsequences. *J. Mol. Biol.* 147, 195–197.

Waterman MS & Eggert M (1987) A new algorithm for best subsequence alignments with application to tRNA–tRNA comparisons. *J. Mol. Biol.* 197, 723–728.

Time can be saved with a loss of rigor by not calculating the whole matrix

Zhang Z, Berman P & Miller W (1998) Alignments without low-scoring regions. *J. Comput. Biol.* 5, 197–210.

Zhang Z, Scwartz S, Wagner L & Miller W (2000) A greedy algorithm for aligning DNA sequences, *J. Comput. Biol.* 7 (1–2), 203–214.

5.3 Indexing Techniques and Algorithmic Approximations

Suffix trees locate the positions of repeats and unique sequences; Hashing is an indexing technique that lists the starting positions of all k-tuples

Gusfield D (1997) Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology. Cambridge: Cambridge University Press.

Waterman MS (1995) Introduction to Computational Biology: Maps, Sequences and Genomes, chapter 8. London: Chapman & Hall.

The FASTA algorithm uses hashing and chaining for fast database searching; The BLAST algorithm makes use of finite-state automata

Altschul SF, Gish W, Miller W et al. (1990) Basic Local Alignment Search Tool. *J. Mol. Biol.* 215, 403–410. (*The original BLAST paper.*)

Altschul SF, Madden TL, Schäffer AA et al. (1997) Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. *Nucleic Acids Res.* 25, 3389–3402.

Pearson WR & Lipman DJ (1988) Improved tools for biological sequence comparison. *Proc. Natl Acad. Sci. USA* 85, 2444–2448. *(The original FASTA paper.)*

Pearson WR, Wood T, Zhang Z & Miller W (1997) Comparison of DNA sequences with protein sequences. *Genomics* 46, 24–36. (*fastx, fasty.*)

Box 5.2: Sometimes things just aren't complex enough

Morgulis A, Gertz EM, Schäffer AA & Agarwala R (2006) A fast and symmetric DUST implementation to mask low-complexity DNA sequences. *J. Comput. Biol.* 13, 1028–1040.

Wootton JC & Federhen S (1996) Analysis of compositionally biased regions in sequence databases. *Methods Enzymol.* 266, 554–571 (*SEG.*)

5.4 Alignment Score Significance

Altshul SF & Gish W (1996) Local alignment statistics. *Methods Enzymol.* 266, 460–480.

Mott R (2000) Accurate formula for P-values of gapped local sequence and profile alignments. *J. Mol. Biol.* 300, 649–659.

Waterman MS (1995) Introduction to Computational Biology: Maps, Sequences and Genomes, chapter 11. London: Chapman & Hall.

5.5 Alignments Involving Complete Genome Sequences

The field of genome sequence alignment is moving very quickly. Some useful references in this area are:

Brudno M, Malde S, Poliakov A et al. (2003) Glocal alignment: finding rearrangements during alignment. *Bioinformatics* 19(suppl. 1), i54–i62. *SLAGAN.*

Delcher AL, Kasif S, Fleischmann RD et al. (1999) Alignment of whole genomes. *Nucleic Acids Res.* 27, 2369–2376. *MUMmer.*

Delcher AL, Phillippy A, Carlton J & Salzberg SL (2002) Fast algorithms for large-scale genome alignment and comparison. *Nucleic Acids Res.* 30, 2478–2483. *MUMmer 2.*

Kent WJ (2002) BLAT—the BLAST-like alignment tool. *Genome Res.* 12, 656–664.

Ma B, Tromp J & Li M (2002) PatternHunter: faster and more sensitive homology search. *Bioinformatics* 18, 440–445.

Ning Z, Cox AJ & Mullikin JC (2001) SSAHA: A fast search method for large DNA databases. *Genome Res.* 11, 1725–1729.

Schwartz S, Kent WJ, Smit A et al. (2003) Human–mouse alignments with BLASTZ. *Genome Res.* 13, 103–107.