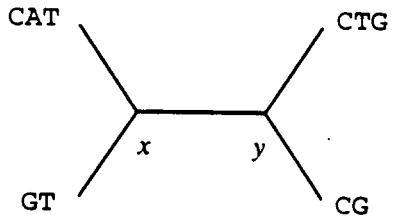

FIGURE 3.16

Input for a tree alignment problem.

Section 3.6.1). In this case we look for a sequence assignment that minimizes the distance sum. References for these results are given in the bibliographic notes.

3.5 DATABASE SEARCH

With the advent of fast and reliable technology for sequencing nucleic acids and proteins, centralized databases were created to store the large quantity of sequence data produced by labs all over the world. This created a need for efficient programs to be used in queries of these databases. In a typical application, one has a *query* sequence that must be compared to all those already in the database, in search of local similarities. This means hundreds of thousands of sequence comparisons.

The quadratic complexity of the methods we have seen so far for computing similarities and optimal alignments between two sequences makes them unsuitable for searching large databases. To speed the search, novel and faster methods have been developed. In general, these methods are based on heuristics and it is hard to establish their theoretical time and space complexity. Nevertheless, the programs based on them have become very important tools and these techniques deserve careful study.

In this section we concentrate on two of the most popular programs for database search. Neither uses pure dynamic programming, although one of them runs a variant of the dynamic programming method to refine alignments obtained by other methods.

Before we start describing these programs we make a little digression to explain the foundations of certain scoring matrices for amino acids, which are very important in database searches and in protein sequence comparison in general.

3.5.1 PAM MATRICES

When comparing protein sequences, simple scoring schemes, such as $+1$ for a match, 0 for a mismatch, and -1 for a space, are not enough. Amino acids, the residues that make up protein sequences, have biochemical properties that influence their relative replaceability in an evolutionary scenario. For instance, it is more likely that amino acids of similar sizes get substituted for one another than those of widely different sizes. Other prop-

erties such as the tendency to bind with water molecules also influence the probability of mutual substitution. Because protein comparisons are often made with evolutionary concerns in mind, it is important to use a scoring scheme that reflects these probabilities as much as possible.

The factors that influence the probability of mutual substitution are so numerous and varied that direct observation of actual substitution rates is often the best way of deriving similarity scores for pairs of residues. A standard procedure toward this goal is based on an important family of scoring matrices, the so-called PAM matrices, very popular among practitioners in the area. The acronym PAM stands for *Point Accepted Mutations*, or *Percent of Accepted Mutations*, in a reference to the fact that the basic 1-PAM matrix reflects an amount of evolution producing on average one mutation per hundred amino acids. In this section we briefly describe how PAMs are computed and what they mean.

Our description departs slightly from the original derivation of the matrices, but it highlights the role played by amino acid frequencies and mutabilities in coming up with the scores. In particular, the definition of mutability may be different in different sources.

Before anything else, the user has to choose an evolutionary distance at which to compare the sequences. These matrices are functions of this distance. For instance, a 250-PAM matrix is suitable for comparing sequences that are 250 units of evolution apart. In our derivation, we first construct the matrix corresponding to 1 PAM and then obtain the matrices for other distances from this one. Also, mutations are viewed at the amino acid level only, not at the DNA level.

For each evolutionary distance we have a *probability transition matrix* M and a *scores matrix* S. The scores matrix is obtained from the probability matrix, so we start our description with M. The necessary ingredients for building the 1-PAM matrix M are the following:

- A list of *accepted mutations*
- The *probabilities of occurrence* p_a for each amino acid a

An *accepted mutation* is a mutation that occurred and was positively selected by the environment; that is, it did not cause the demise of the particular organism where it occurred. One way to collect accepted mutations is to align two homologous proteins from different species, for example, the hemoglobin alpha chain in humans and in orangutans. Each position where the sequences differ will give us an accepted mutation. We consider these accepted mutations as undirected events; that is, given a pair a, b of aligned amino acids, we do not know for sure which one mutated into the other. Whichever was present in this same position in the ancestral sequence is the one that mutated into the other, but we do not know the ancestral sequence. It is even possible that the ancestral sequence contained a third amino acid that mutated into the two present ones, but this possibility is minimized taking very closely related sequences. It is important for the basic 1-PAM matrix that we consider immediate mutations, $a \to b$, not mediated ones like $a \to c \to b$.

The probabilities of occurrence can be estimated simply by computing the relative frequency of occurrence of amino acids over a large, sufficiently varied protein sequence set. These numbers satisfy

$$\sum_a p_a = 1.$$

From the list of accepted mutations we can compute the quantities f_{ab}, the number of times the mutation $a \leftrightarrow b$ was observed to occur. Recall that we are dealing with undirected mutations here, so $f_{ab} = f_{ba}$. We will also need the sums

$$f_a = \sum_{b \neq a} f_{ab},$$

which is the total number of mutations in which a was involved, and

$$f = \sum_a f_a,$$

the total number of amino acid occurrences involved in mutations. The number f is also twice the total number of mutations.

The frequencies f_{ab} and the probabilities p_a are all that is needed to build a 1-PAM transition probability matrix M. This is a 20×20 matrix with M_{ab} being the probability of amino acid a changing into amino acid b. Note that a and b may be the same, in which case we have the probability of a remaining unchanged during this particular evolutionary interval. For PAM matrices, the computation of M_{aa} is done based on the *relative mutability* of amino acid a, defined as

$$m_a = \frac{f_a}{100 f p_a}. \tag{3.12}$$

The mutability of an amino acid is a measure of how much it changes. It is the probability that the given amino acid will change in the evolutionary period of interest. Hence, the probability of a remaining unchanged is the complementary probability

$$M_{aa} = 1 - m_a.$$

On the other hand, the probability of a changing into b can be computed as the product of the conditional probability that a will change into b, given that a changed, times the probability of a changing. We estimate the conditional probability as the ratio between the $a \leftrightarrow b$ mutations and the total number of mutations involving a. Therefore,

$$\begin{aligned} M_{ab} &= \Pr(a \to b) \\ &= \Pr(a \to b \mid a \text{ changed}) \Pr(a \text{ changed}) \\ &= \frac{f_{ab}}{f_a} m_a. \end{aligned}$$

Notice that we are computing these probabilities using a simplified model of protein evolution. For instance, an amino acid is supposed to mutate independently of its past history, which may not be true given the nature of the genetic code. Also, we are ignoring the influence that other amino acids in the same sequence may have on the mutation of a given residue. The independence from past history in particular leads to a Markov-type model of evolution, which has good mathematical properties, some of which will be mentioned in the sequel.

It is easy to verify that M has the following properties.

$$\sum_b M_{ab} = 1 \tag{3.13}$$

$$\sum_a p_a M_{aa} = 0.99 \tag{3.14}$$

Equation (3.13) is merely saying that by adding up the probability of a staying the same and the probabilities of it changing into every other amino acid we get 1. Thus we are justified in calling these numbers "probabilities." Recall that these probabilities refer to a unit of evolutionary change. It is tempting to think of unit of evolution as unit of time, but it is generally accepted that different things change at different speeds, so time and amount of evolution are not directly proportional in a universal sense.

The unit of evolution used in this model is the amount of evolution that will change 1 in 100 amino acids on average. This will be referred to as 1 PAM evolutionary distance. The transition probability matrix has been normalized to reflect this fact as we can see from Equation (3.14). We achieve this normalization by using 100 in the denominator of Equation (3.12). Had we used another number, say 50, instead of 100 there, we would have obtained a matrix with exactly the same properties except that (3.14) would reflect a 1 in 50 average change, which is to say a 0.98 chance of no change. Thus, the unit of evolution would mutate one in 50 amino acids on average.

Once we have the basic matrix M we can derive transition probabilities for larger amounts of evolution. For instance, what is the probability that a will change into b in two PAM units of evolution? Well, in the first unit period a changes into any amino acid c, including itself, with probability M_{ac} and then c changes into b in the second period with probability M_{cb}. Adding this all up, we conclude that the final figure is nothing more than M^2_{ab}, that is, an entry in the square of M. In general, M^k is the transition probability matrix for a period of k units of evolution.

One interesting fact here (that can be proved) is that as k grows very large, say, on the order of a thousand, M^k converges to a matrix with identical rows. Each row will contain the relative frequency p_b in column b. That is, no matter what amino acid you pick to start with, after this long period of evolution the resulting amino acid will be b with probability p_b.

We are now ready to define the scoring matrices. The entries in these matrices are related to the ratio between two probabilities, namely, the probability that a pair is a mutation as opposed to being a random occurrence. This is called a *likelihood* or *odds* ratio.

Let us then compute this ratio for two amino acids a and b. Suppose that we paired a with b in a given alignment. Taking the point of view of a, the probability that b is there in the other sequence due to a mutation is M_{ab}. On the other hand, there is a chance of p_b for a random occurrence of b. The ratio is then

$$\frac{M_{ab}}{p_b}.$$

The reasoning is the same even if b equals a. The actual score is 10 times the logarithm of this ratio, because when we align several pairs the sum of the individual scores will then correspond to the product of the ratios. In practice, the actual values used are rounded to the nearest integer to speed up calculations. The logarithm is multiplied by 10 to reduce the discrepancy between the correct value and the integer approximation.

The foregoing discussion refers to 1 PAM. But we can use exactly the same scheme for an arbitrary evolutionary distance. The scoring matrix for k PAM distance is defined as follows:

$$score_k(a, b) = 10 \log_{10} \frac{M^k_{ab}}{p_b}.$$

It may not be clear at first sight, but this is actually a symmetric matrix. Scoring matrices should be symmetric to yield a comparison score that does not depend on which sequence is given first. It is easy to verify the symmetry for $k = 1$, since

$$\frac{M_{ab}}{p_b} = \frac{f_{ab} m_a}{f_a p_b} = \frac{f_{ab}}{100 f p_a p_b}.$$

For larger values of k, the symmetry can be shown to hold as well (see Exercise 15). This stems from the fact that the accepted mutations considered are undirected so that the same quantity $f_{ab} = f_{ba}$ is used in the computation of the probability of a changing into b and of b changing into a.

In spite of being generated from observed accepted mutations and overall amino acid frequencies alone, these scoring matrices end up reflecting several important chemical and physical properties of amino acids, such as their affinity to water molecules and their size. In fact, it is usually the case that residues with similar properties have high pairwise scores and that unrelated residues have lower scores. This should not be too surprising, because similar amino acids are more frequently interchanged in accepted mutations than are unrelated residues. Substitutions of similar residues tend to preserve most properties of the protein involved.

We said in the beginning that we need to fix the PAM distance in order to pick a scoring matrix and perform our comparisons. But sometimes we have two sequences and no information whatsoever on their true evolutionary distance. In this case, the recommended approach is to compare the sequences using two or three matrices that cover a wide range, for instance, 40 PAM, 120 PAM, and 250 PAM. In general, low PAM numbers are good for finding short, strong local similarities, while high PAM numbers detect long, weak ones.

3.5.2 BLAST

In this section we give an overview of the BLAST family of sequence similarity tools. The BLAST programs are among the most frequently used to search sequence databases worldwide. BLAST is an acronym for Basic Local Alignment Search Tool.

It is important to observe that here the term *database* refers simply to a usually large set of catalogued sequences. It does not imply any extra capabilities of fast access, data sharing, and so on, commonly found in standard database management systems. For us, therefore, this "database" is merely a collection of sequences, although sequence information is copiously complemented with additional information such as the origin of the data, bibliographic references, sequence function (if known), and others.

BLAST returns a list of *high-scoring segment pairs* between the query sequence and sequences in the database. Before explaining what this means and how BLAST obtains its results, we will briefly introduce some terminology on segment pairs to keep this discussion consistent with the original paper describing BLAST.

A *segment* is a substring of a sequence. Given two sequences, a *segment pair* between them is a pair of segments of the same length, one from each sequence. Because

the substrings in a segment pair have the same length, we can form a gapless alignment with them. This alignment can be scored using a matrix of substitution scores. No gap-penalty functions are needed, as there are no gaps. The score thus obtained is by definition the score of the segment pair. An example of a segment pair scored with the PAM120 substitution matrix is given in Figure 3.17. Segment pairs are basically gapless local alignments.

K	A	L	M	R		
V	A	K	N	S		
−4	3	−4	−3	−1	→	Total: −9

FIGURE 3.17

Segment pair and its score under PAM120.

We are now in possession of all the information necessary to describe precisely what BLAST does. Given a query sequence, BLAST returns all segment pairs between the query and a database sequence with scores above a certain threshold S. The parameter S can be set by the user, although a default value is provided in most servers that run the program. We have mentioned that a characteristic of reported alignments is the absence of gaps. This is actually a major reason why BLAST is so fast, since looking for good alignments with gaps is a good deal more time consuming.

A *maximum segment pair* (MSP) between two sequences is a segment pair of maximum score. This score is a measure of sequence similarity and can be computed precisely by dynamic programming. However, BLAST estimates this number much faster than any dynamic programming method. The program also returns *locally* maximal segment pairs, that is, those that cannot be improved further by extending or shortening them.

How BLAST Works

The BLAST approach to computing high-scoring segment pairs is as follows. It finds certain "seeds," which are very short segment pairs between the query and a database sequence. These seeds are then extended in both directions, without including gaps, until the maximum possible score for extensions of this particular seed is reached. Not all extensions are looked at. The program has a criterion to stop extensions when the score falls below a carefully computed limit. There is a very small chance of the right extension not being found due to this time optimization, but in practice this tradeoff is highly acceptable.

We may think of BLAST as a three-step algorithmic procedure, undertaking the following tasks.

1. Compile list of high-scoring strings (or *words*, in BLAST jargon).
2. Search for hits — each hit gives a seed.
3. Extend seeds.

The particular algorithmic steps depend on the type of sequences compared: DNA or protein. We detail each case next.

For protein sequences, the list of high-scoring words consists of all words with w characters (called w-mers) that score at least T with some w-mer of the query, using a PAM matrix to compute scores. Here w and T are program parameters. Note that this list may not contain all query w-mers! If a query w-mer consists of very common amino acids, it may be left out because even its score with itself may fall below T. However, there is an option to force inclusion of all query w-mers. The recommended value of w, the seed size, is 4 for protein searches.

Two approaches are tried for scanning the database in search for hits in the list constructed in the previous step. One of them is to arrange the list words into a hash table. Then, for each database word of size w, it is easy to get their index in the table and compare it to the words there, which will be a small fraction of all list words.

The second method uses a *deterministic finite automaton* to search for hits. This device has states and transitions and operates like a machine. It begins in a fixed initial state, and for each character in the database a transition is made to another state. Depending on the state and on the transition, a word from the list is recognized. The automaton is built only once, using the list of high-scoring words as input, and is a compact way of storing all these words. The search is fast, as it requires only a transition per character.

The final extension is straightforward. As we mentioned, to save time the algorithm stops when the score falls a certain distance below the best one obtained to that point for shortest extensions. This is done in both directions, and the high-scoring segment pair originated from this seed is kept. There is a small probability of missing important extensions, but it is negligible.

For DNA searches, the initial list contains only the query w-mers. Because scoring of DNA sequences is easier, this is enough for all practical purposes. The scanning strategy is radically different from the protein case. Taking advantage of the fact that the alphabet size is 4, the database is first compressed so that each nucleotide is represented using 2 bits. Four nucleotides fit in a byte. Apart from the space saved, the search can now be made much faster, because a byte is compared each time. There is an extra filtering step that removes from the initial list very common words from the database, to avoid a large number of spurious hits.

Extension is done in a way similar to the case of proteins. In both cases, the extension is based on a well-founded statistical theory that gives exact distribution of gapless local maximum score for random sequences, and permits a very accurate computation of the probability that the segment pair found could be possible due to chance alone. The smaller this probability, the more significant is the match.

We briefly sketch now the main points of this statistical theory. The distribution of the MSP score (defined above) for random sequences s and t of lengths m and n, respectively, can be accurately approximated as described next. This approximation gets better as m and n increase.

Given a matrix of replacement costs s_{ij} for the pairs of characters in the alphabet, and the probability p_i of occurrence of each individual character in the sequences, we first compute a value λ, solving the equation

$$\sum_{i,j} p_i p_j e^{\lambda s_{ij}} = 1.$$

The parameter λ is the unique positive solution to this equation and can be obtained by Newton's method. Once λ is known, the expected number of distinct segment pairs between s and t with score above S is

$$Kmne^{-\lambda S},$$

where K is a calculable constant. Actually, the distribution of the number of segment pairs scoring above S is a Poisson distribution with mean given by the previous formula. From this, it is easy to derive expressions for useful quantities like the average score, intervals where the score will fall 90% of the time, and so on.

3.5.3 FAST

FAST is another family of programs for sequence database search. The first program to be made generally available, FASTP, is designed to search a database of sequences looking for similarities to a given query sequence. The basic algorithm used by FASTP is to compare each database string in turn to the query and report those found significantly similar to it, along with alignments and other relevant information. The speed of FASTP is therefore mainly due to its ability to compare two sequences very quickly. Accordingly, our focus in this section is on how FASTP performs this basic step.

Let s and t denote the two sequences being compared, and let us assume for the moment that they are protein sequences. Their lengths are denoted by $m = |s|$ and $n = |t|$. The comparison starts by determining k-tuples common to both sequences, with $k = 1$ or 2. The value of k is a parameter called *ktup* in the program. In addition, the *offset* of a common k-tuple is important in the algorithm. The offset is a value between $-n + 1$ and $m - 1$ that determines a relative displacement of one sequence relative to the other. Specifically, if the common k-tuple starts at positions $s[i]$ and $t[j]$, we say that the offset is $i - j$.

The following data structures are needed: (1) a lookup table, and (2) a vector indexed by offsets and initialized with zeros. Sequence s is scanned and a table listing all positions of a given k-tuple in s is produced (see Figure 3.18). Then sequence t is scanned and each k-tuple in it is looked up in the table. For all common occurrences the vector entry of the corresponding offset is incremented. Figure 3.18 exhibits the final contents of the offset vector and the lookup table for sequences s = `HARFYAAQIVL` and t = `VDMAAQIA`. Notice that offset +2 has the highest entry value, meaning that many matches were found at this offset. This method is known as the *diagonal* method, because an offset can be viewed as a diagonal in a dynamic programming matrix. One possible use of the highest offset is to run a dynamic programming algorithm around the diagonal corresponding to this offset.

But FASTP does a more detailed analysis of the common k-tuples and joins two or more such k-tuples when they are in the same diagonal and not very far apart. The exact criteria are heuristic. These combined k-tuples form what is called a *region*. A region can be thought of as a segment pair, in the terminology of BLAST, or else as a gapless local alignment. Regions are given a certain score depending on the matches and mismatches contained in them. The important thing to remember is that regions have no gaps.

The next step consists in rescoring the five best regions received from the previous

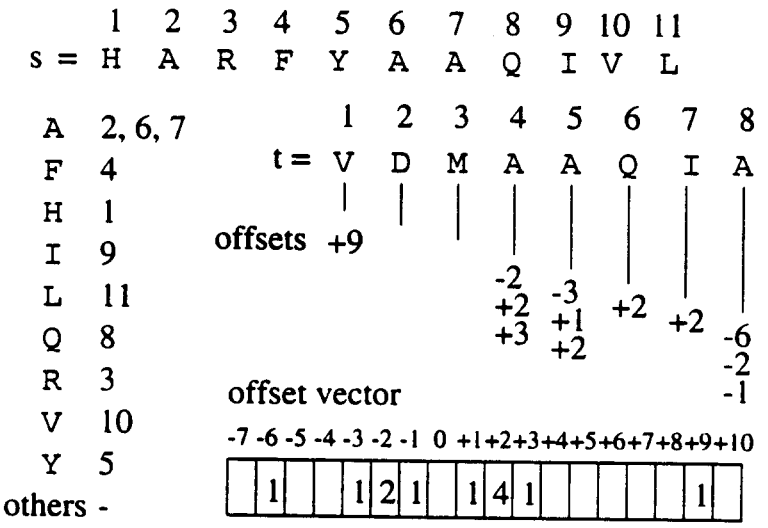

FIGURE 3.18

Lookup table and offset vector for sequence comparison by FASTP. The value of ktup is 1.

phase using a PAM matrix, usually PAM120 or PAM250. The best of these new scores is a first measure of the similarity between s and t and is termed the *initial score*. An initial score is computed for every database sequence with respect to the query sequence. These values are reported in histogram form along with the mean score. The initial score is also used to rank all database sequences. For the highest ranking such sequences, an optimized score is computed running a dynamic programming algorithm restricted to a band around the initial alignment — the one that produced the initial score. This procedure is much like the method exposed in Section 3.3.4. In practice, when sequences are truly related, the optimized score is usually significantly higher than the initial score. This observation often helps distinguish between good alignments occurring by chance and true relationships.

The value of *ktup* affects the performance of the algorithm in terms of its sensitivity and selectivity. *Sensitivity* is the ability of a search tool to recognize distantly related sequences. *Selectivity* is the ability of the tool to discard false positives — matches between unrelated sequences. In general, sensitivity and selectivity are opposite goals. When using FASTP and other programs in the family, low *ktup* increases sensitivity and high *ktup* favors selectivity.

Apart from the search tools, the FAST family includes a program that is useful in assessing the statistical significance of a score. This tool works by scrambling one of the sequences, say, t, maintaining the amino acid composition but changing their order, and running a full dynamic programming algorithm between the scrambled version of t and the original s. This is repeated many times, so that the average score and its standard deviation can be computed. From these, a z-value is determined by the formula

$$z = \frac{\text{score} - \text{average score}}{\text{standard deviation of scores}}.$$

However, since the statistical distribution of similarity scores for random sequences is not a normal distribution, these z-values have limited usefulness.

Improvement to the FASTP program led to FASTA. One added feature is the ability to process DNA as well as protein sequences. In this respect, FASTA can be seen as a combination of FASTP with FASTN, a program designed specifically to work with nucleotide sequences. Another program in the family — TFASTA — is devoted to comparing a protein query sequence to a DNA database, doing the necessary translations as it goes.

Another addition is related to the computation of initial scores. FASTA takes an extra step after the best regions have been selected and tries to join nearby regions, even if they do not belong to the same diagonal. With this, initial scores improve significantly for related sequences and get closer to the improved scores ("optimized scores" in FASTA parlance). Furthermore, the ten best regions are kept in FASTA as opposed to five best in FASTP. Other programs using the same techniques are also part of the package. LFASTA is a tool for local similarity, in the sense that it reports more than one good alignment between a pair of sequences.

The statistical significance tool was also improved. Its most outstanding addition is the possibility of doing local shuffling of sequences. It was observed that very often a biologically unimpressive alignment got a high z-value because shuffling destroyed an uneven distribution of residues in the sequence. Local shuffling mitigates this problem. Shuffling is done in blocks of 10 to 20 residues, thus yielding shuffled sequences that are random yet have the same local composition of the original one. Then, if high scores were due to biased local composition, this will affect the mean score of the locally shuffled sequences. Other improvements in this tool include flexibility in choosing the scoring matrix and calculation of more scores for each shuffled sequence.

3.6 OTHER ISSUES

We now study some miscellaneous topics related to sequence comparison. The first has to do with the notion of *distance* between sequences and what its relationship is to similarity. In the second topic we discuss rules for the various choices we have when comparing sequences. Finally we briefly discuss the topics of string matching and exact sequence comparison.

⋆ 3.6.1 SIMILARITY AND DISTANCE

Similarity and distance are two approaches to comparing character strings. In the similarity approach, we are interested in the best alignment between two strings, and the score of such an alignment gives a measure of how much the strings are alike. In the distance approach, we assign costs to elementary edit operations and seek the less expensive composition of these that transforms one string into the other. Thus, the distance is a measure of how much the strings differ. In either case we are looking for a numeric value that measures the degree by which the sequences are alike or are different.

Up to now we have concentrated on computing similarity. This section introduces the concept of distance and relates it to similarity. We show that in many cases these two measures are related by a simple formula, so that we can easily obtain one measure from the other.

We must stress that our study of distances is restricted to global comparison only. The distance approach is not suitable for local comparison. This limitation is one of the reasons we based this chapter on the similarity approach. Table 3.2 summarizes the main differences between similarity and distance. Some of the terms may not be clear to the reader now, but they are explained in what follows.

TABLE 3.2

Summary of properties of similarity and distance.

	Similarity	*Distance*
Triangle inequality?	no	yes
Local comparison?	yes	no
$p(a, a) \neq p(b, b)$ possible?	yes	no

We start by defining precisely our notions of similarity and distance. Let s, t be two sequences over an alphabet Σ. This alphabet will usually be the DNA or amino acid alphabet, but the results here hold for more general cases. Recall that we are looking for a number that measures how much the strings are similar or are different. As before, we denote similarity by $\text{sim}(s, t)$, distance by $\text{dist}(s, t)$.

Similarity

A similarity measure is always based on alignments. Let us recall and refine the precise definition of alignment. An *alignment* between s and t is a pair of sequences (s', t') obtained from s and t, respectively, by insertion of space characters in them. The alignment $\alpha = (s', t')$ must satisfy:

1. $|s'| = |t'|$.
2. Removal of all spaces from s' gives s.
3. Removal of all spaces from t' gives t.
4. For each i, either $s'[i]$ or $t'[i]$ is not a space.

An alignment creates a connection between symbols $s'[i]$ and $t'[i]$ that occupy the same position i in each sequence. Symbols $s'[i]$ and $t'[i]$ are said to be aligned under α.

The similarity is the highest score of any alignment. We assume an additive scoring system. Let us also review and refine what we mean by a scoring system. A scoring system is composed of a pair (p, g) whose members are a function $p : \Sigma \times \Sigma \mapsto \mathbf{R}$, used to score pairs of aligned characters, and a space penalty g used to penalize spaces. Usually, $g < 0$, but we do not require this. With such a scoring system we are able to assign a numerical value, or *score*, to each possible alignment. We add $p(a, b)$ each time a is

matched with b in α, and add g every time a character a is paired with a space symbol. The total sum is the score of α, denoted $score(\alpha)$. The *similarity* between two sequences s and t according to this scoring system is

$$\text{sim}(s, t) = \max_{\alpha \in \mathcal{A}(s,t)} score(\alpha),$$

where $\mathcal{A}(s, t)$ is the set of all alignments between s and t. This scoring system is called *additive* because if we cut any alignment in two consecutive blocks, the score of the entire alignment is the sum of the scores of the blocks.

Distance

A *distance* on a set E is a function $d : E \times E \mapsto \mathbf{R}$ such that

1. $d(x, x) = 0$ for all $x \in E$ and $d(x, y) > 0$ for $x \neq y$
2. $d(x, y) = d(y, x)$ for all $x, y \in E$ (d is *symmetric*)
3. $d(x, y) \leq d(x, z) + d(y, z)$ for all x, y, and $z \in E$.

The third condition is known as the *triangle inequality*. This property is very useful in many contexts, and many algorithms rely on its validity. In the case of strings it is possible to define a distance on the set of all strings over Σ based on the amount of effort needed to transform one of them into the other.

By successive application of a number of admissible operations, any string can be transformed into any other. If we assign a cost to each admissible operation, we can define the distance between two strings as the minimum total cost needed to transform one into the other. Admissible operations are as follows.

1. Substitution of a character a by a character b
2. Insertion or deletion of an arbitrary character

To charge for these operations, we use a *cost measure* (c, h), where c is a function $c : \Sigma \times \Sigma \mapsto \mathbf{R}$ and h is a real value. The substitution of b for a costs $c(a, b)$. Insertion or deletion of a character costs h. The cost of a series σ of operations is just the sum of individual costs and is denoted by $cost(\sigma)$.

The *distance* between two sequences s and t according to a cost measure is

$$\text{dist}(s, t) = \min_{\sigma \in \mathcal{S}(s,t)} cost(\sigma),$$

where $\mathcal{S}(s, t)$ is the set of all series of operations transforming s into t.

Some restrictions on (c, h) are necessary to make this definition sound. First of all, we deal only with nonnegative cost values, otherwise the minimum would not make sense. We also assume that the cost function c is a symmetric function, otherwise the distance would not be necessarily symmetric. These minimal requirements guarantee that $\text{dist}(s, t)$ is symmetric and satisfies the triangle inequality. To make sure it is a distance, we also need the property that $c(a, b) > 0$ for $a \neq b$ and that $h > 0$. This is necessary to avoid cases where two sequences s and t are not equal but $\text{dist}(s, t) = 0$. We want the distance to be zero only when the sequences are identical.

We will also assume that c satisfies the triangle inequality, that is,

$$c(x, y) \leq c(x, z) + c(y, z) \tag{3.15}$$

for all x, y, and z in Σ. The reason for this last assumption is that even if we start with a pair (c, h) that does not satisfy it, we can always define a new pair (c', h) that does satisfy it and produces the very same distance function. For instance, if three characters x, y, and z are such that $c(x, y) > c(x, z) + c(z, y)$, then every time we need to replace x by y we will not do it directly but rather replace x by z and later z by y, producing the same effect at a lower cost. For all practical matters, it looks as though the effective cost of replacing x by y is $c(x, z) + c(z, y)$, not $c(x, y)$, as the former is the actual cost incurred for the substitution. By using (3.15) we avoid such situations.

Example 3.3 Taking

$$c(x, y) = \begin{cases} 0 & \text{if } x = y, \\ 1 & \text{if } x \neq y, \end{cases}$$

and $h = 1$, we have the so-called *edit distance* between two sequences, also known as the Levenshtein distance.

Computing the Distance

Given a cost measure (c, h) and a constant M, we can define a scoring system (p, g) as follows:

$$p(a, b) = M - c(a, b), \tag{3.16}$$

$$g = -h + \frac{M}{2}. \tag{3.17}$$

If we have an alignment α between two sequences s and t, it is possible to define a series σ of operations such that

$$score(\alpha) + cost(\sigma) = \frac{M}{2}(m + n), \tag{3.18}$$

where $m = |s|$ and $n = |t|$. To get σ, it suffices to divide the alignment α in columns as we did previously to compute the score. The columns correspond to admissible operations in a natural way. Character matches correspond to substitutions. Spaces correspond to insertions or deletions. The operations can be applied in any order because they act on disjoint regions of the alignment and do not interfere with one another.

Let us call σ the series of these operation done left to right. We shall now compute the score of α and the cost of σ. Suppose there are exactly l character matches in α, with the ith match formed by a_i in s and b_i in t. Suppose further that there are exactly r spaces in α. Granted that, we have the following expression for the score of α.

$$score(\alpha) = \sum_{i=1}^{l} p(a_i, b_i) + rg.$$