As always, we can recover the secondary structure itself (not just its value) by recording how the minima in (6.13) are achieved and tracing back through the computation.

6.6 Sequence Alignment

For the remainder of this chapter, we consider two further dynamic programming algorithms that each have a wide range of applications. In the next two sections we discuss *sequence alignment*, a fundamental problem that arises in comparing strings. Following this, we turn to the problem of computing shortest paths in graphs when edges have costs that may be negative.

The Problem

Dictionaries on the Web seem to get more and more useful: often it seems easier to pull up a bookmarked online dictionary than to get a physical dictionary down from the bookshelf. And many online dictionaries offer functions that you can't get from a printed one: if you're looking for a definition and type in a word it doesn't contain—say, *ocurrance*—it will come back and ask, "Perhaps you mean *occurrence*?" How does it do this? Did it truly know what you had in mind?

Let's defer the second question to a different book and think a little about the first one. To decide what you probably meant, it would be natural to search the dictionary for the word most "similar" to the one you typed in. To do this, we have to answer the question: How should we define similarity between two words or strings?

Intuitively, we'd like to say that *ocurrance* and *occurrence* are similar because we can make the two words identical if we add a *c* to the first word and change the *a* to an *e*. Since neither of these changes seems so large, we conclude that the words are quite similar. To put it another way, we can *nearly* line up the two words letter by letter:

```
o-currance
occurrence
```

The hyphen (-) indicates a *gap* where we had to add a letter to the second word to get it to line up with the first. Moreover, our lining up is not perfect in that an *e* is lined up with an *a*.

We want a model in which similarity is determined roughly by the number of gaps and mismatches we incur when we line up the two words. Of course, there are many possible ways to line up the two words; for example, we could have written

```
o-curr-ance
occurre-nce
```

which involves three gaps and no mismatches. Which is better: one gap and one mismatch, or three gaps and no mismatches?

This discussion has been made easier because we know roughly what the correspondence ought to look like. When the two strings don't look like English words—for example, abbbaabbbbaab and ababaaabbbbbab—it may take a little work to decide whether they can be lined up nicely or not:

```
abbbaa--bbbbaab
ababaaabbbbba-b
```

Dictionary interfaces and spell-checkers are not the most computationally intensive application for this type of problem. In fact, determining similarities among strings is one of the central computational problems facing molecular biologists today.

Strings arise very naturally in biology: an organism's *genome*—its full set of genetic material—is divided up into giant linear DNA molecules known as *chromosomes,* each of which serves conceptually as a one-dimensional chemical storage device. Indeed, it does not obscure reality very much to think of it as an enormous linear *tape*, containing a string over the alphabet $\{A, C, G, T\}$. The string of symbols encodes the instructions for building protein molecules; using a chemical mechanism for reading portions of the chromosome, a cell can construct proteins that in turn control its metabolism.

Why is similarity important in this picture? To a first approximation, the sequence of symbols in an organism's genome can be viewed as determining the properties of the organism. So suppose we have two strains of bacteria, X and Y, which are closely related evolutionarily. Suppose further that we've determined that a certain substring in the DNA of X codes for a certain kind of toxin. Then, if we discover a very "similar" substring in the DNA of Y, we might be able to hypothesize, before performing any experiments at all, that this portion of the DNA in Y codes for a similar kind of toxin. This use of computation to guide decisions about biological experiments is one of the hallmarks of the field of *computational biology*.

All this leaves us with the same question we asked initially, while typing badly spelled words into our online dictionary: How should we define the notion of *similarity* between two strings?

In the early 1970s, the two molecular biologists Needleman and Wunsch proposed a definition of similarity, which, basically unchanged, has become the standard definition in use today. Its position as a standard was reinforced by its simplicity and intuitive appeal, as well as through its independent discovery by several other researchers around the same time. Moreover, this definition of similarity came with an efficient dynamic programming algorithm to compute it. In this way, the paradigm of dynamic programming was independently discovered by biologists some twenty years after mathematicians and computer scientists first articulated it.

The definition is motivated by the considerations we discussed above, and in particular by the notion of "lining up" two strings. Suppose we are given two strings X and Y, where X consists of the sequence of symbols $x_1x_2 \cdots x_m$ and Y consists of the sequence of symbols $y_1y_2 \cdots y_n$. Consider the sets $\{1, 2, \ldots, m\}$ and $\{1, 2, \ldots, n\}$ as representing the different positions in the strings X and Y, and consider a matching of these sets; recall that a *matching* is a set of ordered pairs with the property that each item occurs in at most one pair. We say that a matching M of these two sets is an *alignment* if there are no "crossing" pairs: if $(i, j), (i', j') \in M$ and $i < i'$, then $j < j'$. Intuitively, an alignment gives a way of lining up the two strings, by telling us which pairs of positions will be lined up with one another. Thus, for example,

```
stop-
-tops
```

corresponds to the alignment $\{(2, 1), (3, 2), (4, 3)\}$.

Our definition of similarity will be based on finding the *optimal* alignment between X and Y, according to the following criteria. Suppose M is a given alignment between X and Y.

- First, there is a parameter $\delta > 0$ that defines a *gap penalty*. For each position of X or Y that is not matched in M—it is a *gap*—we incur a cost of δ.
- Second, for each pair of letters p, q in our alphabet, there is a *mismatch cost* of α_{pq} for lining up p with q. Thus, for each $(i, j) \in M$, we pay the appropriate mismatch cost $\alpha_{x_iy_j}$ for lining up x_i with y_j. One generally assumes that $\alpha_{pp} = 0$ for each letter p—there is no mismatch cost to line up a letter with another copy of itself—although this will not be necessary in anything that follows.
- The *cost* of M is the sum of its gap and mismatch costs, and we seek an alignment of minimum cost.

The process of minimizing this cost is often referred to as *sequence alignment* in the biology literature. The quantities δ and $\{\alpha_{pq}\}$ are external parameters that must be plugged into software for sequence alignment; indeed, a lot of work goes into choosing the settings for these parameters. From our point of view, in designing an algorithm for sequence alignment, we will take them as given. To go back to our first example, notice how these parameters determine which alignment of *ocurrance* and *occurrence* we should prefer: the first is strictly better if and only if $\delta + \alpha_{ae} < 3\delta$.

Designing the Algorithm

We now have a concrete numerical definition for the similarity between strings X and Y: it is the minimum cost of an alignment between X and Y. The lower this cost, the more similar we declare the strings to be. We now turn to the problem of computing this minimum cost, and an optimal alignment that yields it, for a given pair of strings X and Y.

One of the approaches we could try for this problem is dynamic programming, and we are motivated by the following basic dichotomy.

- In the optimal alignment M, either $(m, n) \in M$ or $(m, n) \notin M$. (That is, either the last symbols in the two strings are matched to each other, or they aren't.)

By itself, this fact would be too weak to provide us with a dynamic programming solution. Suppose, however, that we compound it with the following basic fact.

(6.14) *Let M be any alignment of X and Y. If $(m, n) \notin M$, then either the m^{th} position of X or the n^{th} position of Y is not matched in M.*

Proof. Suppose by way of contradiction that $(m, n) \notin M$, and there are numbers $i < m$ and $j < n$ so that $(m, j) \in M$ and $(i, n) \in M$. But this contradicts our definition of *alignment*: we have $(i, n), (m, j) \in M$ with $i < m$, but $n > i$ so the pairs (i, n) and (m, j) cross. ■

There is an equivalent way to write (6.14) that exposes three alternative possibilities, and leads directly to the formulation of a recurrence.

(6.15) *In an optimal alignment M, at least one of the following is true:*

(i) $(m, n) \in M$*; or*

(ii) the m^{th} position of X is not matched; or

(iii) the n^{th} position of Y is not matched.

Now, let $\text{OPT}(i, j)$ denote the minimum cost of an alignment between $x_1x_2 \cdots x_i$ and $y_1y_2 \cdots y_j$. If case (i) of (6.15) holds, we pay $\alpha_{x_m y_n}$ and then align $x_1x_2 \cdots x_{m-1}$ as well as possible with $y_1y_2 \cdots y_{n-1}$; we get $\text{OPT}(m, n) = \alpha_{x_m y_n} + \text{OPT}(m-1, n-1)$. If case (ii) holds, we pay a gap cost of δ since the m^{th} position of X is not matched, and then we align $x_1x_2 \cdots x_{m-1}$ as well as possible with $y_1y_2 \cdots y_n$. In this way, we get $\text{OPT}(m, n) = \delta + \text{OPT}(m-1, n)$. Similarly, if case (iii) holds, we get $\text{OPT}(m, n) = \delta + \text{OPT}(m, n-1)$.

Using the same argument for the subproblem of finding the minimum-cost alignment between $x_1x_2 \cdots x_i$ and $y_1y_2 \cdots y_j$, we get the following fact.

(6.16) *The minimum alignment costs satisfy the following recurrence for $i \geq 1$ and $j \geq 1$:*

$$\text{OPT}(i, j) = \min[\alpha_{x_iy_j} + \text{OPT}(i-1, j-1), \delta + \text{OPT}(i-1, j), \delta + \text{OPT}(i, j-1)].$$

Moreover, (i, j) is in an optimal alignment M for this subproblem if and only if the minimum is achieved by the first of these values.

We have maneuvered ourselves into a position where the dynamic programming algorithm has become clear: We build up the values of $\text{OPT}(i, j)$ using the recurrence in (6.16). There are only $O(mn)$ subproblems, and $\text{OPT}(m, n)$ is the value we are seeking.

We now specify the algorithm to compute the value of the optimal alignment. For purposes of initialization, we note that $\text{OPT}(i, 0) = \text{OPT}(0, i) = i\delta$ for all i, since the only way to line up an i-letter word with a 0-letter word is to use i gaps.

```
Alignment($X$,$Y$)
  Array $A[0 \ldots m, 0 \ldots n]$
  Initialize $A[i, 0] = i\delta$ for each $i$
  Initialize $A[0, j] = j\delta$ for each $j$
  For $j = 1, \ldots, n$
     For $i = 1, \ldots, m$
         Use the recurrence (6.16) to compute $A[i, j]$
     Endfor
  Endfor
  Return $A[m, n]$
```

As in previous dynamic programming algorithms, we can trace back through the array A, using the second part of fact (6.16), to construct the alignment itself.

Analyzing the Algorithm

The correctness of the algorithm follows directly from (6.16). The running time is $O(mn)$, since the array A has $O(mn)$ entries, and at worst we spend constant time on each.

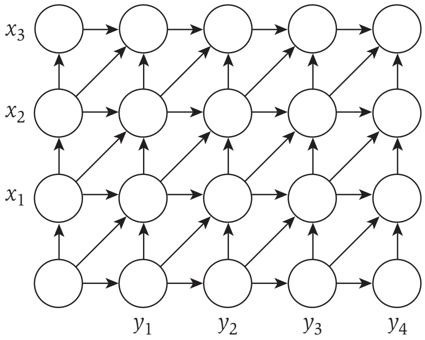


Figure 6.17 A graph-based picture of sequence alignment.

There is an appealing pictorial way in which people think about this sequence alignment algorithm. Suppose we build a two-dimensional $m \times n$ grid graph G_{XY}, with the rows labeled by symbols in the string X, the columns labeled by symbols in Y, and directed edges as in Figure 6.17.

We number the rows from 0 to m and the columns from 0 to n; we denote the node in the i^{th} row and the j^{th} column by the label (i, j). We put *costs* on the edges of G_{XY}: the cost of each horizontal and vertical edge is δ, and the cost of the diagonal edge from $(i-1, j-1)$ to (i, j) is $\alpha_{x_i y_j}$.

The purpose of this picture now emerges: the recurrence in (6.16) for $\text{OPT}(i, j)$ is precisely the recurrence one gets for the minimum-cost path in G_{XY} from $(0, 0)$ to (i, j). Thus we can show

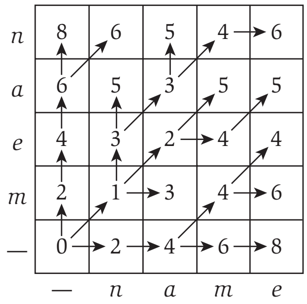
(6.17) *Let $f(i, j)$ denote the minimum cost of a path from $(0, 0)$ to (i, j) in G_{XY}. Then for all i, j, we have $f(i, j) = \text{OPT}(i, j)$.*

Proof. We can easily prove this by induction on $i + j$. When $i + j = 0$, we have $i = j = 0$, and indeed $f(i, j) = \text{OPT}(i, j) = 0$.

Now consider arbitrary values of i and j, and suppose the statement is true for all pairs (i', j') with $i' + j' < i + j$. The last edge on the shortest path to (i, j) is either from $(i-1, j-1)$, $(i-1, j)$, or $(i, j-1)$. Thus we have

$$\begin{aligned} f(i, j) &= \min[\alpha_{x_i y_j} + f(i-1, j-1), \delta + f(i-1, j), \delta + f(i, j-1)] \\ &= \min[\alpha_{x_i y_j} + \text{OPT}(i-1, j-1), \delta + \text{OPT}(i-1, j), \delta + \text{OPT}(i, j-1)] \\ &= \text{OPT}(i, j), \end{aligned}$$

where we pass from the first line to the second using the induction hypothesis, and we pass from the second to the third using (6.16). ■



n	8	6	5	4	6
a	6	5	3	5	5
e	4	3	2	4	4
m	2	1	3	4	6
—	0	2	4	6	8
	—	*n*	*a*	*m*	*e*

Figure 6.18 The OPT values for the problem of aligning the words *mean* to *name*.

Thus the value of the optimal alignment is the length of the shortest path in G_{XY} from $(0, 0)$ to (m, n). (We'll call any path in G_{XY} from $(0, 0)$ to (m, n) a *corner-to-corner path*.) Moreover, the diagonal edges used in a shortest path correspond precisely to the pairs used in a minimum-cost alignment. These connections to the Shortest-Path Problem in the graph G_{XY} do not directly yield an improvement in the running time for the sequence alignment problem; however, they do help one's intuition for the problem and have been useful in suggesting algorithms for more complex variations on sequence alignment.

For an example, Figure 6.18 shows the value of the shortest path from $(0, 0)$ to each node (i, j) for the problem of aligning the words *mean* and *name*. For the purpose of this example, we assume that $\delta = 2$; matching a vowel with a different vowel, or a consonant with a different consonant, costs 1; while matching a vowel and a consonant with each other costs 3. For each cell in the table (representing the corresponding node), the arrow indicates the last step of the shortest path leading to that node—in other words, the way that the minimum is achieved in (6.16). Thus, by following arrows backward from node $(4, 4)$, we can trace back to construct the alignment.

6.7 Sequence Alignment in Linear Space via Divide and Conquer

In the previous section, we showed how to compute the optimal alignment between two strings X and Y of lengths m and n, respectively. Building up the two-dimensional m-by-n array of optimal solutions to subproblems, $\text{OPT}(\cdot, \cdot)$, turned out to be equivalent to constructing a graph G_{XY} with mn nodes laid out in a grid and looking for the cheapest path between opposite corners. In either of these ways of formulating the dynamic programming algorithm, the running time is $O(mn)$, because it takes constant time to determine the value in each of the mn cells of the array OPT; and the space requirement is $O(mn)$ as well, since it was dominated by the cost of storing the array (or the graph G_{XY}).

The Problem

The question we ask in this section is: Should we be happy with $O(mn)$ as a space bound? If our application is to compare English words, or even English sentences, it is quite reasonable. In biological applications of sequence alignment, however, one often compares very long strings against one another; and in these cases, the $\Theta(mn)$ space requirement can potentially be a more severe problem than the $\Theta(mn)$ time requirement. Suppose, for example, that we are comparing two strings of 100,000 symbols each. Depending on the underlying processor, the prospect of performing roughly 10 billion primitive