and therefore

$$\text{sim}(s,t) \leq M\left(n - \frac{k}{2} - 1\right) + 2\left(\frac{k}{2} + 1\right)g,$$

which leads to

$$k \leq 2\left(\frac{Mn - \text{sim}(s,t)}{M - 2g} - 1\right),$$

almost the same bound as above.

Observe that $M - 2g$ is a constant. The time complexity is then $O(dn)$, where d is the difference between the maximum possible score Mn — the score of two identical sequences — and the optimal score. Thus, the higher the similarity, the faster the answer.

It is straightforward to extend this method to general sequences, not necessarily with the same length. Space-saving versions can also be easily derived.

3.4 COMPARING MULTIPLE SEQUENCES

So far in this chapter we have concentrated on the comparison between a pair of sequences. However we often are given several sequences that we have to align simultaneously in the best possible way. This happens, for example, when we have the sequences for certain proteins that have similar function in a number of different species. We may want to know which parts of these sequences are similar and which parts are different. To get this information, we need to build a **multiple alignment** for these sequences, and that is the topic of this section.

The notion of multiple alignment is a natural generalization of the two-sequence case. Let $s_1, \ldots, s_k$ be a set of sequences over the same alphabet. A multiple alignment involving $s_1, \ldots, s_k$ is obtained by inserting spaces in the sequences in such a way as to make them all of the same size. It is customary to place the extended sequences in a vertical list so that characters — or spaces — in corresponding positions occupy the same column. We further require that no column be made exclusively of spaces. Figure 3.10 shows a multiple alignment involving four short amino acid sequences. (Multiple alignments are more common with proteins, so in some of our examples in this section we use sequences of amino acids.)

One important issue to decide in multiple alignment is the precise definition of the

```
MQPILLL
MLR-LL-
MK-ILLL
MPPVLIL
```

FIGURE 3.10

Multiple alignment of four amino acid sequences.

quality of an alignment. We will next study one way of scoring a multiple alignment based on pairwise alignments. In addition, scientists also look at multiple alignments by placing the sequences in a tree structure rather than piling them up. This leads to different measures of quality that we discuss in subsequent sections.

3.4.1 THE SP MEASURE

Scoring a multiple alignment is more complex than its pairwise counterpart. We restrict ourselves to purely additive functions here; that is, the alignment score is the sum of column scores. Therefore we need a way to assign a score to each column and then add them up to get the alignment score. However, to score a column, we want a function with k arguments, where k is the number of sequences. Each one of these arguments is a character or a space. One way to do that would be to have a k-dimensional array that could be indexed with the arguments and return the value. The problem with this approach is that it is necessary to specify a value for each possible combination of arguments, and there could be as many as $2^k - 1$ such combinations. A typical value for k is 10, and that results in more than 1000 possibilities. Some more manageable methods to define such a function are necessary.

Such methods can be obtained by determining "reasonable" properties that such a function should have. First, the function must be independent of the order of arguments. For instance, if a column has `I`, `-`, `I`, `V` and another has `V`, `I`, `I`, `-`, they should both receive the same score. Second, it should be a function that rewards the presence of many equal or strongly related residues and penalizes unrelated residues and spaces. A solution that satisfies these properties is the so-called **sum-of-pairs** (SP) function. It is defined as the sum of pairwise scores of all pairs of symbols in the column. For instance, the score of a column with the above content is

$$\begin{aligned} \textit{SP-score}(\mathtt{I}, -, \mathtt{I}, \mathtt{V}) = {} & p(\mathtt{I}, -) + p(\mathtt{I}, \mathtt{I}) + p(\mathtt{I}, \mathtt{V}) + \\ & p(-, \mathtt{I}) + p(-, \mathtt{V}) + p(\mathtt{I}, \mathtt{V}), \end{aligned}$$

where $p(a, b)$ is the pairwise score of symbols a and b. Notice that this may include a space penalty specification when either a or b is a space. This is a very convenient scheme, because it relies on pairwise scores like the ones we use for two-sequence comparison. The SP scoring system is widely used due to its simplicity and effectiveness.

A small but important detail needs to be addressed to complete the definition. Although no column can be composed exclusively by spaces, it is possible to have two or more spaces in a given column. When computing the SP score, we need a value for $p(-, -)$. This value is not specified in two-sequence comparison because it never appears there. However, it is necessary for SP-based multiple comparison. The common practice is to set $p(-, -) = 0$. This may seem strange, given that spaces are generally penalized (i.e., when there is a space, the pairwise score is negative), so two spaces should be even more so. Nevertheless, there are good reasons to define $p(-, -)$ as zero. One of them is related to pairwise alignments again. We often draw conclusions about a multiple alignment by looking at the pairwise alignments it induces. Indeed, in any multiple alignment, we may select two of the sequences and just look at the way they are aligned to each other, forgetting about all the rest. It is not difficult to see that this produces a pairwise

alignment, except for the fact that we may have columns with two spaces in them. But then we just remove these columns and derive a true pairwise alignment. An example of this procedure is presented in Figure 3.11. This is what we call the *induced* pairwise alignment, or the *projection* of a multiple alignment.

A very useful fact, which is true only if we have $p(-, -) = 0$, is the following formula for the SP score of a multiple alignment α:

$$SP\text{-}score(\alpha) = \sum_{i<j} score(\alpha_{ij}), \tag{3.10}$$

where α_{ij} is the pairwise alignment induced by α on sequences s_i and s_j. This is true because it reflects two ways of doing the same thing. We may compute the score of each column and then add all column scores, or compute the score for each induced pairwise alignment and then add these scores. In any case, we are adding, for each column c and for each pair (i, j), the score $p(s'_i[c], s'_j[c])$, where s' indicates the extended sequence (with spaces inserted). But this is true only if $p(-, -) = 0$, because these quantities appear in the first computation only.

Multiple alignment

```
1 PEAALYGRFT---IKSDVW
2 PEAALYGRFT---IKSDVW
3 PESLAYNKF---SIKSDVW
4 PEALNYGRY---SSESDVW
5 PEALNYGWY---SSESDVW
6 PEVIRMQDDNPFSFQSDVY
```

Get only sequences 2 and 4

```
PEAALYGRFT---IKSDVW
PEALNYGRY---SSESDVW
```

Remove columns with two spaces

```
PEAALYGRFT-IKSDVW
PEALNYGRY-SSESDVW
```

FIGURE 3.11

Induced pairwise alignment (projection).

Dynamic Programming

Having decided on a measure, or score, for determining the quality of a multiple alignment, we would like to compute the alignments of maximum score, given a set of sequences. These will be our *optimal alignments*.

It is possible to use a dynamic programming approach here, as we did in the two-sequence case. Suppose, for simplicity, that we have k sequences, all of the same length n.

We use a k-dimensional array a of length $n + 1$ in each dimension to hold the optimal scores for multiple alignments of prefixes of the sequences. Thus, $a[i_1, \ldots, i_k]$ holds the score of the optimal alignment involving $s_1[1..i_1], \ldots, s_k[1..i_k]$.

After initializing with $a[0, \ldots, 0] \leftarrow 0$, we must fill in this entire array. Just to store it requires $O(n^k)$ space. This is also a lower bound for the computation time, because we have to compute the value of each entry. The actual time complexity is higher for a number of reasons. First, each entry depends on $2^k - 1$ previously computed entries, one for each possible composition of the current column of the alignment. In this composition, each sequence can participate with either a character or a space. Because we have k sequences, we have 2^k compositions. Removing the forbidden composition of all spaces, we obtain the final count of $2^k - 1$. This incorporates a factor of 2^k to the already exponential time complexity.

Then there is the question of accessing the data in the array. Very few programming languages, and certainly none of the most popular ones, will let users define an array with the number of dimensions k set at run-time. The alternative is to implement our own access routines. In any case, with or without language support, we can expect to spend $O(k)$ steps per access.

Another issue is the computation of column scores. The SP method requires $O(k^2)$ steps per column, as there are $k(k-1)/2$ pairwise scores to add up. Simpler schemes — for instance, just count the number of nonspace symbols — require at least $O(k)$ steps, given that we have to look at all arguments.

Finally, there is the actual computing of the value of $a[i_1, \ldots, i_k]$, which involves a maximum operation. Using boldface letters to indicate k-tuples, the command we must perform can be written as

$$a[\,\boldsymbol{i}\,] \leftarrow \max_{\boldsymbol{b} \neq \boldsymbol{0}} \{a[\,\boldsymbol{i} - \boldsymbol{b}\,] + \mathit{SP\text{-}score}(\mathit{Column}(\,\boldsymbol{s},\ \boldsymbol{i},\ \boldsymbol{b}))\},$$

where $\boldsymbol{b}$ ranges over all nonzero binary vectors of k elements, and

$$\mathit{Column}(\,\boldsymbol{s},\ \boldsymbol{i},\ \boldsymbol{b}) = (c_j)_{1 \le j \le k}$$

with

$$c_j = \begin{cases} s_j[i_j] & \text{if } b_j = 1 \\ - & \text{if } b_j = 0. \end{cases}$$

The total running time estimate for this first plan for implementation is therefore $O(k^2 2^k n^k)$ if we use SP, or $O(k 2^k n^k)$ if column scores can be computed in $O(k)$. If k is fixed, k nested **for** loops can be used to fill in the array. Optimal alignments can be recovered from this array by a backtracking procedure analogous to the one used in the pairwise case (Section 3.2.1). Straightforward extensions yield a similar method for the case where the sequences do not necessarily have the same length. In any case, the complexity of this algorithm is exponential in the number of input sequences, and the existence of a polynomial algorithm seems unlikely: It has been shown that the multiple alignment problem with the SP measure is NP-complete (see the bibliographic notes).

Saving Time

The exponential complexity of the pure dynamic programming approach makes it unappealing for general use. The main problem is the size of the array. For three sequences we already have a cube of $O(n^3)$ cells, and as the number k of sequences grows we have larger and larger "volumes" to fill in. Clearly, if we could somehow reduce the amount of cells to compute, this would have a direct impact on processing time.

This section describes a heuristic that does exactly that. We will show how to incorporate it into the dynamic programming algorithm to speed up its computation. It is a heuristic because in the worst case all cells will have to be computed; in practice, however, we can expect a good speedup. The heuristic is based on the relationship between a multiple alignment and its projections on two-sequence arrays, and, in particular, it uses Equation (3.10) relating SP scores to pairwise scores. Thus, the method we are about to see works only for the SP measure.

The outline of the method is as follows. We have k sequences of length n_i, for $1 \leq i \leq k$, and we want to compute the optimal alignments according to the SP measure. We will still use dynamic programming, but now we do not want to treat all cells. We just want the cells "relevant" to optimal alignments, in some sense. But exactly which cells will we deem relevant, and why?

The answer is to look at the pairwise projections of the cell. In a preprocessing step, we create conditions that will allow us to perform a test of relevance for arbitrary cells. To take advantage of this test and reduce the number of cells we need to look at, we have to modify the fill-in order as well.

Relevance Test

Let α be an optimal alignment involving $s_1, \ldots, s_k$. The first thing we must know is that even though α is optimal its projections are not necessarily the best ones for the given sequence pair. Figure 3.12 shows a case in which a projection fails to be optimal. It is unfortunate that such cases can happen. Were it not for them, we could easily establish a test for relevant cells: A cell is relevant when each of its pairwise projections is part of an optimal alignment of the two sequences corresponding to the projection.

Before going on, let us remark that it is easy to use the comparison algorithms we have seen for two sequences s and t to produce a matrix in which each entry (i, j) con-

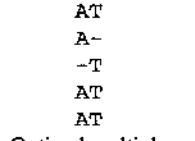

FIGURE 3.12

Optimal alignment with nonoptimal projection.

tains the highest score of an alignment that includes the *cut* (i, j). A pairwise alignment α contains a cut (i, j) when α can be divided into two subalignments, one aligning $s[1..i]$ with $t[1..j]$ and the other aligning the rest of s with the rest of t.

To obtain the desired values, we add two dynamic programming matrices a and b with the contents,

$$a[i, j] = \text{sim}(s[1..i], t[1..j])$$
$$b[i, j] = \text{sim}(s[i + 1..n], t[j + 1..m]),$$

where $n = |s|$ and $m = |t|$. The matrix a is just the standard matrix we have been using all along. The matrix b can be computed just like a, but backward. We initialize the last row and column and proceed backward until we reach $b[0, 0]$. We did this in Section 3.3.1, when we discussed linear space implementations of dynamic programming algorithms. The function *BestScoreRev* there does what we want in b. Then, sum $c = a + b$ contains exactly the highest score of an alignment that cuts at (i, j). We call c the matrix of *total scores*, while a and b are the matrices of *prefix* and *suffix* scores, respectively.

The matrix c is more appealing to the eyes than either a or b. We can quickly spot the best alignments just by looking at c, a feat that is considerably more difficult to do with either a or b. Let us explain this with an example. In Figure 3.13 we have the matrices a and c for a certain pair of sequences and a certain scoring system. Can you see exactly where the optimal alignments are by looking at a? It is much easier with c because we merely follow the cells with the highest score.

		G	A	T	T	C
	0	-2	-4	-6	-8	-10
A	-2	-1	-1	-3	-5	-7
T	-4	-3	-2	0	-2	-4
T	-6	-5	-4	-1	1	-1
C	-8	-7	-6	-3	-1	2
G	-10	-7	-8	-5	-3	0
G	-12	-9	-8	-7	-5	-2

		G	A	T	T	C
	-2	-2	-7	-12	-17	-22
A	-7	-4	-2	-7	-12	-17
T	-10	-7	-5	-2	-7	-12
T	-13	-10	-7	-5	-2	-7
C	-14	-13	-10	-5	-4	-2
G	-17	-14	-13	-8	-4	-2
G	-22	-17	-14	-11	-7	-2

FIGURE 3.13

Dynamic programming matrices: prefix scores (left) and total scores (right).

Although projections of optimal alignments may not be optimal themselves, we can establish a lower bound for the projection scores, as long as we have a lower bound for the optimal score. The following result tells us exactly how this works.

THEOREM 3.1 Let α be an optimal alignment involving $s_1, \ldots, s_k$. If $SP\text{-}score(\alpha) \geq L$, then

$$score(\alpha_{ij}) \geq L_{ij},$$

where

$$L_{ij} = L - \sum_{\substack{x < y \\ (x,y) \neq (i,j)}} (\text{sim}(s_x, s_y)).$$

Proof. We have, successively,

$$SP\text{-}score(\alpha) \geq L,$$

$$\sum_{x<y} score(\alpha_{xy}) \geq L,$$

$$\sum_{\substack{x < y \\ (x,y) \neq (i,j)}} (score(\alpha_{xy})) \geq L - score(\alpha_{ij}),$$

$$\sum_{\substack{x < y \\ (x,y) \neq (i,j)}} (\text{sim}(s_x, s_y)) \geq L - score(\alpha_{ij}),$$

from which the result follows. ■

We are now in a position to test whether a cell with index $i = (i_1, \ldots, i_k)$ is relevant to optimal alignments with respect to lower bound L. Simply put, this cell is relevant if all of its projections satisfy the conditions of the previous theorem. In other words, i is relevant when

$$c_{xy}[i_x, i_y] \geq L_{xy},$$

for all x and y such that $1 \leq x < y \leq k$, where c_{xy} is the matrix of total scores for s_x and s_y. According to the theorem, only relevant cells can participate in optimal alignments, although not *all* relevant cells will participate in optimal alignments. Nevertheless, this affords a reduction in the number of cells that are potential candidates for an optimal alignment. This reduction becomes more significant as L approaches the true optimal score.

To obtain a suitable bound L, we may just choose an arbitrary multiple alignment involving all sequences and take its score as L. Of course this is a lower bound since optimal alignments have the highest possible score. If we already have a reasonably good alignment, we can use its score and improve the alignment to get an optimal one by the method just sketched. An alternative lower bound can be obtained using the results of Section 3.4.2.

Implementation Details

We have to be a bit careful when implementing the heuristic just described. It is not enough to test all cells for relevance and then use only the relevant ones. This will achieve no appreciable time reduction, because we are still looking at all the cells to test them.

We need a way of actually cutting off completely the irrelevant cells, so that they are not even looked at.

Here is one possible strategy. We start with the cell at $\mathbf{0} = (0, 0, \ldots, 0)$, which is always relevant, and expand its influence to dependent relevant cells. Each one of these will in turn expand its influence, and so on, until we reach the final corner cell at $(n_1, \ldots, n_k)$. In the whole process, only relevant cells will be analyzed.

To explain the strategy better, we need a few definitions. A cell i *influences* another cell j if i is one of the cells used in the maximum computation to determine the value of $a[j]$. In this case, we also say that j *depends on* i. Another characterization of this fact is that $b = j - i$ is a vector with either 0 or 1 in each component, and $i \neq j$. So, each cell depends on at most $2^k - 1$ others, and influences at most $2^k - 1$ others. We say "at most" because border cells may influence or depend on less than $2^k - 1$ other cells.

We keep a pool of cells to be examined. Initially, only $\mathbf{0}$ is in the pool. The pool contains only relevant cells at all times. When a cell i enters the pool, its value $a[i]$ is initialized. During its stay in the pool, this value is updated. When this cell is removed from the pool, the current value of $a[i]$ is taken as the true value of this cell and used in a propagation to the relevant cells that depend on i.

The value of a cell is propagated as follows. Let j be a relevant cell that depends on i. If j is not in the pool, we put it there and initialize its value with the command

$$a[j] \leftarrow a[i] + \mathit{SP\text{-}score}(\mathit{Column}(s, i, b)).$$

If j is already in the pool, we conditionally update its value with the command

$$a[j] \leftarrow \max(a[j], a[i] + \mathit{SP\text{-}score}(\mathit{Column}(s, i, b))).$$

It is important to make sure that each time some cell is to be removed from the pool, the lexicographically smaller one is selected. This guarantees that this cell has already received the influences due to other relevant cells, and its value need not be further updated. The process stops when we reach cell $(n_1, \ldots, n_k)$, which is necessarily the last cell examined; its value is the SP score sought. The algorithm for computing the score is shown in Figure 3.14. To recover the optimal alignments, we need to keep track of the updates somehow. One way is to construct a dependence graph where the relevant cells are nodes and the edges represent influences that provided the maximum value. Time and space complexity of this algorithm are proportional to the number of relevant cells.

3.4.2 STAR ALIGNMENTS

Computing optimal multiple alignments can take a long time if we use the standard dynamic programming approach, even with the savings sketched in the previous section; so other methods have been developed. These alternative ways are often heuristic, in the sense that they do not yield any guarantee on the quality of the resulting alignment. They are simply faster ways of getting an answer, which in many cases turns out to be a reasonably good answer.

One such method is what has been termed the *star alignment* method. It consists in building a multiple alignment based upon the pairwise alignments between a fixed sequence of the input set and all others. This fixed sequence is the *center* of the star. The

Algorithm *Multiple-Sequence Alignment*
input: $s = (s_1, \ldots, s_k)$ and lower bound L
output: The value of an optimal alignment
// Compute L_{xy}, $1 \leq x < y \leq k$
for all x and y, $1 \leq x < y \leq k$ **do**
    Compute c_{xy}, the total score array for s_x and s_y
for all x and y, $1 \leq x < y \leq k$ **do**
    $L_{xy} \leftarrow L - \sum_{(p,q)\neq(x,y)} \text{sim}(s_p, s_q)$
// Compute array a
pool $\leftarrow \{0\}$
while *pool* not empty **do**
    $i \leftarrow$ the lexicographically smallest cell in the *pool*
    pool $\leftarrow$ *pool* $\setminus \{ i \}$
    if $c_{xy}[i_x, i_y] \geq L_{xy}, \forall x, y, 1 \leq x < y \leq k$, **then** *// Relevance test*
        for all j dependent on i **do**
            if $j \notin$ *pool* **then**
                pool $\leftarrow$ *pool* $\cup \{ j \}$
                $a[j] \leftarrow a[i] + SP\text{-}score(Column(s, i, j - i))$
            else
                $a[j] \leftarrow \max(a[j], a[i] + SP\text{-}score(Column(s, i, j - i)))$
return $a[n_1, \ldots, n_k]$

FIGURE 3.14

Dynamic programming algorithm for multiple-sequence comparison with heuristics for saving time.

alignment α constructed is such that its projections α_{ij} are optimal when either i or j is the index of the center sequence.

Let $s_1, \ldots, s_k$ be k sequences that we want to align. To construct a star alignment, we must first pick one of the sequences as the center. We will postpone the discussion as to how this selection should be done. For the moment, let us just assume that the center sequence has been selected and its index is a number c between 1 and k. Next we need, for each index $i \neq c$, an optimal alignment between s_i and s_c. This can be obtained with standard dynamic programming, so this phase takes $O(kn^2)$ time, assuming all sequences have $O(n)$ length.

We aggregate these pairwise alignments using a technique known as "once a gap, always a gap," applied to the center sequence s_c. The construction starts with one of the pairwise alignments, say the one between s_1 and s_c, and goes on with each pairwise alignment being added to the bunch. During the process, we progressively increase the gaps in s_c to suit further alignments, never removing gaps already present in s_c — once a gap in s_c, always a gap.

Each subsequent pairwise alignment is added using s_c as a guide. We have a multiple alignment involving s_c and some other sequences in one hand, and a pairwise alignment between s_c and a new sequence in the other. We add as few gaps as necessary in both alignments so that the extended copies of s_c agree. Then just include the new extended sequence in the cluster, given that it now has the same length as the other extended sequences.

The time complexity of this joining operation depends on the data structure used to

represent alignments, but it should not be higher than $O(kl)$ using reasonable structures, where l is an upper bound on the alignment lengths. Because we have $O(k)$ sequences to add, we end up with $O(k^2l)$ for the joining phase. The total time complexity is then $O(kn^2 + k^2l)$. If we want to know the resulting score, this costs an extra $O(k^2l)$, but the asymptotic complexity remains the same.

How should we select the center sequence? One way is to just try them all and then pick the best resulting score. Another way is to compute all $O(k^2)$ optimal pairwise alignments and select as the center the string that maximizes

$$\sum_{i \neq c} \text{sim}(s_i, s_c). \tag{3.11}$$

Example 3.1 Consider the following five DNA sequences. We begin by constructing a table with the pairwise similarities among the sequences (see Figure 3.15). The score system used is the same as in Section 3.2.1, where we explain the basic algorithm.

$s_1 =$ ATTGCCATT
$s_2 =$ ATGGCCATT
$s_3 =$ ATCCAATTTT
$s_4 =$ ATCTTCTT
$s_5 =$ ACTGACC

	s_1	s_2	s_3	s_4	s_5
s_1		7	−2	0	−3
s_2	7		−2	0	−4
s_3	−2	−2		0	−7
s_4	0	0	0		−3
s_5	−3	−4	−7	−3	

FIGURE 3.15

Pairwise scores for the sequences in Example 3.1.

From the table we see that the first sequence, s_1, is the one that maximizes expression (3.11). Our next step is then to choose optimal alignments between s_1 and all the other sequences. Suppose we choose the following optimal alignments:

```
ATTGCCATT
ATGGCCATT

ATTGCCATT--
ATC-CAATTTT
```

```
ATTGCCATT
ATCTTC-TT

ATTGCCATT
ACTGACC
```

In this example the only spaces introduced in s_1, the center sequence, and in all alignments were the two spaces at the end forced by sequence s_3. Therefore, s_1 will have just these two gaps in the final multiple alignment. The other sequences are aligned to s_1 as in the chosen alignments. The result is shown here:

```
ATTGCCATT--
ATGGCCATT--
ATC-CAATTTT
ATCTTC-TT--
ACTGACC----
```

3.4.3 TREE ALIGNMENTS

An alternative to comparing multiple sequences is discussed in this section. The motivation for this approach is that sometimes we have an evolutionary tree for the sequences involved (evolutionary trees are the subject of Chapter 6). In this case, we can compute the overall similarity based on pairwise alignments along tree edges. Contrast this with the SP measure, which takes into account *all* parwise similarities.

Suppose that we are given k sequences and a tree with exactly k leaves, with a one-to-one correspondence between leaves and sequences. If we assign sequences to the interior nodes of the tree, we can compute a *weight* for each edge, which is the similarity between the two sequences in the nodes incident to this edge. The sum of all these weights is the *score* of the tree with respect to this particular sequence assignment to interior nodes. Finding a sequence assignment that maximizes the score is what has been called the **tree alignment** problem. Star alignments can be viewed as particular cases of tree alignments in which the tree is a star.

Example 3.2 Consider the input shown in Figure 3.16. Assigning sequence CT to vertex x and sequence CG to vertex y, we have a score of 6. The scoring system used is $p(a, b) = 1$ if $a = b$ and 0 otherwise, and $p(a, -) = -1$.

The tree alignment problem is NP-hard. There exists an algorithm that finds an optimal solution, but it is exponential in the number of sequences. By using the ideas developed in Section 3.4.1 it is possible to improve space and time requirements in practice. Approximation algorithms with good performance guarantees have been designed for the case when edge weights are defined in terms of distance rather than similarity (see

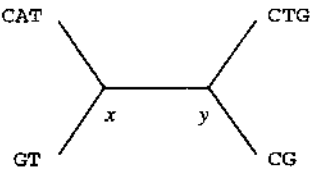

FIGURE 3.16

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-