7

Rapid Alignment Methods: FASTA and BLAST

7.1 The Biological Problem

In the last chapter, we indicated how alignment could be performed rigorously and some of the reasons for performing it. In this chapter, we consider the practicalities of the alignment process, and we demonstrate how it can be speeded up. The need for accelerated methods of alignment is connected with the potentially large number of possible alignments between two sequences (Section 6.5) and with the very large sizes of the databases that must be searched. Why is it necessary to search large databases?

Remember that alignment involves a **query** or **target** sequence and a **search space**. The query sequence typically comes from the organism that is under investigation. The investigator will have obtained the sequence of a portion of the genome and usually seeks information about its possible function either by direct experimentation or by comparing this sequence with related sequences in other organisms. Direct experimentation on a gene of interest (query) in an arbitrary organism may be difficult for a number of reasons. For example, some organisms are difficult to grow in the laboratory (such as certain types of marine bacteria). Other organisms can be grown but may have little genetic or biochemical data (such as the nine-banded armadillos, which can serve as an animal model for leprosy). Or, there may be organisms that are experimentally refractory (we can't perform arbitrary genetic crosses with *Homo sapiens*, for example) or expensive to work with (such as chimpanzees, which are endangered, expensive, and have relatively long generation times).

A solution to this type of problem is to seek comparisons with genes from a number of **model organisms**—organisms chosen for intensive genomic, genetic, or biochemical studies (Section 1.1). Examples of traditional model organisms are *Escherichia coli* (a bacterium), *Saccharomyces cerevisiae* (baker's yeast), *Caenorhabditis elegans* (a nematode "worm"), *Drosophila melanogaster* (fruit fly), *Arabidopsis thaliana* (a flowering plant—"mustard weed"), and *Mus musculus* (the common mouse). Others are being added to

the list as sequencing projects expand. Because of extensive genetic and biochemical study over several decades, many of the genes, gene products, and functions of gene products are known for these model organisms. Because of the evolutionary relationships among organisms, we ordinarily expect that a gene from other experimental organisms may have homologs in the genomes of one or more model organisms. For example, the homeotic genes that act during the development of human embryos have homologs with *Drosophila* homeotic genes, and in some cases these genes have the same functions. The important point is that a target gene is likely to have a function similar or related to functions of a homolog in one or more model organisms. This means that we can (judiciously) attach to the target gene part of the functional annotation associated with homologs in model organisms.

The problem is to search in protein or DNA sequence databases for sequences that match (to a significant degree) the sequence of interest. These databases contain many entries and large numbers of letters. For example, at the time this book was written, there were over 2 million nonredundant entries accessible using the BLAST server at NCBI (Appendix B), and these contained over 10^{10} letters. In recent years, the amount of DNA sequence in databases has been growing exponentially. From the discussion in the previous chapter, we can readily see that the rigorous alignment method described in the previous chapter is too demanding in memory and computation time for routine searching of large databases: the time to compute an alignment between a string of length n and a string of length m was seen to be proportional to $n \times m$. We need something faster.

So far, we have been talking about performing an alignment between a single query sequence and sequences in databases. What happens if we are performing a whole-genome shotgun sequence assembly of a eukaryotic genome? We'll present more about shotgun sequencing later, but for now we need to know that typically the sequencing includes "reads" of about 500 bp from both ends of each insert in a small plasmid library (insert size 1–3 kb). Typically, enough clones are generated to cover the genome 5× or more. So for a mammal having 3×10^9 bp in its genome, 1× coverage by plasmids with 3 kb inserts would involve 10^6 clones, and 5× coverage would involve 5×10^6 clones. With two sequence reads per clone, the total number of sequence reads is 10^7. To look for overlaps during sequence assembly, every read would, in principle, need to be compared with (aligned with) every other read. For N reads, there are $N(N-1)/2$ pairwise comparisons. This means that there are 5×10^{13} comparisons to perform, each of which would require $4 \times (500)^2$ computations if done by dynamic programming. In this case, rapid methods are necessary because of the very large numbers of comparisons that must be made (in principle, any sequence "read" against every other sequence read and its complement).

7.2 Search Strategies

One way to speed up sequence comparison is by reducing the number of sequences to which any candidate sequence must be compared. This can be done by restricting the search for a particular matching sequence to "likely" sequence entries. The logic of the overall process is easily understood if we visualize each sequence to be analyzed as a book in an uncataloged library. Given a book like this one, how could we tell what books are similar just based on word content? The present book has words such as "probabilistic," "genome," "statistics," "composition," and "distribution." If we picked up a book at random from an uncataloged library and did not find these words (as might be the case if we had picked books by Jane Austen or Moses Maimonides), we would know immediately that there is no need to search further in *Sense and Sensibility* or *The Guide of the Perplexed* for help in computational biology.

We can reduce the search space by analyzing word content (see Section 3.6). Suppose that we have the query string I indicated below:

I : `TGATGATGAAGACATCAG`

This can be broken down into its constituent set of overlapping k-tuples. For $k = 8$, this set is

```
TGATGATG
 GATGATGA
  ATGATGAA
   TGATGAAG
        .
        .
        .
          GACATCAG
```

If a string is of length n, then there are $n - k + 1$ k-tuples that are produced from the string. If we are comparing string I to another string J (similarly broken down into words), the absence of any one of these words is sufficient to indicate that the strings are not identical. If I and J do not have at least some words in common, then we can decide that the strings are not similar.

We already know that when $\mathbb{P}(\mathtt{A}) = \mathbb{P}(\mathtt{C}) = \mathbb{P}(\mathtt{G}) = \mathbb{P}(\mathtt{T}) = 0.25$, the probability that an octamer beginning at any position in string J will correspond to a particular octamer in the list above is $1/4^8$. Provided that J is short, this is not very probable, but if J is long, then it is quite likely that one of the eight-letter words in I can be found in J by chance. Therefore, the appearance of a subset of these words is a *necessary but not sufficient condition* for declaring that I and J have at least some sequence similarity.

7.2.1 Word Lists and Comparison by Content

Rather than scanning each sequence for each k-word, there is a way to collect the k-word information in a set of lists. A list will be a row of a table, where the table has 4^k rows, each of which corresponds to one k-word. For example, with $k = 2$ and the sequences below,

$$\begin{aligned} \mathrm{J} &= \mathtt{C\,C\,A\,T\,C\,G\,C\,C\,A\,T\,C\,G} \\ \mathrm{I} &= \mathtt{G\,C\,A\,T\,C\,G\,G\,C} \end{aligned}$$

we obtain the word lists shown in Table 7.2. Thinking of the rows as k-words, we denote the list of positions in the row corresponding to the word w as $L_w(\mathrm{J})$ (e.g., with $w = \mathtt{CG}$, $L_{\mathtt{CG}}(\mathrm{J}) = \{5, 11\}$). These tables are sparse, since the sequences are short, but serve to illustrate our methods. They can be constructed in a time proportional to the sum of the sequence lengths.

One approach to speeding up comparison is to limit detailed comparisons only to those sequences that share enough "content," by which we mean k-letter words in common. The statistic that counts k-words in common is

$$\sum_{i=1}^{n-k+1} \sum_{j=1}^{m-k+1} X_{i,j},$$

where $X_{i,j} = 1$ if $\mathrm{I}_i\mathrm{I}_{i+1}\ldots\mathrm{I}_{i+k-1} = \mathrm{J}_j\mathrm{J}_{j+1}\ldots\mathrm{J}_{i+k-1}$ and 0 otherwise. The computation time is proportional to $n \times m$, the product of the sequence lengths. To improve this, note that for each w in I, there are $\#L_w(\mathrm{J})$ occurrences in J. So the sum above is equal to

$$\sum_w (\#L_w(\mathrm{I})) \times (\#L_w(\mathrm{J}))\,.$$

This equality is a restatement of the relationship between multiplication and addition(!). This second computation is much easier. First we find the frequency of k-letter words in each sequence. This is accomplished by scanning each sequence (of lengths n and m). Then the word frequencies are multiplied and added. Therefore, the total time is proportional to $4^k + n + m$. For our sequence of numbers of 2-word matches, the statistic above is

$$\begin{aligned} &0^2 + 0^2 + 0^2 + 2 \times 1 + 2 \times 1 + 2 \times 0 + 2 \times 1 + 0^2 + 0^2 + 1 \times 2 + 0 \times 1 \\ &\quad +0^2 + 0^2 + 2 \times 1 + 0^2 + 0^2 = 10 \end{aligned}$$

If 10 is above a threshold that we specify, then a full sequence comparison can be performed. (Low thresholds require more comparisons than high thresholds.) This method is quite fast, but the comparison totally ignores the relative positions of the k-words in the sequence. A more sensitive method would be useful.

7.2.2 Binary Searches

Suppose that I and J contain the k-words listed in Table 7.1. How do we find the first word in list I, `TGAT`, within list J? In this example, we can find the matches by inspection. But what would we do if the lists were 500 entries long, and composed of words of $k = 10$? Rather than just scanning down a list from the top, a better way to find matching entries is a **binary search**. Since list J (of length m) is stored in a computer, we can extract the entry number $m/2$, which in this example is entry 16, `GACA`. Then we proceed as follows:

Step 1: Does `TGAT` occur after entry 16 in the alphabetically sorted list? Since it occurs after entry 16, we don't need to look at the first half of the list.

Step 2: In the second half of the list, does `TGAT` occur after the entry at position $m/2 + m/4$? This is entry 24, `TCGA`. `TGAT` occurs after this entry, so we have now eliminated the need to search in the first 3/4 of the list after only two comparisons.

Step 3: Does `TGAT` occur after entry 24 but before entry 29? (We have split the last 1/4 of the list into two $m/8$ segments.) Since it is before 29 and after 24, we only examine the four remaining entries.

Steps 4 and 5: Two more similar steps are needed to "zero in" on entry 25.

Had we gone through the whole list, 25 steps would have been required to find the word. With the binary search, we used only five steps. This process is analogous to finding a word in a dictionary by successively splitting the remaining pages in half until we find the page containing our word. (Try it! We found the page containing "quiescent" after ten splits of pages in a dictionary having 864 pages.)

In general, if we are searching a list of length m starting at the top and going item by item, on average we will need to search half the list before we find the matching word (if it is present). If we perform a binary search as above, we will need only $\log_2(m)$ steps in our search. This is because $m = 2^{\log_2(m)}$, and we can think of all positions in our list of length m as having been generated by $\log_2(m)$ doublings of an initial position. In the example above, $32 = 2^5$, so we should find any entry after five binary steps. In the dictionary experiment, finding the entry should have required ten steps (nine-letter word, $2^9 = 512$, and $2^{10} = 1024$—nine "splits" are not enough since $864 > 512$).

7.2.3 Rare Words and Sequence Similarity

For the method described in Section 7.2.1, if k is large the table size can be enormous, and it will be mostly empty. For large k, another method for detecting sequence similarity is to put the k-words in an ordered list.

To find k-word matches between I and J, first break I down into a list of $n - k + 1$ k-words and J into a list of $m - k + 1$ k-words. Then put the words

Table 7.1. Ordered word lists of query sequence I and sequence J to which it is to be compared. Numbers beside each 4-word indicate the position of each word in the list. Binary searches reduce the search space by half for each iteration, checking whether the search word from I is in the remaining first or second half.

I			
TGAT			
GATG			
ATGA			
...			
J			
AAAT	1	GATG	17
AATC	2	GATT	18
AATG	3	GGAT	19
ACAA	4	GGGA	20
ATCC	5	GTCG	21
ATGA	6	TCAC	22
ATGT	7	TCCG	23
ATTT	8	TCGA	24
CAAA	9	TGAT	25
CCGA	10	TGGG	26
CGAA	11	TGTC	27
CGAC	12	TTGG	28
CGTT	13	TTTA	29
CTTT	14	TTTC	30
GAAT	15	TTTG	31
GACA	16	TTTT	32

in each list in order, from AA...A to TT...T. This takes time $n\log(n)$ and $m\log(m)$ by standard methods which are routinely available but too advanced to present here. Let's index the list by $(W(i), Pw(i)), i = 1, \ldots, n-k+1$ and $(V(j), Pv(j)), j = 1, \ldots, m-k+1$, where, for example, $W(i)$ is the ith word in the ordered list and $Pw(i)$ is the position that word had in I.

We discover k-word matches by the following algorithm which basically merges two ordered lists into one long ordered list. Start at the beginning of one list. So long as that element is smaller than the beginning of the second list continue to add elements from that list. When this is no longer the case, switch to the other list. Proceed until reaching the end of one of the lists. During this process we will discover all k-words that are equal between the lists, along with producing the merged ordered list. Because the positions in the original sequences are carried along with each k-word, we will know the location of the matches as well. Obviously matches longer than length k will be observed as successive overlapping matches.

7.3 Looking for Regions of Similarity Using FASTA

FASTA (Pearson and Lipman, 1988) is a rapid alignment approach that combines methods to reduce the search space (it depends on k-tuples) and Smith-Waterman local sequence alignment, as described in the previous chapter. As an introduction to the rationale of the FASTA method, we begin by describing **dot matrix** plots, which are a very basic and simple way of visualizing regions of sequence similarity between two different strings. We have already alluded to them in Section 5.4.

7.3.1 Dot Matrix Comparisons

Dot matrix comparisons are a special type of alignment matrix with positions i in sequence I corresponding to rows, positions j in sequence J corresponding to columns, and sequence identities indicated by placing a dot at matrix element (j, i) if the word or letter at J_j is identical to the word or letter at I_i. An example for two DNA strings is shown in Fig. 7.1A. In this example, the string `CATCG` in I appears twice in J, and these regions of local sequence similarity appear as two diagonal arrangements of dots: diagonals represent regions having sequence similarity.

When I and J are DNA sequences and are short, the patterns of this type are relatively easy to see. When I and J are DNA sequences and very long, there will be many dots in the matrix since, for any letter at position j in J, the probability of having a dot at any position i in I will equal the frequency of the letter J_j in the DNA. For 50% `A+T`, this means that on average 1/4 of the matrix elements will have a dot. When I and J are proteins, dots in the matrix elements record matches between amino acid residues at each particular pair of positions. Since there are 20 different amino acids, if the amino acid frequencies were identical, the probability of having a dot at any particular position would be 1/20.

7.3.2 FASTA: Rationale

The rationale for FASTA (Wilbur and Lipman, 1983) can be visualized by considering what happens to a dot matrix plot when we record matches of k-tuples ($k > 1$) instead of recording matches of single letters (Fig. 7.1 B and C). We again place entries in the alignment matrix, except this time we only make entries at the first position of each dinucleotide or trinucleotide (k-tuple matches having $k = 2$ (plotted numerals 2) or $k = 3$ (plotted numerals 3). Notice how the number of matrix entries is reduced as k increases. By looking for words with $k > 1$, we find that we can ignore most of the alignment matrix since the absence of shared words means that subsequences don't match well. There is no need to examine areas of the alignment matrix where there are

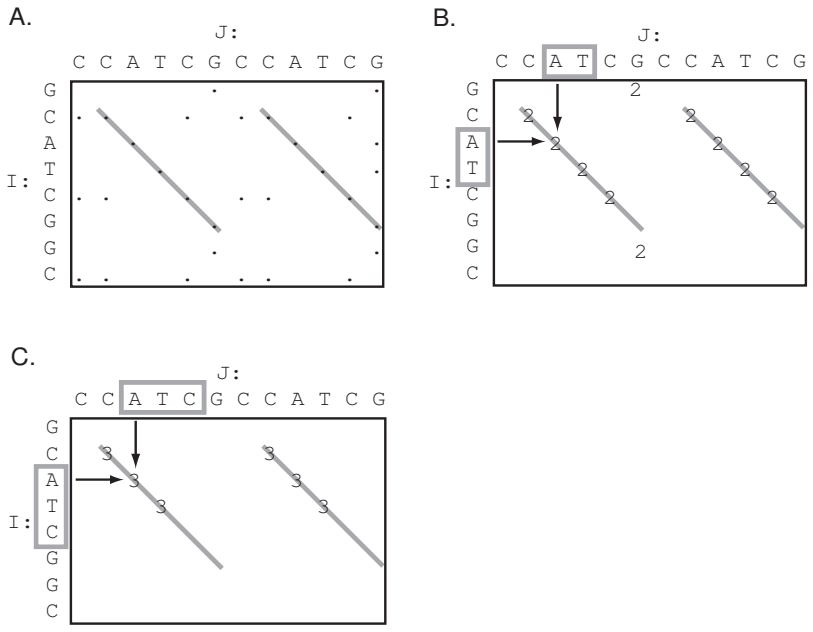


Fig. 7.1. Dot matrix plots showing regions of sequence similarity between two strings (diagonal lines). Panels A, B, and C are plots for k-tuples $k = 1$, 2, and 3, respectively. Typical k-tuples used for plotting a particular element are indicated by boxes.

no word matches. Instead, we only need to focus on the areas around any diagonals.

Our task is now to compute efficiently diagonal sums of scores, S_l, for diagonals such as those in Fig. 7.1B. (The method for forming these scores is explained below.) Consider again the two strings I and J that we used in Section 7.2.1. There are $7 \times 11 = 77$ potential 2-matches, but in reality there are ten 2-matches with four nonzero diagonal sums. We index diagonals by the offset, $l = i - j$. In this notation, the nonzero diagonal sums are $S_{+1} = 1$, $S_0 = 4$, $S_{-5} = 1$, and $S_{-6} = 4$. It is possible to find these sums in time proportional to $n + m + \#\{k\text{-word matches}\}$. Here is how this is done.

Make a k-word list for J. In our case, this is the list for J in Table 7.2. Then initialize all row sums to 0:

ℓ	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1	0	1	2	3	4	5	6
S_ℓ	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Next proceed with the 2-words of I, beginning with $i = 1$, `GC`. Looking in the list for J, we see that $L_{\texttt{GC}}(\text{ J}) = \{6\}$, so we know that at $l = 1 - 6 = -5$ there is a 2-word match of `GC`. Therefore, we replace $S_{-5} = 0$ by $S_{-5} = 0 + 1 = 1$.

Table 7.2. k-word lists for J = CCATCGCCATCG and I = GCATCGGC, $k = 2$.

J		I	
AA		AA	
AC		AC	
AG		AG	
AT	3, 9	AT	3
CA	2, 8	CA	2
CC	1, 7	CC	
CG	5, 11	CG	5
CT		CT	
GA		GA	
GC	6	GC	1, 7
GG		GG	6
GT		GT	
TA		TA	
TC	4, 10	TC	4
TG		TG	
TT		TT	

Next, for $i = 2$, CA, we have $L_{\texttt{CA}}$(J)={2,8}. Therefore replace $S_{2-2} = S_0 = 0$ by $S_0 = 0+1$, and replace $S_{2-8} = S_{-6} = 0$ by $S_{-6} = 0+1$. These operations, and the operations for all of the rest of the 2-words in I, are summarized below. Note that for each successive step, the then current score at S_i is employed: S_0 was set to 1 in step 2, so 1 is incremented by 1 in step 3.

$i = 1$, GC $L_{\texttt{GC}}$(J)={6} $l = 1 - 6 = -5$
$S_{-5} = 0 \rightarrow S_{-5} = 0 + 1 = 1$

$i = 2$, CA $L_{\texttt{CA}}$(J)={2,8} $l = 2 - 2 = 0$
$S_0 = 0 \rightarrow S_0 = 0 + 1$, and
$l = 2 - 8 = -6$
$S_{-6} = 0 \rightarrow S_{-6} = 0 + 1$

$i = 3$, AT $L_{\texttt{AT}}$(J)={3,9} $l = 3 - 3 = 0$
$S_0 = 1 \rightarrow S_0 = 1 + 1 = 2$
$l = 3 - 9 = -6$
$S_{-6} = 1 \rightarrow S_{-6} = 1 + 1 = 2$

$i = 4$, TC $L_{\texttt{TC}}$(J)={4,10} $l = 4 - 4 = 0$
$S_0 = 2 \rightarrow S_0 = 2 + 1 = 3$
$l = 4 - 10 = -6$
$S_{-6} = 2 \rightarrow S_{-6} = 2 + 1 = 3$

$i = 5$, CG $L_{\texttt{CG}}$(J)={5,11} $l = 5 - 5 = 0$
$S_0 = 3 \rightarrow S_0 = 3 + 1 = 4$
$l = 5 - 11 = -6$
$S_{-6} = 3 \rightarrow S_{-6} = 3 + 1 = 4$

$i = 6$, GG	$L_{\mathtt{GG}}(\mathrm{J})=\{\varnothing\}$	$L_{\mathtt{GG}}(\mathrm{J})$ is the empty set: no sums are increased
$i = 7$, GC	$L_{\mathtt{GC}}(J) = \{6\}$	$l = 7 - 6 = 1$ $S_1 = 0 \rightarrow S_1 = 0 + 1 = 1$

The final result for scores of the diagonals at various offsets is

ℓ	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1	0	1	2	3	4	5	6
S_ℓ	0	0	0	0	4	1	0	0	0	0	4	1	0	0	0	0	0

Notice that we only performed additions when there were 2-word matches. The algorithm that we employed is indicated in pseudocode in Computational Example 7.1.

Computational Example 7.1: Pseudocode for diagonal sums of scores

Set $S_\ell \leftarrow 0$ for all $1 - m \leq \ell \leq n - 1$
Compute $L_w(\mathrm{J})$ for all words w
for $i = 1$ to $n - k - 1$
    $w \leftarrow I_i I_{i+1} \ldots I_{i+k-1}$
    for $j \in L_w(\mathrm{J})$
        $\ell \leftarrow i - j$
        $S_\ell \leftarrow S_\ell + 1$
    end
end

It is possible to find local alignments using a gap length penalty of $-gx$ for a gap of length x along a diagonal. Let A_ℓ be the local alignment score and S_l be the maximum of all of the A_l's on the diagonal. Then the scoring is done as follows, after initializing $A_\ell \leftarrow 0, S_\ell \leftarrow 0$, for each element (i, j) along the diagonal $\ell = i - j$, beginning at $i = 1$:

$$A_\ell \leftarrow \max \left\{ \begin{array}{c} A_\ell + 1 \text{ if } a_i = b_{j+l} \\ A_\ell - g \text{ if } a_i \neq b_{j+l} \\ 0 \end{array} \right\}$$

$$S_\ell \leftarrow \max\{S_\ell,\ A_\ell\}$$

Five steps are involved in FASTA:

1. Use the look-up table to identify k-tuple identities between I and J.
2. Score diagonals containing k-tuple matches, and identify the ten best diagonals in the search space.
3. Rescore these diagonals using an appropriate scoring matrix (especially critical for proteins), and identify the subregions with the highest score (initial regions).

4. Join the initial regions with the aid of appropriate joining or gap penalties for short alignments on offset diagonals.
5. Perform dynamic programming alignment within a band surrounding the resulting alignment from step 4.

To implement the first step, we pass through I once and create a table of the positions i for each possible word of predetermined size k. Then we pass through the search space J once, and for each k-tuple starting at successive positions j, "look up" in the table the corresponding positions for that k-tuple in I. Record the i, j pairs for which matches are found. We have already illustrated this process for 2-words in Section 7.2.1. The i, j pairs define where potential diagonals can be found in the alignment matrix.

FASTA step 2 is identification of high-scoring diagonals. If I has n letters and J has m letters, there are $n + m - 1$ diagonals. (Think of starting in the lower left-hand corner, drawing successive diagonals all the way up to the top of the column representing J (m diagonals). Then continue rightward, drawing diagonals through all positions in I (n diagonals). Since you will have counted the first position twice, you need to subtract 1.) To score the diagonals, calculate the number of k-tuple matches for every diagonal having at least one k-tuple. Scoring may take into account distances between matching k-tuples along the diagonal. Note that the number of diagonals that needs to be scored will be much less than the number of all possible diagonals. Identify the significant diagonals as those having significantly more k-tuple matches than the mean number of k-tuple matches. This means, of course, that we should set a threshold, such as two standard deviations above the mean. For example, if the mean number of 6-tuples is 5 ± 1, then with a threshold of two standard deviations, you might consider diagonals having seven or more 6-tuple matches as significant. Take the top ten significant diagonals.

In step 3, we rescore the diagonals using a scoring table and allowing identities shorter than k-tuple length. We retain the subregions with the highest scores. The need for this rescoring is illustrated by the two examples below.

```
I:  C C A T C G C C A T C G        (Number 4-tuple matches: 0)
J:  C C A A C G C A A T C A

I': C C A T C G C C A T C G
J': A C A T C A A A T A A A
```

In the first case, the placement of mismatches spaced by three letters means that there are no 4-tuple matches, even though the sequences are 75% identical. The second pair shows one 4-tuple match, but the two sequences are only 33% identical. Rescoring reveals sequence similarity not detected because of the arbitrary demand for uninterrupted identities of length k.

Step 4 is joining together appropriate diagonals that may be *offset* from each other, as might occur if there were a gap in the alignment (i.e., vertical or horizontal displacements in the alignment matrix, as described in the previous

chapter). Diagonal d_l is the one having k-tuple matches at positions i in string I and j in string J such that $i - j = l$. As described earlier in this chapter, $l = i - j$ is called the **offset**. Offsets are explained pictorially in Fig. 7.2. Alignments are extended by joining offset diagonals if the result is an extended aligned region having a higher alignment score, taking into account appropriate joining (gap) penalties.

Step 5 is to perform a rigorous Smith-Waterman local alignment. This can be restricted to a comparatively narrow window extending $+w$ to the right and $-w$ to the left of the positions included within the highest-scoring diagonal (see Fig. 7.3D).

7.4 BLAST

The most used database search programs are BLAST and its descendants. BLAST is modestly named for *B*asic *L*ocal *A*lignment *S*earch *T*ool, and it was introduced in 1990 (Altschul et al., 1990). Whereas FASTA speeds up the search by filtering the k-word matches, BLAST employs a quite different

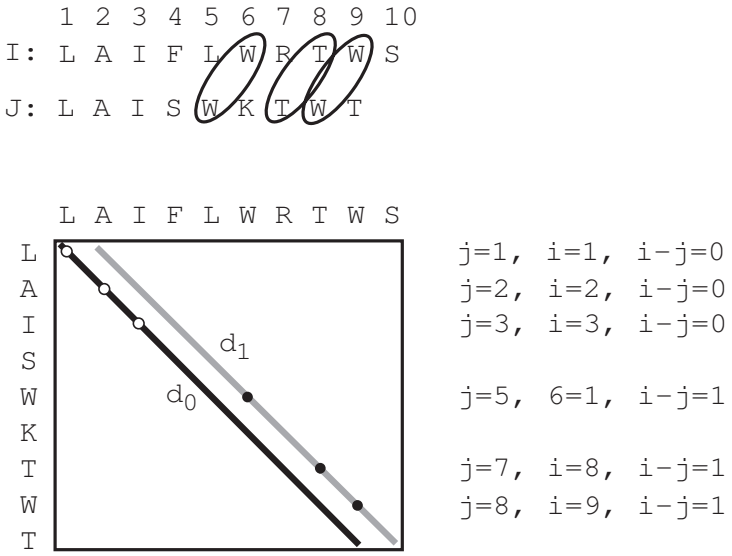


Fig. 7.2. Illustration of offset diagonals. The first three letters for the alignment between I and J as drawn have no offset. Corresponding diagonal d_0 is drawn in black. Farther down I and J, there are additional matches that are offset from each other (residues enclosed by ellipses). These define another diagonal, d_1, that is offset from the first one (grey line). Such offsets may indicate indels, suggesting that the local alignments represented by the two diagonals should be joined to form a longer alignment.

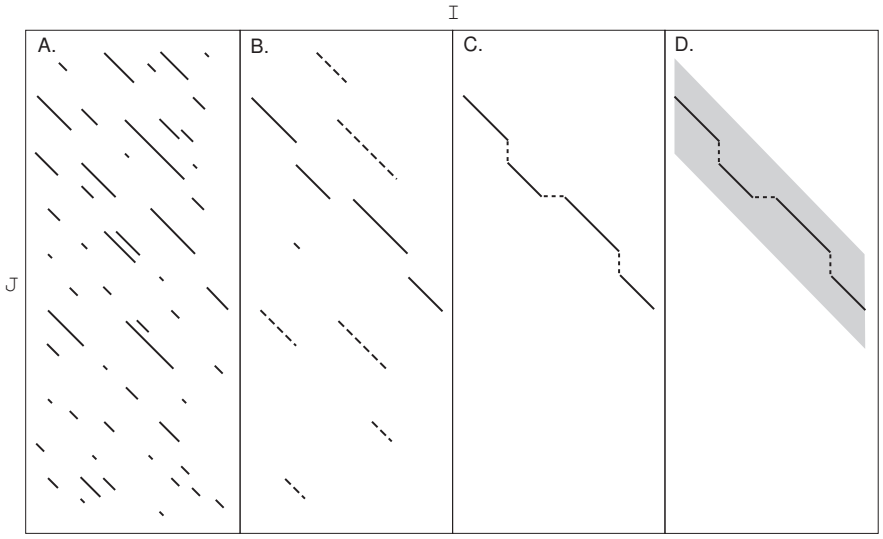


Fig. 7.3. Illustration of FASTA steps 2–5. Panel A: Identify diagonals sharing k-tuples (step 2). Panel B: Rescore to generate initial regions (step 3). Panel C: Join initial regions to give the combination having maximum score (step 4). Panel D: Perform dynamic programming in a "window space" or "band" centered around the highest-scoring initial region (step 5).

strategy. This can be summarized in two parts: the method for finding local alignments between a query sequence and a sequence in a database, and the method for producing p-values and a rank ordering of the local alignments according to their p-values. High-scoring local alignments are called "*high scoring segment pairs*," or HSPs. The output of BLAST is a list of HSPs together with a measure of the probability that such matches would occur by chance.

7.4.1 Anatomy of BLAST: Finding Local Matches

First, the query sequence is used as a template to construct a set of subsequences of length w that can score at least T when compared with the query. A substitution matrix, containing **neighborhood sequences**, is used in the comparison. Then the database is searched for each of these neighborhood sequences. This can be done very rapidly because the search is for an exact match, just as our word processor performs exact searches. We have not developed such sophisticated tools here, but such a search can be performed in time proportional to the sum of the lengths of the sequence and the database.

Let's return to the idea of using the query sequence to generate the neighborhood sequences. We will employ the same query sequence I and search space J that we used previously (Section 7.2.1):

$$\mathrm{J} = \mathtt{C\,C\,A\,T\,C\,G\,C\,C\,A\,T\,C\,G}$$
$$\mathrm{I} = \mathtt{G\,C\,A\,T\,C\,G\,G\,C}$$

We use subsequences of length $k = 5$. For the *neighborhood size*, we use all 1-mismatch sequences, which would result from scoring matches 1, mismatches 0, and the test value (threshold) $T = 4$. For sequences of length $k = 5$ in the neighborhood of GCATC with $T = 4$ (excluding exact matches), we have:

$$\left\{\begin{matrix}\mathtt{A}\\ \mathtt{C}\\ \mathtt{T}\end{matrix}\right\}\mathtt{CATC},\quad \mathtt{G}\left\{\begin{matrix}\mathtt{A}\\ \mathtt{G}\\ \mathtt{T}\end{matrix}\right\}\mathtt{ATC},\quad \mathtt{GC}\left\{\begin{matrix}\mathtt{C}\\ \mathtt{G}\\ \mathtt{T}\end{matrix}\right\}\mathtt{TC},\quad \mathtt{GCA}\left\{\begin{matrix}\mathtt{A}\\ \mathtt{C}\\ \mathtt{G}\end{matrix}\right\}\mathtt{C},\quad \mathtt{GCAT}\left\{\begin{matrix}\mathtt{A}\\ \mathtt{G}\\ \mathtt{T}\end{matrix}\right\}$$

Each of these terms represents three sequences, so that in total there are $1 + (3 \times 5) = 16$ exact matches to search for in J. For the three other 5-word patterns in I (CATCG, ATCGG, and TCGGC), there are also 16 exact 5-words, for a total of $4 \times 16 = 64$ 5-word patterns to locate in J.

A **hit** is defined as an instance in the search space (database) of a k-word match, within threshold T, of a k-word in the query sequence. There are several hits in I to sequence J. They are

5-words in I	5-words in J	J position(s)	Score
CATCG	CATCG	2, 8	5
GCATC	CCATC	1	4
ATCGG	ATCGC	3	4
TCGGC	TCGCC	4	4

In actual practice, the hits correspond to a tiny fraction of the entire search space. The next step is to extend the alignment starting from these "seed" hits. Starting from any seed hit, this extension includes successive positions, with corresponding increments to the alignment score. This is continued until the alignment score falls below the maximum score attained up to that point by a specified amount. Later, improved versions of BLAST only examine diagonals having *two* nonoverlapping hits no more than a distance A residues away from each other, and then extend the alignment along those diagonals. Unlike the earlier version of BLAST, gaps can be accommodated in the later versions.

With the original version of BLAST, over 90% of the computation time was employed in producing the ungapped extensions from the hits. This is because the initial step of identifying the seed hits was effective in making this alignment tool very fast. Later versions of BLAST require the same amount of time to find the seed hits and have reduced the time required for the ungapped extensions considerably. Even with the additional capabilities for allowing gaps in the alignment, the newer versions of BLAST run about three times faster than the original version (Altschul et al., 1997).

7.4.2 Anatomy of BLAST: Scores

The second aspect of a BLAST analysis is to rank-order the sequences found by p-values. If the database is $\mathcal{D}$ and a sequence X scores $S(\mathcal{D}, \mathrm{X}) = s$ against

the database, the p-value is $\mathbb{P}(S(\mathcal{D}, \mathrm{Y}) \geq s)$, where Y is a random sequence. The smaller the p-value, the greater the "surprise" and hence the greater the belief that something real has been discovered. A p-value of 0.1 means that with a collection of query sequences picked at random, in 1/10 of the instances a score that large or larger would be discovered. A p-value of 10^{-6} means that only once in a million instances would a score of that size appear by chance alone.

There is a nice way of computing BLAST p-values that has a solid mathematical basis. Although a rigorous treatment is far beyond the scope of this book, an intuitive and accurate account is quite straightforward. In a sequence-matching problem where the score is 1 for identical letters and $-\infty$ otherwise (i.e., no mismatches and no indels), the best local alignment score is equal to the longest exact matching between the sequences. In our $n \times m$ alignment matrix, there are (approximately) $n \times m$ places to begin an alignment. Generally, an optimal alignment begins with a mismatch, and we are interested in those that extend at least t matching (identical) letters. We set

$$p = \mathbb{P}(\text{two random letters are equal}).$$

The event of a mismatch followed by t identities has probability $(1-p)p^t$. There are $n \times m$ places to begin this event, so the mean or expected number of local alignments of at least length t is $nm(1-p)p^t$. Obviously, we want this to be a rare event that is well-modeled by the Poisson distribution (see Chapter 3) with mean

$$\lambda = nm(1-p)p^t,$$

so

$$\begin{aligned}\mathbb{P}(\text{there is local alignment } t \text{ or longer}) &\approx 1 - \mathbb{P}(\text{no such event}) \\ &= 1 - \mathrm{e}^{-\lambda} \\ &= 1 - \exp(-nm(1-p)p^t).\end{aligned}$$

This equation is of the same form used in BLAST, which estimates

$$\mathbb{P}(S(\mathcal{D}, \mathrm{Y}) \geq s) \approx 1 - \exp(-nm\gamma\xi^t),$$

where $\gamma > 0$ and $0 < \xi < 1$. There are conditions for the validity of this formula, in which γ and ξ are estimated parameters, but this is the idea! (In BLAST output, the last quantity is called an E-value.)

The take-home message of this discussion is that the probability of finding an HSP by chance using a random query sequence Y in database $\mathcal{D}$ is approximately equal to E.

7.5 Scoring Matrices for Protein Sequences

The alignment scores obviously depend on the scoring matrices. We discussed scoring matrices for DNA in Section 6.6. Now we seek a method to score alignments between proteins X and Y such as

$$\begin{aligned} \mathrm{X} &= \ldots \mathtt{NVSDVNLNK} \ldots \\ \mathrm{Y} &= \ldots \mathtt{NASNLSLSK} \ldots \end{aligned}$$

We need to assign scores for the alignment of residues at any particular position, such as the one underlined above. In other words, we need to find the probability p_{ab} of "matching" amino acid a with amino acid b. The values of the p_{ab} will differ depending on the identities of a and b. For example, the score at the position indicated in this example should take into account the hydrophobic character shared by valine and leucine, which conserves the properties (and possibly the function) of the two proteins.

7.5.1 Rationale for Scoring: Statement of the Problem

We are given two sequences, $\mathrm{A} = a_1 a_2 \ldots a_n$ and $\mathrm{B} = b_1 b_2 \ldots b_n$, of equal length. The alignment will be over the entire set of letters in each sequence (i.e., a global alignment). No gaps are employed in our simple illustration, although as we have seen, FASTA and BLAST do allow gaps. We seek to devise a scoring scheme based on the probabilities of matching amino acid a with amino acid b. We use an approach that we will also employ later when describing signals in DNA (Section 9.2.1). We take the ratio of two probabilities: the probability that the sequence strings match (match model $\mathcal{M}$) and the probability that the sequence strings were chosen at random (random model $\mathcal{R}$). The probability of having X and Y given the random model is

$$\mathbb{P}(\mathrm{A}, \mathrm{B} \mid \mathcal{R}) = \prod_i q_{a_i} \prod_i q_{b_i},$$

where q_{x_i} is the probability of occurrence of the amino acid of the type x_i in a collection of proteins, irrespective of position. The model above assumes that the identity of x_i is independent of the identity of x_{i-1}.

What is the probability of having these two sequences according to the "match" model? By "match" we recognize explicitly that amino acids at corresponding positions may have *degrees of relationship* based upon how much divergence has occurred since the two strings evolved from a common ancestor. In other words, we won't simply be assigning a single score value for all matches and identical penalties for mismatches. We define p_{ab} as the probability of finding an amino acid of type a aligned with an amino acid of type b given that they have both evolved from an ancestor who had c at that position ($c = a, b$, or something else). This probability is

$$\mathbb{P}(\mathrm{A}, \mathrm{B} \mid \mathcal{M}) = \prod_i p_{a_i b_i}.$$

The score S is obtained by taking the ratio of probabilities under the two models—match relative to random sequences. This ratio is

$$\frac{\mathbb{P}(\mathrm{A}, \mathrm{B} \mid \mathcal{M})}{\mathbb{P}(\mathrm{A}, \mathrm{B} \mid \mathcal{R})} = \frac{\prod_i p_{a_i b_i}}{\prod_i q_{a_i} \prod_i q_{b_i}} = \prod_i \left(\frac{p_{a_i b_i}}{q_{a_i} q_{b_i}} \right).$$

We define the score S as

$$S = \log_2\left(\frac{\mathbb{P}(\mathrm{A},\mathrm{B} \mid \mathcal{M})}{\mathbb{P}(\mathrm{A},\mathrm{B} \mid \mathcal{R})}\right) = \sum_{i=1}^{n} \log_2\left(\frac{p_{a_i b_i}}{q_{a_i} q_{b_i}}\right) = \sum_{i=1}^{n} s(a_i, b_i),$$

which indicates that we are adding together scores for aligning individual amino acid pairs a and b:

$$s(a,b) = \log_2\left(\frac{p_{ab}}{q_a q_b}\right).$$

The $p_{a_i b_i}$ are extracted from collections of data, as described in the next section.

7.5.2 Calculating Elements of the Substitution Matrices

What we ultimately wish to find is a **substitution matrix**, whose components are the scaled scores $s(a,b)$ for aligning amino acid a with amino acid b,

$$\begin{array}{cccccccc}
 & \mathtt{A} & & \mathtt{R} & & \mathtt{N} & & \mathtt{D} \quad \cdots \quad \mathtt{V} \\
\mathtt{A} & s(\mathtt{A},\mathtt{A}) & & & & & & \\
\mathtt{R} & s(\mathtt{R},\mathtt{A}) & & s(\mathtt{R},\mathtt{R}) & & & & \\
\mathtt{N} & s(\mathtt{N},\mathtt{A}) & & s(\mathtt{N},\mathtt{R}) & & s(\mathtt{N},\mathtt{N}) & & \\
\mathtt{D} & s(\mathtt{D},\mathtt{A}) & & s(\mathtt{D},\mathtt{R}) & & s(\mathtt{D},\mathtt{N}) & & s(\mathtt{D},\mathtt{D}) \\
\vdots & & & & & & & \\
\mathtt{V} & s(\mathtt{V},\mathtt{A}) & & s(\mathtt{V},\mathtt{R}) & & s(\mathtt{V},\mathtt{N}) & & s(\mathtt{V},\mathtt{D}) \cdots s(\mathtt{V},\mathtt{V})
\end{array}$$

where $s(a,b) = s(b,a)$. (Letters labeling rows and columns are single-letter amino acid codes, listed so that the amino acid *names* are in alphabetical order: A = alanine, R = arginine, etc.)

The first substitution matrix set to be devised was the **PAM** (point accepted mutation) family (e.g., PAM100) (Dayhoff et al., 1978). With PAM100, for example, the p_{ab}s used to obtain the $s(a,b)$ values are calculated so that they correspond to an average of 100 changes per 100 amino acid residues. (Note that there will still be sequence similarity after 100 changes per 100 residues since some residues will not have mutated at all, others will have changed repeatedly, and still other residues will back-mutate to their original identity.) These were evaluated by multiplying together matrices of probabilities, the originals of which depended upon a set of proteins that had diverged by a fixed amount. Now the preferred substitution matrices are the **BLOSUM** set (BLOcks SUbstitution Matrices: Henikoff and Henikoff, 1992). BLOSUM matrices are based on aligned protein sequence blocks without assumptions about mutation *rates*. BLOSUM45 is similar to PAM250. BLOSUM62 (comparable to PAM160) is commonly used (Table 7.3).

Earlier, we developed an equation for calculating the $s(a,b)$, and it requires q_a, q_b, and p_{ab}. It is obvious that the q_a (and q_b) can be obtained just by

Table 7.3. BLOSUM62 scoring matrix for scoring protein alignments. Data are in half-bits. For the meaning of single-letter IUPAC-IUB amino acid symbols, see Appendix C.1 (or `http://www.fruitfly.org/blast/blastFasta.html`). Less commonly used symbols are `*` = translational stop, `B` = `D` or `N`, and `Z` = `E` or `Q`. Data in this table are from `http://www.ncbi.nlm.nih.gov/Class/FieldGuide/BLOSUM62.txt`.

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

counting the number of occurrences of each amino acid type in an appropriate collection of protein sequences, and then dividing by the total number of amino acids represented. But we are still left with the problem of where to obtain the p_{ab}. There are a number of repositories of protein data that are extremely useful for obtaining both types of quantities:

SWISS-PROT (`http://au.expasy.org/sprot/`)
: This is an annotated database of protein sequences, that is minimally redundant (multiple entries for the same sequence are avoided) and heavily cross-indexed with other protein databases. At this time, there are about 154,000 protein sequences representing 57×10^6 letters in this database.

PROSITE (`http://www.expasy.ch/prosite/`)
: This is a database of protein families and signature motifs (characteristic short sequence patterns) that characterize these families. Approximately 1700 families and domains are archived.

BLOCKS (`http://blocks.fhcrc.org/`)
: This is a compilation of the most highly conserved regions for proteins in PROSITE. Here are listed multiply aligned, ungapped, conserved segments characteristic of each protein family. Examples are shown in Fig. 7.4.

```
Block PR00851A

ID XRODRMPGMNTB; BLOCK
AC PR00851A; distance from previous block=(52,131)
DE Xeroderma pigmentosum group B protein signature
BL adapted; width=21; seqs=8; 99.5%=985; strength=1287
XPB_HUMAN|P19447   ( 74)       RPLWVAPDGHIFLEAFSPVYK  54
XPB_MOUSE|P49135   ( 74)       RPLWVAPDGHIFLEAFSPVYK  54
P91579             ( 80)       RPLYAPDGHIFLESFSPVYK   67
XPB_DROME|Q02870   ( 84)       RPLWVAPNGHVFLESFSPVYK  79
RA25_YEAST|Q00578 (131)       PLWISPSDGRIILESFSPLAE 100
Q38861             ( 52)       RPLWACADGRIFLETFSPLYK  71
O13768             ( 90)       PLWINPIDGRIILEAFSPLAE 100
O00835             ( 79)       RPIWVCPDGHIFLETFSAIYK  86
//


Block PR00851B

ID XRODRMPGMNTB; BLOCK
AC PR00851B; distance from previous block=(65,65)
DE Xeroderma pigmentosum group B protein signature
BL adapted; width=20;seqs=8; 99.5%=902; strength=1435
XPB_HUMAN|P19447   ( 160)      TVSYGKVKLVLKHNRYFVES  68
XPB_MOUSE|P49135   ( 160)      TVSYGKVKLVLKHNRYFVES  68
P91579             ( 166)      TQSYGKVKLVLKHNKYYVES  85
XPB_DROME|Q02870   ( 170)      TLSYGKVKLVLKHNKYFIES  77
RA25_YEAST|Q00578 ( 217)      TISYGKVKLVIKHNRYFVET 100
Q38861             ( 138)      TANYGKVKLVLKKNRYFIES  90
O13768             ( 176)      TVSYGKVKLVLKKNRYFIES  72
O00835             ( 165)      TQSYGKVKLVLQKNKYFVES  87
//
```

Fig. 7.4. Examples of sequence blocks from the Blocks database. In this case, two different blocks from the same set of proteins are presented. All proteins are related to a human gene associated with the DNA-repair defect xeroderma pigmentosum (leading to excessive sensitivity to ultraviolet light). Reprinted, with permission, from the Blocks database (`http://blocks.fhcrc.org/`). Copyright 2003 Fred Hutchinson Cancer Research Center. Data for entry PR00851 adapted from the Prints database. (`http://www.bioinf.man.ac.uk/dbrowser/PRINTS`).

7.5.3 How Do We Create the BLOSUM Matrices?

Mechanics for enumerating the occurrences of various types of paired amino acids (as a preliminary to calculation of p_{ab}) are as follows. Each block consists of n aligned sequences each having w residues. For each column in a block (archived in the Blocks database), we count the number of pairwise matches and mismatches for each amino acid type. In column 3 of the block shown below (underlined residues) we find that the number of pairwise matches of L with L is $4 + 3 + 2 + 1 = 10$ or $5(5 - 1)/2$. (Matches of L in a sequence to itself are not counted.)

```
R P L W V A P D ...
R P L W V A P D ...
R P L Y L A P D ...
R P L W V A P N ...
P L W I S P S D ...
R P L W A C A D ...
P L W I N P I D ...
R P I W V C P D ...
```

The enumeration of matches between all leucine (L) residues in *the same column* can be understood by the matrix below, which applies to the indicated column:

```
  L L L L W L W I
L - + + +   +
L   - + +   +
L     - +   +
L       -   +
W           -
L               -
W                   -
I                       -
```

From this we see that the total number of pairwise matches per column is equal to the number of "+" entries in the triangular area above the diagonal. If we take the total number of entries in the matrix (n^2), subtract the number of entries on the diagonal (n), and divide by 2 (to avoid counting the match of L_1 with L_2 as both $\mathrm{L}_1\mathrm{L}_2$ and $\mathrm{L}_2\mathrm{L}_1$), we obtain a total of $n(n-1)/2$ possible pairings. Each block then provides $w \times n(n-1)/2$ possible pairings, and we count the number of pairings of each type for more than 24,000 blocks. We keep a running total of the number of each kind of pairing.

Now make a matrix for the number of occurrences f_{ab} for each pairing of each type:

$$
\begin{array}{c|cccccc}
 & \mathtt{A} & \mathtt{R} & \mathtt{N} & \mathtt{D} & \cdots & \mathtt{V} \\
\mathtt{A} & f_{\mathtt{A},\mathtt{A}} & & & & & \\
\mathtt{R} & f_{\mathtt{R},\mathtt{A}} & f_{\mathtt{R},\mathtt{R}} & & & & \\
\mathtt{N} & f_{\mathtt{N},\mathtt{A}} & f_{\mathtt{N},\mathtt{R}} & f_{\mathtt{N},\mathtt{N}} & & & \\
\mathtt{D} & f_{\mathtt{D},\mathtt{A}} & f_{\mathtt{D},\mathtt{R}} & f_{\mathtt{D},\mathtt{N}} & f_{\mathtt{D},\mathtt{D}} & & \\
\vdots & & & & & & \\
\mathtt{V} & f_{\mathtt{V},\mathtt{A}} & f_{\mathtt{V},\mathtt{R}} & f_{\mathtt{V},\mathtt{N}} & f_{\mathtt{V},\mathtt{D}} & \cdots & f_{\mathtt{V},\mathtt{V}}
\end{array}
$$

We can use these f_{ab} to calculate p_{ab} using the following equation:

$$p_{ab} = f_{ab} \Big/ \sum_{a=1}^{20} \sum_{b=1}^{a} f_{ab}.$$

Notice the limits on the summation in the denominator. Think of the first summation as extending over rows and the second over columns. Since $p_{ab} = p_{ba}$, we are interested in the diagonal together with the terms below it. Therefore, the second summation (performed for each row a) needs only to proceed up to column a, as indicated. The denominator is the total number of pairwise matches, including residue a with itself.

Using the same reasoning as above, the number of terms below the diagonal is $20(20-1)/2 = 190$. The number of terms on the diagonal is 20, so there are 210 distinct p_{ab}s. We get the estimated probability for each amino acid, q_a, based on its frequency of occurrence in the whole collection of blocks:

$$q_a = \sum_{b=1}^{20} \left(f_{ab} \Big/ \sum_{a=1}^{20} \sum_{b=1}^{a} f_{ab} \right).$$

Then we take $s(a,b) = \log_2(p_{ab}/q_a q_b)$. In practice, these values are rounded off and scaled. So, for BLOSUM, the scores are reported in half-bits: $s(a,b)$ [half-bits] $= 2s(a,b)$.

There is one more matter to deal with: How do we "tune" the matrix to the amount of sequence divergence we are expecting in our similarity search? This has to do with how we count pairs in blocks. Consider Fig. 7.5, where the tips of the dendrogram indicate the identity of the amino acid at a particular position in members of a block of eight sequences.

Obviously, there is a group of four sequences that are very similar to each other that have L at that position. Should we count the top four examples as separate individuals for evaluating the f_{ij}? Normally, clustering is performed for the sequence entries in the blocks (clustering will be discussed in Chapter 10), and contributions from sequences that cluster at similarities greater than some specified cutoff (broken line) are averaged. In this case (where all letters are identical), the effect would be that of replacing four Ls with one L prior to counting the number of pairwise matches. Moving the cutoff to lower levels of similarity produces a BLOSUM matrix whose entries correspond to greater amounts of evolutionary separation.

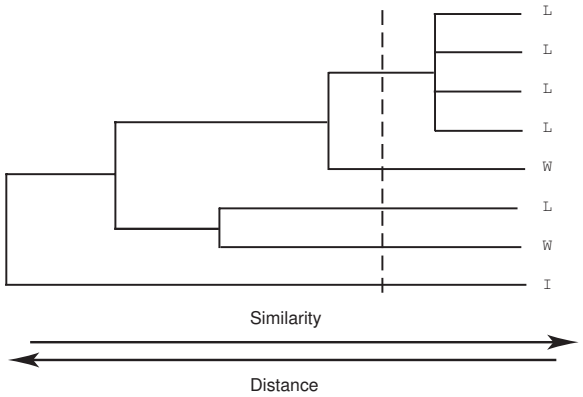


Fig. 7.5. Calculating f_{ab} as a function of evolutionary distance. The dendrogram (branching pattern) is based upon the evolutionary distance between the proteins from which the sequences in a particular block were taken. The single-letter amino acid codes on the right represent those amino acids present at a particular position in the sequences constituting the sequence block. If we are concerned with recently diverged proteins, each of the four `L` residues in the cluster at the top should be counted separately. If the concern is with more distantly related proteins (with distance indicated by the dotted cutoff line), then the cluster of four `L` residues should only be counted as one instance of `L` instead of four.

Computational Example 7.2: Using BLOSUM matrices

Score the alignment

```
M Q L E A N A D T S V
  :     :   :
L Q E Q A E A Q G E M
```

Using the BLOSUM62 scoring matrix (Table 7.3), we see that

$$\begin{aligned} S &= 2+5-3-4+4+0+4+0-2+0+1 \\ &= 7 \quad \text{(half-bits)}. \end{aligned}$$

Now that we have seen how to obtain and use BLOSUM matrices, we should examine Table 7.3 to make sure that the scores make biological sense. The largest score (11 half-bits) is for conservation of tryptophan (`W`) in two sequences. Tryptophan is a relatively rare amino acid, so there should be a larger "reward" when it appears at corresponding positions in an alignment of two sequences. The lowest scores are -4 for aligning a translational stop `*` with any other amino acid or some unfavorable alignments such as `D` opposite `L`. The low score in the latter case can be understood because aspartic acid

(D) codons GAC and GAU are at least two mutations away from those of leucine (L)(UUA, UUG, CUA, CUC, CUG, CUU) and because the properties of D (polar and negatively charged) are quite different from those of L (nonpolar and neutral). In contrast, aligning isoleucine (I) opposite leucine produces a positive score of 2 half-bits. All three codons for isoleucine (AUA, AUC, AUU) are only one mutation away from a leucine codon, and isoleucine has the same properties as leucine (nonpolar and neutral). Thus it is relatively easier to produce this mutation, and the mutation is more likely to be tolerated under selection. In summary, if the alignment is between chemically similar amino acids, the score will be positive. It will be zero or negative for dissimilar amino acid residues. Also, when aligning an amino acid with itself, scores for aligning rare amino acids are larger than scores for aligning common ones.

7.6 Tests of Alignment Methods

At this point, we should remind ourselves *why* we are performing alignments in the first place. In many cases, the purpose is to identify homologs of the query sequence so that we can attribute to the query annotations associated with its homologs in the database. The question is, "What are the chances of finding in a database search HSPs that are *not* homologs?" Over evolutionary time, it is possible for sequences of homologous proteins to diverge significantly. This means that to test alignment programs, some approach other than alignment scores is needed to find homologs. Often the three dimensional structures of homologs and their domain structures will be conserved, even though the proteins may have diverged in sequence. Structure can be used as a criterion for identifying homologs in a test set.

A "good" alignment program meets at least two criteria: it maximizes the number of homologs found (true positives), and it minimizes the number of nonhomologous proteins found (false positives). Another way to describe these criteria is in terms of *sensitivity* and *specificity*, which are discussed in more detail in Chapter 9. In this context, sensitivity is a measure of the fraction of actual homologs that are identified by the alignment program, and the specificity is a measure of the fraction of HSPs that are not actually homologs. Brenner et al. (1998) tested a number of different alignment approaches, including Smith-Waterman, FASTA, and an early version of BLAST. They discovered that, at best, only about 35% of homologs were detectable at an error frequency of 0.1% per query sequence.

An intuitive measure of homology employed in the past was the percentage of sequence identity. The rule of thumb was that sequence identities of 25%−30% in an alignment signified true homology. Brenner et al. employed a database of known proteins annotated with respect to homology/nonhomology relationships. Their results are shown in Fig. 7.6. Figure 7.6B shows percentage identity plotted against alignment length for proteins that are *not* homologs. For comparison, a threshold percentage identity taken to imply

similar structure is plotted as a line (see Brenner et al., 1998 for details). The point is that for alignments 100 residues in length, about half of the *nonhomologous* proteins show *more* than 25% sequence identity. At 50 ± 10 residues of alignment length, there are a few nonhomologous proteins having over 40% sequence identity. A particular example of this is shown in Fig. 7.6A. This serves as a reminder of why methods providing detailed statistical analysis of HSPs are required (Section 7.4.2).

References

Altschul SF, Gish W, Miller W, Myers EW, Lipman DJ (1990) Basic local alignment search tool. *Journal of Molecular Biology* 215:403–410.

Altschul SF, Madden TL, Schaffer AA, Zhang J, Zheng Z, Miller W, Lipman DJ (1997) Gapped BLAST and PSI-BLAST: A new generation of protein database search programs. *Nucleic Acids Research* 25:3389–3402.

Brenner SE, Chothia C, Hubbard TJP (1998) Assessing sequence comparison methods with reliable structurally identified distant evolutionary relationships. *Proceedings of the National Academy of Sciences USA* 95:6073–6078.

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

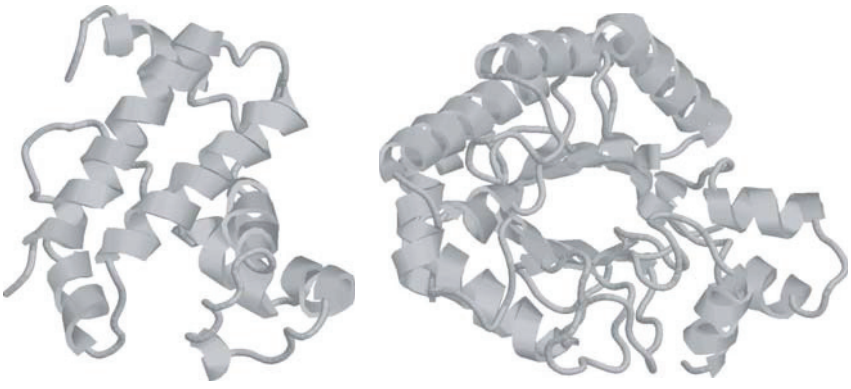
Henikoff S, Henikoff JG (1992) Amino acid substitution matrices from protein blocks. *Proceedings of the National Academy of Sciences USA* 89:10915–10919.

Pearson WR, Lipman DJ (1988) Improved tools for biological sequence comparison. *Proceedings of the National Academy of Sciences USA* 85:2444–2448.

Wilbur WJ, Lipman DJ (1983) Rapid similarity searches of nucleic acid and protein data banks. *Proceedings of the National Academy of Sciences USA* 80:726–730.

Fig. 7.6 ***(opposite page).*** The limitations of sequence identity as an indicator of homology. Panel A: Unrelated proteins that have a 40% sequence identity over a segment of approximately 60 residues. Panel B: Scores of unrelated, nonhomologous proteins as a function of alignment length. The line indicates the sequence identity cutoff, sometimes taken as an indicator of homology. Reprinted, with permission, from Brenner SE et al. (1998) *Proceedings of the National Academy of Sciences USA* 95:6073-6078.

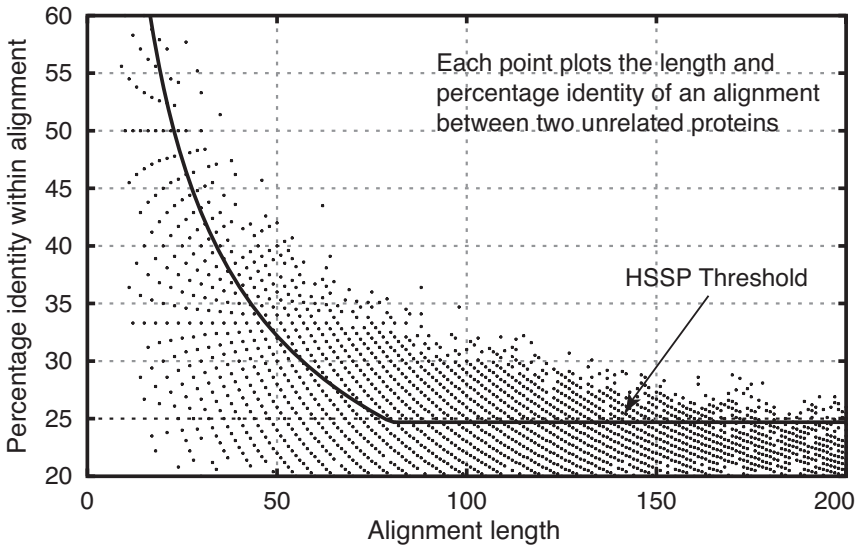
A.



Hemoglobin β-chain (1hdsb) Cellulase E2 (1tml_)

```
1hdsb  GKVDVDVVGAQALGR--LLVVYPWTQRFFQHFGNLSSAGAVMNNPKVKAHGKRVLDAFTQGLKH
       :.:::. . .::: :.  .::::    :     :: ::.::    :. .:. .  .: :. :::.
1tml_  GQVDALMSAAQAAGKIPILVVYNAPGR---DCGNHSSGGA----PSHSAY-RSWIDEFAAGLKN
```

B.



Exercises

Exercise 1. Find "serendipity" in a dictionary using splits as described in the text. How many splits were required? Compare this number with $|\log_2(\#\text{ pages})|$. Suppose that each split divided the pages remaining into thirds instead of halves. What is the formula relating the number of steps to the number of pages in that case?

Exercise 2. In Section 7.2.1, it was suggested that the statistic

$$\sum_w (\#L_w(\mathrm{I})) \times (\#L_w(\mathrm{J}))$$

could be used to determine quickly whether I and J share sufficient sequence similarity to justify a "full-bore" dynamic programming alignment. For DNA with $k = 3$, `length(I)` $= 100$, `length(J)` $= 1000$, indicate how you could use simulation with iid letters to set a threshold value for this statistic, for deciding when to employ dynamic programming.

Exercise 3. Suggest an alternative to FASTA for rapidly searching a large search space J >> I using the approach in Section 7.2.1. Would you need to modify your method for setting the threshold?

Exercise 4. For the sequences I = `GCATCGGC` and J = `CCATCGCCATCG`, find matching 4-words shared by I and J, as described in Section 7.2.3. Do this by making a table similar to Table 7.2, but only listing 4-words that actually occur in I or J. (Otherwise, the table would have $4^4 = 256$ rows!)

Exercise 5. Compute the average number of non-empty elements in a dot matrix comparison for $k = 1$, with I and J both drawn from a human DNA sequence (41% `G+C`).

Exercise 6. Given strings X and Y, each having differing base compositions, write out the formula for calculating $p = \mathbb{P}$(two random letters are equal), defined in Section 7.4.2. Clearly define the symbols that you use.

Exercise 7. For I = `TTGGAATACCATC` and J = `GGCATAATGCACCCC`, make dot matrices for the k-tuple hits for $k = 1$, 2, and 3.

Exercise 8. The *E. coli* F plasmid transfer origin region contains the sequence

I:
`5′-ATAAATAGAGTGTTATGAAAAATTAGTTTCTCTTACTCTCTTTATGATATTT`
`AAAAAAGCG-3′`

The TraY protein has been shown to bind to a region some 1600 bp away at a site that contains the sequence

J:
`5′-TAACACCTCCCGCTGTTTAT-3′`

Perform dot-matrix analyses for $k = 1$ and $k = 2$ to locate in I a subsequence that is similar to J. This identifies a potential binding site for TraY in sequence I. [Hint: Use R to construct a matrix having dimensions determined by the lengths of J and I, initialize elements to zero, and then substitute the appropriate elements with the number 1 to indicate matches. Produce two matrices: one for $k = 1$ and another for $k = 2$.]

Exercise 9. For I = GTATCGGCGC and J = CGGTTCGTATCGTCG, make a 2-word list for J. Then execute the FASTA algorithm (Example 7.1) beginning with $S_l = 0$. Compute S_i as you go through I, beginning at $i = 1$ and the 2-word GT, and ending at $i = 9$.

Exercise 10. To search J = CGGTTCGTATCGTCG for matches to TTCG within one mismatch, first make a list of all possible matches. How many matches are there within a single mismatch neighborhood of TTCG? [Hint: There is one exact pattern, and there are $3 \times 4 = 12$ single-mismatch patterns.]

Exercise 11. Download AB125166 and X68309 from the NCBI nucleotide databases (see Appendix B for the URL). Edit both sequences so that they contain the first 1000 positions in FASTA format. Then perform a Smith-Waterman local alignment using resources at http://www.cmb.usc.edu/, setting mismatch and gap parameters at 1000, and requesting return of the top 100 alignments.

a. How many alignments are there of length $t \geq 8$?
b. Use the expression for λ in Section 7.4.2 to compute the expected number of alignments of length at least 8 for sequences of this size (see Exercise 4 for computing p).
c. Use R to simulate ten pairs I and J of iid sequences having the same base compositions as in the first 1000 nucleotides of AB125166 and X68309. Then perform the Smith-Waterman alignment on each of the 10 pairs, and calculate the average number of alignments of length at least 8. Compare your result with those from part a and part b above, and explain agreements or disagreements.
d. The probability that there is a local alignment of length t or more is approximately
$$1 - \exp[-nm(1-p)p^t]\,.$$
Calculate the probability (called a p-value) for t = optimal alignment score in part a. What do you conclude from this p-value? Explain your answer carefully.

Exercise 12. Using only the data contained in the two blocks shown in Fig. 7.4, compute f_{RR} and p_{RR} as defined in Section 7.5.3. [Hint: Only columns that contain two or more instances of an R residue need be considered.]

Note: Because the computation above uses only a tiny sample of the total number of blocks available, the result computed here is not expected to lead to a score that agrees with the one in a BLOSUM matrix.

Exercise 13. To test the probability of irrelevant hits from a BLAST search, download the first two paragraphs of Jane Austen's *Emma* from a Web page located with a search engine of your choice. Remove all spaces and punctuation, and then replace letters "o" and "u" by "a" (alanine) and letters "b", "x", "j", and "z" by "g" (glycine). This should yield a string having about 500 characters. Add a first line `>emma` to convert to FASTA format, and then run WU-BLAST2 for proteins on the server at the European Bioinformatics Institute (see Appendix B). What are the E values and percentage identities for the top three sequences? How do these compare with real biological sequences?