CHAPTER

# 3

# SEQUENCE COMPARISON AND DATABASE SEARCH

In this chapter we present some of the most practical and widely used methods for sequence comparison and database search. Sequence comparison is undoubtedly the most basic operation in computational biology. It appears explicitly or implicitly in all subsequent chapters of this book. It has also many applications in other subareas of computer science.

## 3.1 BIOLOGICAL BACKGROUND

Why compare sequences? Why use a computer to do that? How to compare sequences using computers? These are the main questions that will concern us in this chapter. We will answer the first two in this section, and devote the rest of the chapter to the last one.

Sequence comparison is the most important primitive operation in computational biology, serving as a basis for many other, more complex, manipulations. Roughly speaking, this operation consists of finding which parts of the sequences are alike and which parts differ. However, behind this apparently simple concept, a great variety of truly distinct problems exist, with diverse formalizations and sometimes requiring completely different data structures and algorithms for an efficient solution.

As examples, we will give a list of problems that often appear in computational biology. In these examples we use two notions that will be precisely defined in later sections. One is the *similarity* of two sequences, which gives a measure of how similar the sequences are. The other is the *alignment* of two sequences, which is a way of placing one sequence above the other in order to make clear the correspondence between similar characters or substrings from the sequences. The examples (and the whole chapter) also use basic concepts about strings, which are defined in Section 2.1. Here are the examples.

*1.* We have two sequences over the same alphabet, both about the same length (tens of thousands of characters). We know that the sequences are almost equal, with

only a few isolated differences such as insertions, deletions, and substitutions of characters. The average frequency of these differences is low, say, one each hundred characters. We want to find the places where the differences occur.

2. We have two sequences over the same alphabet with a few hundred characters each. We want to know whether there is a prefix of one which is similar to a suffix of the other. If the answer is yes, the prefix and the suffix involved must be produced.
3. We have the same problem as in (2), but now we have several hundred sequences that must be compared (each one against all). In addition, we know that the great majority of sequence pairs are unrelated, that is, they will not have the required degree of similarity.
4. We have two sequences over the same alphabet with a few hundred characters each. We want to know whether there are two substrings, one from each sequence, that are similar.
5. We have the same problem as in (4), but instead of two sequences we have one sequence that must be compared to thousands of others.

Problems like (1) appear when, for instance, the same gene is sequenced by two different labs and they want to compare the results; or even when the same long sequence is typed twice into the computer and we are looking for typing errors. Problems like (2) and (3) appear in the context of fragment assembly in programs to help large-scale DNA sequencing. Problems like (4) and (5) occur in the context of searches for local similarities using large biosequence databases.

We will see in this chapter that a single basic algorithmic idea can be used to solve all of the above problems. However, this may not be the most efficient solution. Sometimes less general but faster methods are better suited to each task.

The use of computers hardly needs justification when we deal with large quantities of data, as in searches in large databases. Yet, even in cases where we could conceivably do comparisons "by hand," the use of computers is safer and more convenient, as the following examples, taken from a paper by Smith and coworkers [176], illustrate.

In a 1979 paper, Sims and colleagues examined similar regions from the DNA of two bacteriophages [173]. They presented an alignment between regions of the H-gene from phages St-1 and G4 containing 11 matches (that is, 11 columns in which there are equal characters in both sequences). The procedure they used to obtain good alignments was described as the "insertion of occasional gaps to maximize homology [number of identities]." Smith and colleagues [176], however, produced 12 matches in an alignment found by computer, which was certainly overlooked by the first group of researchers.

In another 1979 paper, Rosenberg and Court completed a study aligning 46 promoter sequences from various organisms [164]. The multiple alignment was constructed starting from the alignment of certain regions reputed relatively invariant and then extending this "seed" alignment in both directions without introducing gaps. Although the resulting alignment was good as a whole, the pairwise alignments obtained from it were relatively poor. This happens fairly frequently in multiple alignments, but in this case Smith and colleagues found, with the help of a computer, an alignment between two substrings of the sequences involved that had 44 identical characters over a total of 45. This is certainly an astonishingly good local alignment, and the fact that it is not even mentioned

by Rosenberg and Court must mean again that they were not able to find it by hand.

These two examples show that the use of computers can reveal intriguing similarities that might go unnoticed otherwise.

## 3.2 COMPARING TWO SEQUENCES

In this section we study methods for comparing two sequences. More specifically, we are interested in finding the best alignments between these two sequences. Several versions of this problem occur in practice, depending on whether we are interested in alignments involving the entire sequences or just substrings of them. This leads to the definition of global and local comparisons. There is also a third kind of comparison in which we are interested in aligning not arbitrary substrings, but prefixes and suffixes of the given sequences. We call this third kind semiglobal comparison. All the problems mentioned can be solved efficiently by dynamic programming, as we will see in the sequel.

### 3.2.1 GLOBAL COMPARISON — THE BASIC ALGORITHM

Consider the following DNA sequences: GACGGATTAG and GATCGGAATAG. We cannot help but notice that they actually look very much alike, a fact that becomes more obvious when we align them one above the other as follows.

```
GA-CGGATTAG
GATCGGAATAG
```
(3.1)

The only differences are an extra T in the second sequence and a change from A to T in the fourth position from right to left. Observe that we had to introduce a space (indicated by a dash above) in the first sequence to let equal bases before and after the space in the two sequences align perfectly.

Our goal in this section is to present an efficient algorithm that takes two sequences and determines the best alignment between them as we did above. Of course, we must define what the "best" alignment is before approaching the problem. To simplify the discussion, we will adopt a simple formalism; later, we will see possible generalizations.

To begin, let us be precise about what we mean by an *alignment* between two sequences. The sequences may have different sizes. As we saw in the recent example, alignments may contain spaces in any of the sequences. Thus, we define an alignment as the insertion of spaces in arbitrary locations along the sequences so that they end up with the same size. Having the same size, the augmented sequences can now be placed one over the other, creating a correspondence between characters or spaces in the first sequence and characters or spaces in the second sequence. In addition, we require that no space in one sequence be aligned with a space in the other. Spaces can be inserted even in the beginning or end of sequences.

Given an alignment between two sequences, we can assign a *score* to it as follows. Each column of the alignment will receive a certain value depending on its contents and the total score for the alignment will be the sum of the values assigned to its columns. If a column has two identical characters, it will receive value $+1$ (a *match*). Different characters will give the column value $-1$ (a *mismatch*). Finally, a space in a column drops down its value to $-2$. The best alignment will be the one with maximum total score. This maximum score will be called the *similarity* between the two sequences and will be denoted by $\text{sim}(s, t)$ for sequences $s$ and $t$. In general, there may be many alignments with maximum score.

To exercise these definitions, let us compute the score of alignment (3.1). There are nine columns with identical characters, one column with distinct characters, and one column with a space, giving a total score of

$$9 \times 1 + 1 \times (-1) + 1 \times (-2) = 6.$$

Why did we choose these particular values ($+1$, $-1$, and $-2$)? This scoring system is often used in practice. We reward matches and penalize mismatches and spaces. In Section 3.6.2 we discuss in more detail the choice of these parameters.

One approach to computing the similarity between two sequences would be to generate all possible alignments and then pick the best one. However, the number of alignments between two sequences is exponential, and such an approach would result in an intolerably slow algorithm. Fortunately, a much faster algorithm exists, which we will now describe.

The algorithm uses a technique known as *dynamic programming*. It basically consists of solving an instance of a problem by taking advantage of already computed solutions for smaller instances of the same problem. Given two sequences $s$ and $t$, instead of determining the similarity between $s$ and $t$ as whole sequences only, we build up the solution by determining all similarities between arbitrary *prefixes* of the two sequences. We start with the shorter prefixes and use previously computed results to solve the problem for larger prefixes.

Let $m$ be the size of $s$ and $n$ the size of $t$. There are $m + 1$ possible prefixes of $s$ and $n + 1$ prefixes of $t$, including the empty string. Thus, we can arrange our calculations in an $(m + 1) \times (n + 1)$ array where entry $(i, j)$ contains the similarity between $s[1..i]$ and $t[1..j]$.

Figure 3.1 shows the array corresponding to $s = \text{AAAC}$ and $t = \text{AGC}$. We placed $s$ along the left margin and $t$ along the top to indicate the prefixes more easily. Notice that the first row and the first column are initialized with multiples of the space penalty ($-2$ in our case). This is because there is only one alignment possible if one of the sequences is empty: Just add as many spaces as there are characters in the other sequence. The score of this alignment is $-2k$, where $k$ is the length of the nonempty sequence. Hence, filling the first row and column is easy.

Now let us concentrate on the other entries. The key observation here is that we can compute the value for entry $(i, j)$ looking at just three previous entries: those for $(i - 1, j)$, $(i - 1, j - 1)$, and $(i, j - 1)$. The reason is that there are just three ways of obtaining an alignment between $s[1..i]$ and $t[1..j]$, and each one uses one of these previous values. In fact, to get an alignment for $s[1..i]$ and $t[1..j]$, we have the following three choices:

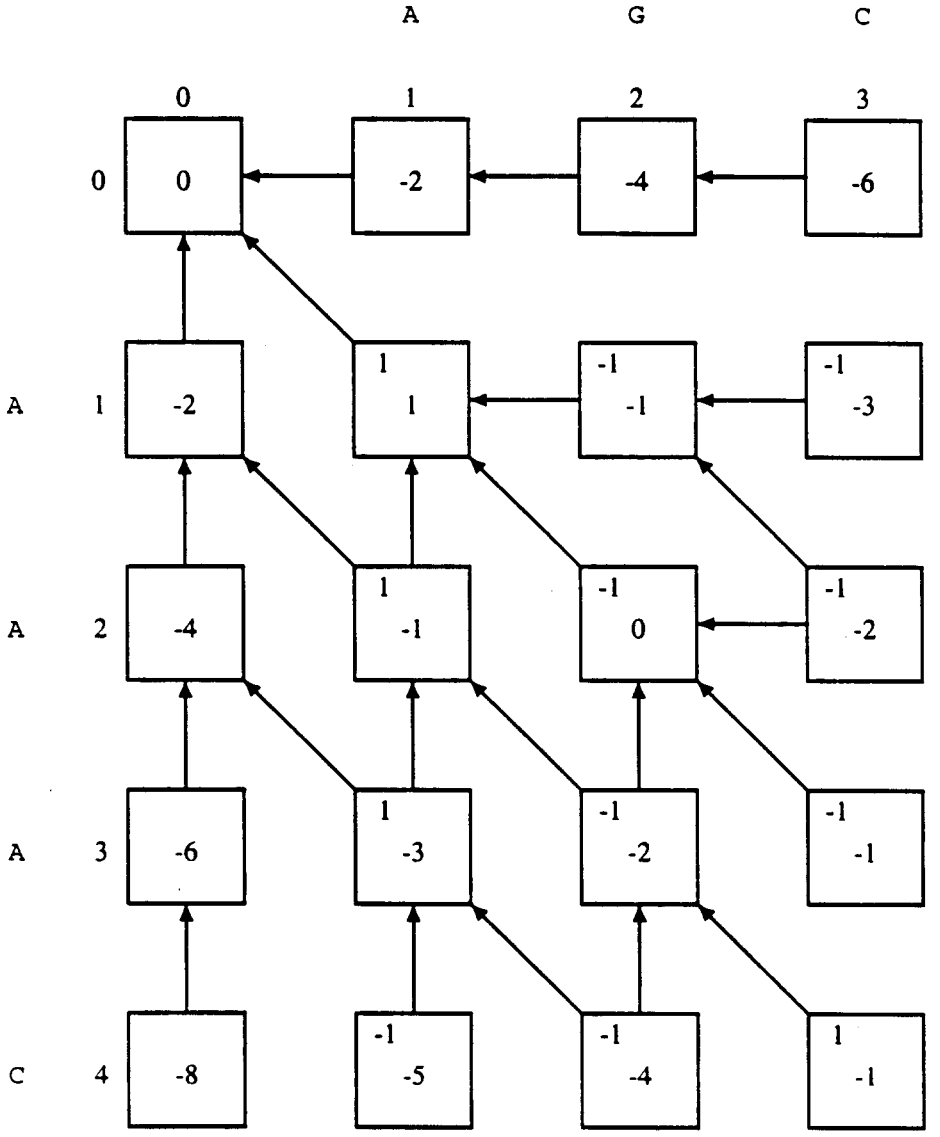


**FIGURE 3.1**

*Bidimensional array for computing optimal alignments. The value in the upper left corner of cell $(i, j)$ indicates whether $s[i] = t[j]$. Indices of rows and columns start at zero.*

- Align $s[1..i]$ with $t[1..j-1]$ and match a space with $t[j]$, or
- Align $s[1..i-1]$ with $t[1..j-1]$ and match $s[i]$ with $t[j]$, or
- Align $s[1..i-1]$ with $t[1..j]$ and match $s[i]$ with a space.

These possibilities are exhaustive because we cannot have two spaces paired in the last column of the alignment. Scores of the best alignments between smaller prefixes are already stored in the array if we choose an appropriate order in which to compute the entries. As a consequence, the similarity sought can be determined by the formula

$$\text{sim}(s[1..i], t[1..j]) = \max \begin{cases} \text{sim}(s[1..i], t[1..j-1]) - 2 \\ \text{sim}(s[1..i-1], t[1..j-1]) + p(i, j) \\ \text{sim}(s[1..i-1], t[1..j]) - 2, \end{cases} \tag{3.2}$$

where $p(i, j)$ is $+1$ if $s[i] = t[j]$ and $-1$ if $s[i] \neq t[j]$. The values of $p(i, j)$ are written in the upper left corners of the boxes in Figure 3.1. If we denote the array by $a$, this equation can be rewritten as follows:

$$a[i, j] = \max \begin{cases} a[i, j-1] - 2 \\ a[i-1, j-1] + p(i, j) \\ a[i-1, j] - 2. \end{cases} \tag{3.3}$$

As we mentioned earlier, a good computing order must be followed. This is easy to accomplish. Filling in the array either row by row, left to right on each row, or column by column, top to bottom on each column, suffices. Any other order that makes sure $a[i, j-1]$, $a[i-1, j-1]$, and $a[i-1, j]$ are available when $a[i, j]$ must be computed is fine, too.

Finally, we drew arrows in Figure 3.1 to indicate where the maximum value comes from according to Equation (3.3). For instance, the value of $a[1, 2]$ was taken as the maximum among the following figures.

$$a[1, 1] - 2 = -1$$
$$a[0, 1] - 1 = -3$$
$$a[0, 2] - 2 = -6.$$

Therefore, there is only one way of getting this maximum value, namely, coming from entry (1, 1), and that is what the arrows show.

Figure 3.2 presents an algorithm for filling in an array as explained. This algorithm computes entries row by row. It depends on a parameter $g$ that specifies the space penalty

```
Algorithm Similarity
    input: sequences s and t
    output: similarity between s and t
    m ← |s|
    n ← |t|
    for i ← 0 to m do
        a[i, 0] ← i × g
    for j ← 0 to n do
        a[0, j] ← j × g
    for i ← 1 to m do
        for j ← 1 to n do
            a[i, j] ← max(a[i − 1, j] + g,
                          a[i − 1, j − 1] + p(i, j),
                          a[i, j − 1] + g)
    return a[m, n]
```

**FIGURE 3.2**

*Basic dynamic programming algorithm for comparison of two sequences.*

(usually $g < 0$) and on a scoring function $p$ for pairs of characters. We have used $g = -2$, $p(a, b) = 1$ if $a = b$, and $p(a, b) = -1$ if $a \neq b$ in Figure 3.1.

## Optimal Alignments

We saw how to compute the similarity between two sequences. Here we will see how to construct an optimal alignment between them. The arrows in Figure 3.1 will be useful in this respect. All we need to do is start at entry $(m, n)$ and follow the arrows until we get to $(0, 0)$. Each arrow used will give us one column of the alignment. In fact, consider an arrow leaving entry $(i, j)$. If this arrow is horizontal, it corresponds to a column with a space in $s$ matched with $t[j]$; if it is vertical, it corresponds to $s[i]$ matched with a space in $t$; finally, a diagonal arrow means $s[i]$ matched with $t[j]$. Observe that the first sequence, $s$, is always placed along the vertical edge. Thus, an optimal alignment can be easily constructed from right to left if we have the matrix $a$ computed by the basic algorithm. It is not necessary to implement the arrows explicitly — a simple test can be used to choose the next entry to visit.

Figure 3.3 shows a recursive algorithm for determining an optimal alignment given the matrix $a$ and sequences $s$ and $t$. The call *Align*$(m, n, len)$ will construct an optimal alignment. The answer will be given in a pair of vectors *align-s* and *align-t* that will hold in the positions $1..len$ the aligned characters, which can be either spaces or symbols from the sequences. The variables *align-s* and *align-t* are treated as globals in the code. The

```
Algorithm Align
    input: indices i, j, array a given by algorithm Similarity
    output: alignment in align-s, align-t, and length in len
    if i = 0 and j = 0 then
        len ← 0
    else if i > 0 and a[i, j] = a[i − 1, j] + g then
        Align(i − 1, j, len)
        len ← len + 1
        align-s[len] ← s[i]
        align-t[len] ← -
    else if i > 0 and j > 0 and a[i, j] = a[i − 1, j − 1] + p(i, j) then
        Align(i − 1, j − 1, len)
        len ← len + 1
        align-s[len] ← s[i]
        align-t[len] ← t[j]
    else  // has to be j > 0 and a[i, j] = a[i, j − 1] + g
        Align(i, j − 1, len)
        len ← len + 1
        align-s[len] ← -
        align-t[len] ← t[j]
```

**FIGURE 3.3**

*Recursive algorithm for optimal alignment.*

length of the alignment is also returned by the algorithm in the parameter *len*. Note that $\max(|s|, |t|) \leq len \leq m + n$.

As we said previously, many optimal alignments may exist for a given pair of sequences. The algorithm of Figure 3.3 returns just one of them, giving preference to the edges leaving $(i, j)$ in counterclockwise order (see Figure 3.4).

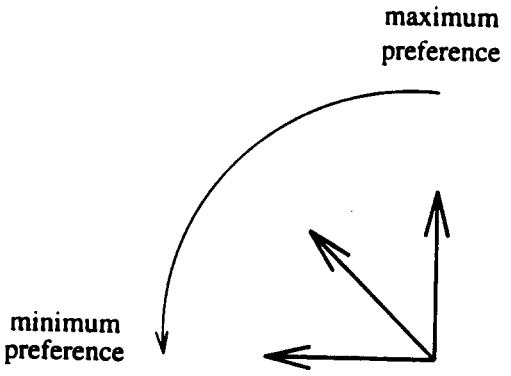


**FIGURE 3.4**

*Arrow preference.*

As a result, the optimal alignment returned by this algorithm has the following general characteristic: When there is choice, a column with a space in $t$ has precedence over a column with two symbols, which in turn has precedence over a column with a space in $s$. For instance, when aligning $s =$ ATAT with $t =$ TATA, we get

```
-ATAT
TATA-
```

rather than

```
ATAT-
-TATA
```

which is the other optimal alignment for these two sequences. Similarly, when aligning $s =$ AA with $t =$ AAAA, we get

```
--AA
AAAA
```

although there are five other optimal alignments. This alignment is sometimes referred to as the *upmost* alignment because it uses the arrows higher up in the matrix. To reverse these preferences, we would reverse the order of the **if** statements in the code, obtaining the *downmost* alignment in this case. A column appearing in both the upmost and downmost alignments will be present in all optimal alignments between the two sequences considered.

It is possible to modify the algorithm to produce *all* optimal alignments between $s$ and $t$. We need to keep a stack with the points at which there are options and backtrack to them to explore all possibilities of reaching (0, 0) through the arrows. However, there might be a very large number of optimal alignments. In these cases, it is often advisable

to keep some sort of short representation of all or part of these alignments, for instance, the upmost and downmost ones.

Let us now determine the complexity of the algorithms described in this section. The basic algorithm of Figure 3.2 has four loops. The first two do the initialization and consume time $O(m)$ and $O(n)$, respectively. The last two loops are nested and fill the rest of the matrix. The number of operations performed depends essentially on the number of entries that must be computed, that is, the size of the matrix. Thus, we spend time $O(mn)$ in this part and this is the dominant term in the time complexity. The space used is also proportional to the size of the matrix. Hence, the complexity of the basic algorithm is $O(mn)$ both for time and space. If the sequences have the same or nearly the same length, say $n$, we get $O(n^2)$. That is why we say that these algorithms have quadratic complexity.

The construction of the alignment — given the already filled matrix — is done in time $O(len)$, where *len* is the size of the returned alignment, which is $O(m+n)$.

### 3.2.2 LOCAL COMPARISON

A *local alignment* between $s$ and $t$ is an alignment between a substring of $s$ and a substring of $t$. In this section we present an algorithm to find the highest scoring local alignments between two sequences.

This algorithm is a variation of the basic algorithm. The main data structure is, as before, an $(m+1) \times (n+1)$ array. Only this time the interpretation of the array values is different. Each entry $(i, j)$ will hold the highest score of an alignment between a *suffix* of $s[1..i]$ and a *suffix* of $t[1..j]$. The first row and the first column are initialized with zeros.

For any entry $(i, j)$, there is always the alignment between the empty suffixes of $s[1..i]$ and $t[1..j]$, which has score zero; therefore the array will have all entries greater than or equal to zero. This explains in part the initialization above.

Following initialization, the array can be filled in the usual way, with $a[i, j]$ depending on the value of three previously computed entries. The resulting recurrence is

$$a[i, j] = \max \begin{cases} a[i, j-1] + g \\ a[i-1, j-1] + p(i, j) \\ a[i-1, j] + g \\ 0, \end{cases}$$

that is, the same as in the basic algorithm, except that now we have a fourth possibility, not available in the global case, of an empty alignment.

In the end, it suffices to find the maximum entry in the whole array. This will be the score of an optimal local alignment. Any entry containing this value can be used as a starting point to get such an alignment. The rest of the alignment is obtained tracing back as usual, but stopping as soon as we reach an entry with no arrow going out. Alternatively, we can stop as soon as we reach an entry with value zero.

In general, when doing local comparison, we are interested not only in the optimal alignments, but also in near optimal alignments with scores above a certain threshold. References to methods for retrieving near optimal alignments are given in the bibliographic notes.

### 3.2.3 SEMIGLOBAL COMPARISON

In a semiglobal comparison, we score alignments ignoring some of the *end spaces* in the sequences. An interesting characteristic of the basic dynamic programming algorithm is that we can control the penalty associated with end spaces by doing very simple modifications to the original scheme.

Let us begin by defining precisely what we mean by *end spaces* and why it might be better to let them be included for free in certain situations. End spaces are those that appear before the first or after the last character in a sequence. For instance, all the spaces in the second sequence in the alignment below are end spaces, while the single space in the first sequence is not an end space.

```
CAGCA-CTTGGATTCTCGG
---CAGCGTGG--------        (3.4)
```

Notice that the lengths of these two sequences differ considerably. One has size 8, and the other has 18 characters. When this happens, there will be many spaces in any alignment, giving a large negative contribution to the score. Nevertheless, if we ignore end spaces, the alignment is pretty good, with 6 matches, 1 mismatch, and 1 space.

Observe that this is not the best alignment between these sequences. In alignment (3.5) below we present another alignment with a higher score ($-12$ against $-19$ of the previous one) according to the scoring system we have been using so far.

```
CAGCACTTGGATTCTCGG
CAGC-----G-T----GG        (3.5)
```

In spite of having scored higher and having matched all characters of the second sequence with identical characters in the first one, this alignment is not so interesting from the point of view of finding similar regions in the sequences. The second sequence was simply torn apart brutally by the spaces just for the sake of matching exactly its characters. If we are looking for regions of the longer sequence that are approximately the same as the shorter sequence, then undoubtedly the first alignment (3.4) is more to the point. This is reflected in the scores obtained when we disregard (that is, not charge for) end spaces: (3.4) gets 3 points against the same $-12$ for (3.5).

Let us now describe a variation of the basic algorithm that will ignore end spaces. Consider initially the case where we do not want to charge for spaces after the last character of $s$. Take an optimal alignment in this case. The spaces after the end of $s$ are matched to a suffix of $t$. If we remove this final part of the alignment, the remaining is an alignment between $s$ and a prefix of $t$, with score equal to the original alignment. Therefore, to get the score of the optimal alignment between $s$ and $t$ without charge for spaces after the end of $s$, all we need to do is to find the best similarity between $s$ and a prefix of $t$. But we saw in Section 3.2.1 that the entry $(i, j)$ of matrix $a$ contains the similarity between $s[1..i]$ and $t[1..j]$. Hence, it suffices to take the maximum value in the last row of $a$, that is,

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

Notice that in this section we have changed the definition of $\text{sim}(s, t)$ so that it now indicates similarity, ignoring final spaces in $s$. The expression above gives the score of the

best alignment. To recover the alignment itself, we proceed just as in the basic algorithm, but starting at $(m, k)$ where $k$ is such that $\text{sim}(s, t) = a[m, k]$.

An analogous argument solves the case in which we do not charge for final spaces in $t$. We take the maximum along the last column of $a$ in this case. We can even combine the two ideas and seek the best alignment without charging for final spaces in either sequence. The answer will be found by taking the maximum along the border of the matrix formed by the union of the last row and the last column. In all cases, to recover an optimal alignment, we start at an array entry that contains the similarity value and follow the arrows until we reach (0, 0). Each arrow will give one column of the alignment as in the basic case.

Now let us turn our attention to the case of initial spaces. Suppose that we want the best alignment that does not charge for initial spaces in $s$. This is equivalent to the best alignment between $s$ and a suffix of $t$. To get the desired answer, we use an $(m + 1) \times (n + 1)$ array just as in the basic algorithm, but with a slight difference. Each entry $(i, j)$ now will contain the highest similarity between $s[1..i]$ and a suffix of a prefix, which is to say a suffix of $t[1..j]$.

Doing that, it is clear that $a[m, n]$ will be the answer. What is less clear, but nevertheless true, is that the array can be filled in using exactly the same formula as in the basic algorithm! That is Equation (3.3). The initialization will be different, however. The first row must be initialized with zeros instead of multiples of the space penalty because of the new meaning of the entries. We leave it to the reader to verify that Equation (3.3) works in this case.

We can apply the same trick and initialize the first column with zeros, and by doing this we will be forgiving spaces before the beginning of $t$. If in addition we initialize both the first row and the first column with zeros, and proceed with Equation (3.3) for the other entries, we will be computing in each entry the highest similarity between a suffix of $s$ and a suffix of $t$. To find an optimal alignment, we follow the arrows from the maximum value entry until we reach one of the borders initialized with zeros, and then follow the border back to the origin.

Table 3.1 summarizes these variations. There are four places where we may not want to charge for spaces: beginning or end of $s$, and beginning or end of $t$. We can combine these conditions independently in any way and use the variations above to find the similarity. The only things that change are the initialization and where to look for the maximum value. Forgiving initial spaces translates into initializing certain positions with zero. Forgiving final spaces means looking for the maximum along certain positions. But filling in the array is always the same process, using Equation (3.3).

**TABLE 3.1**

*Summary of end space charging procedures.*

| *Place where spaces are not charged for* | *Action* |
|---|---|
| Beginning of first sequence | Initialize first row with zeros |
| End of first sequence | Look for maximum in last row |
| Beginning of second sequence | Initialize first column with zeros |
| End of second sequence | Look for maximum in last column |