Dynamic Programming Algorithms

We introduced dynamic programming in chapter 2 with the Rocks problem. While the Rocks problem does not appear to be related to bioinformatics, the algorithm that we described is a computational twin of a popular alignment algorithm for sequence comparison. Dynamic programming provides a framework for understanding DNA sequence comparison algorithms, many of which have been used by biologists to make important inferences about gene function and evolutionary history. We will also apply dynamic programming to gene finding and other bioinformatics problems.

The Power of DNA Sequence Comparison

After a new gene is found, biologists usually have no idea about its function. A common approach to inferring a newly sequenced gene's function is to find similarities with genes of known function. A striking example of such a biological discovery made through a similarity search happened in 1984 when scientists used a simple computational technique to compare the newly discovered cancer-causing ν-sis oncogene with all (at the time) known genes. To their astonishment, the cancer-causing gene matched a normal gene involved in growth and development called platelet-derived growth factor (PDGF).[1] After discovering this similarity, scientists became suspicious that cancer might be caused by a normal growth gene being switched on at the wrong time—in essence, a good gene doing the right thing at the wrong time.

1. *Oncogenes* are genes in viruses that cause a cancer-like transformation of infected cells. Oncogene ν-sis in the *simian sarcoma virus* causes uncontrolled cell growth and leads to cancer in monkeys. The seemingly unrelated *growth factor* PDGF is a protein that stimulates cell growth.

Another example of a successful similarity search was the discovery of the cystic fibrosis gene. Cystic fibrosis is a fatal disease associated with abnormal secretions, and is diagnosed in children at a rate of 1 in 3900. A defective gene causes the body to produce abnormally thick mucus that clogs the lungs and leads to lifethreatening lung infections. More than 10 million Americans are unknowing and symptomless carriers of the defective cystic fibrosis gene; each time two carriers have a child, there is a 25% chance that the child will have cystic fibrosis.

In 1989 the search for the cystic fibrosis gene was narrowed to a region of 1 million nucleotides on the chromosome 7, but the exact location of the gene remained unknown. When the area around the cystic fibrosis gene was sequenced, biologists compared the region against a database of all known genes, and discovered similarities between some segment within this region and a gene that had already been discovered, and was known to code for *adenosine triphosphate (ATP) binding proteins*.[2] These proteins span the cell membrane multiple times as part of the ion transport channel; this seemed a plausible function for a cystic fibrosis gene, given the fact that the disease involves sweat secretions with abnormally high sodium content. As a result, the similarity analysis shed light on a damaged mechanism in faulty cystic fibrosis genes.

Establishing a link between cancer-causing genes and normal growth genes and elucidating the nature of cystic fibrosis were only the first success stories in sequence comparison. Many applications of sequence comparison algorithms quickly followed, and today bioinformatics approaches are among the dominant techniques for the discovery of gene function.

This chapter describes algorithms that allow biologists to reveal the similarity between different DNA sequences. However, we will first show how dynamic programming can yield a faster algorithm to solve the Change problem.

The Change Problem Revisited

We introduced the Change problem in chapter 2 as the problem of changing an amount of money M into the smallest number of coins from denominations $\mathbf{c} = (c_1, c_2, \ldots, c_d)$. We showed that the naive greedy solution used by cashiers everywhere is not actually a correct solution to this problem, and ended with a correct—though slow—brute force algorithm. We will con-

2. ATP binding proteins provide energy for many reactions in the cell.

sider a slightly modified version of the Change problem, in which we do not concern ourselves with the actual combination of coins that make up the optimal change solution. Instead, we only calculate the smallest number of coins needed (it is easy to modify this algorithm to also return the coin combination that achieves that number).

Suppose you need to make change for 77 cents and the only coin denominations available are 1, 3, and 7 cents. The best combination for 77 cents will be one of the following:

- the best combination for $77 - 1 = 76$ cents, plus a 1-cent coin;
- the best combination for $77 - 3 = 74$ cents, plus a 3-cent coin;
- the best combination for $77 - 7 = 70$ cents, plus a 7-cent coin.

For 77 cents, the best combination would be the smallest of the above three choices. The same logic applies to 76 cents (best of 75, 73, or 69 cents), and so on (fig. 6.1). If $bestNumCoins_M$ is the smallest number of coins needed to change M cents, then the following recurrence relation holds:

$$bestNumCoins_M = \min \begin{cases} bestNumCoins_{M-1} + 1 \\ bestNumCoins_{M-3} + 1 \\ bestNumCoins_{M-7} + 1 \end{cases}$$

In the more general case of d denominations $\mathbf{c} = (c_1, \ldots, c_d)$:

$$bestNumCoins_M = \min \begin{cases} bestNumCoins_{M-c_1} + 1 \\ bestNumCoins_{M-c_2} + 1 \\ \vdots \\ bestNumCoins_{M-c_d} + 1 \end{cases}$$

This recurrence motivates the following algorithm:

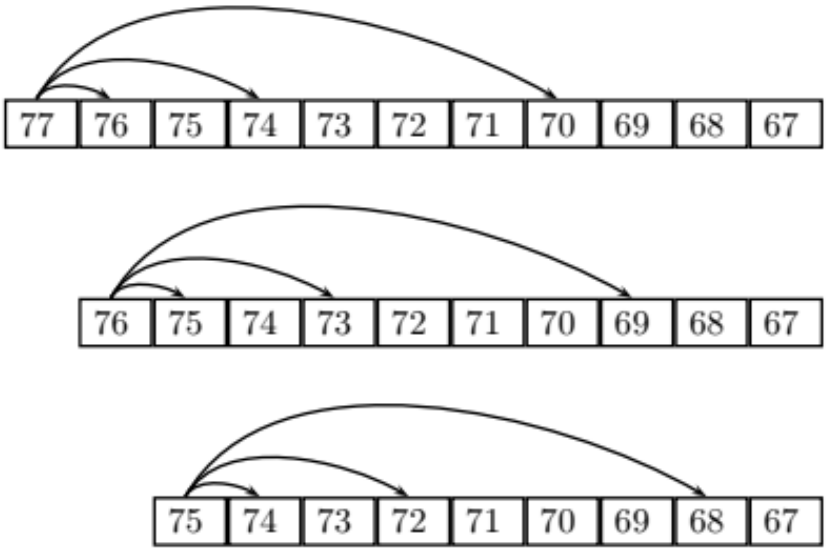


Figure 6.1 The relationships between optimal solutions in the Change problem. The smallest number of coins for 77 cents depends on the smallest number of coins for 76, 74, and 70 cents; the smallest number of coins for 76 cents depends on the smallest number of coins for 75, 73, and 69 cents, and so on.

```
RECURSIVECHANGE(M, c, d)
1  if M = 0
2      return 0
3  bestNumCoins ← ∞
4  for i ← 1 to d
5      if M ≥ c_i
6          numCoins ← RECURSIVECHANGE(M − c_i, c, d)
7          if numCoins + 1 < bestNumCoins
8              bestNumCoins ← numCoins + 1
9  return bestNumCoins
```

The sequence of calls that RECURSIVECHANGE makes has a feature in common with the sequence of calls made by RECURSIVEFIBONACCI, namely, that RECURSIVECHANGE recalculates the optimal coin combination for a given amount of money repeatedly. For example, the optimal coin combination for 70 cents is recomputed repeatedly nine times over and over as $(77-7)$, $(77-3-3-1)$, $(77-3-1-3)$, $(77-1-3-3)$, $(77-3-1-1-1-1)$, $(77-1-3-1-1-1)$, $(77-1-1-3-1-1)$, $(77-1-1-1-3-1)$, $(77-1-1-1-1-3)$, and $(77-1-1-1-1-1-1-1)$. The optimal

coin combination for 20 cents will be recomputed billions of times rendering RECURSIVECHANGE impractical.

To improve RECURSIVECHANGE, we can use the same strategy as we did for the Fibonacci problem—all we really need to do is use the fact that the solution for M relies on solutions for $M - c_1$, $M - c_2$, and so on, and then reverse the order in which we solve the problem. This allows us to leverage previously computed solutions to form solutions to larger problems and avoid all this recomputation.

Instead of trying to find the minimum number of coins to change M cents, we attempt the superficially harder task of doing this for *each* amount of money, m, from 0 to M. This appears to require more work, but in fact, it simplifies matters. The following algorithm with running time $O(Md)$ calculates $bestNumCoins_m$ for increasing values of m. This works because the best number of coins for some value m depends only on values less than m.

```
DPCHANGE(M, c, d)
1  bestNumCoins_0 ← 0
2  for m ← 1 to M
3      bestNumCoins_m ← ∞
4      for i ← 1 to d
5          if m ≥ c_i
6              if bestNumCoins_{m−c_i} + 1 < bestNumCoins_m
7                  bestNumCoins_m ← bestNumCoins_{m−c_i} + 1
8  return bestNumCoins_M
```

The key difference between RECURSIVECHANGE and DPCHANGE is that the first makes d recursive calls to compute the best change for M (and each of these calls requires a lot of work!), while the second analyzes the d already precomputed values to almost instantly compute the new one. As surprising as it may sound, simply reversing the order of computations in figure 6.1 makes a dramatic difference in efficiency (fig. 6.2).

We stress again the difference between the complexity of a problem and the complexity of an algorithm. In particular, we initially showed an $O(M^d)$ algorithm to solve the Change problem, and there did not appear to be any easy way to remedy this situation. Yet the DPCHANGE algorithm provides a simple $O(Md)$ solution. Conversely, a minor modification of the Change problem renders the problem very difficult. Suppose you had a limited number of each denomination and needed to change M cents using no more than the provided supply of each coin. Since you have fewer possible choices in

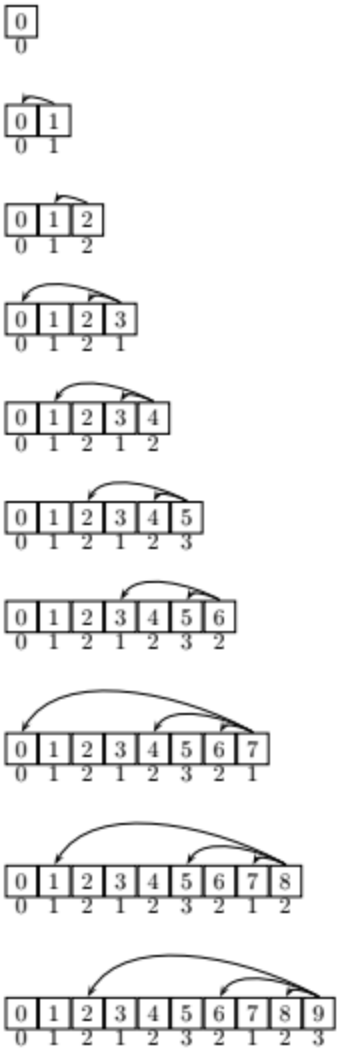


Figure 6.2 The solution for 9 cents ($bestNumCoins_9$) depends on 8 cents, 6 cents and 2 cent, but the smallest number of coins can be obtained by computing $bestNumCoins_m$ for $0 \leq m \leq 9$.

this new problem, it would seem to require even less time than the original Change problem, and that a minor modification to DPCHANGE would work. However, this is not the case and this problem turns out to be very difficult.

The Manhattan Tourist Problem

We will further illustrate dynamic programming with a surprisingly useful toy problem, called the Manhattan Tourist problem, and then build on this intuition to describe DNA sequence alignment.

Imagine a sightseeing tour in the borough of Manhattan in New York City, where a group of tourists are determined to walk from the corner of 59th Street and 8th Avenue to the Chrysler Building at 42nd Street and Lexington Avenue. There are many attractions along the way, but assume for the moment that the tourists want to see as many attractions as possible. The tourists are allowed to move either to the south or to the east, but even so, they can choose from many different paths (exactly how many is left as a problem at the end of the chapter). The upper path in figure 6.3 will take the tourists to the Museum of Modern Art, but they will have to miss Times Square; the bottom path will allow the tourists to see Times Square, but they will have to miss the Museum of Modern Art.

The map above can also be represented as a gridlike structure (figure 6.4) with the numbers next to each line (called *weights*) showing the number of attractions on every block. The tourists must decide among the many possible paths between the northwesternmost point (called the *source vertex*) and the southeasternmost point (called the *sink vertex*). The weight of a path from the source to the sink is simply the sum of weights of its edges, or the overall number of attractions. We will refer to this kind of construct as a *graph*, the intersections of streets we will call *vertices*, and the streets themselves will be *edges* and have a weight associated with them. We assume that horizontal edges in the graph are oriented to the east like $\rightarrow$ while vertical edges are oriented to the south like $\downarrow$. A *path* is a continuous sequence of edges, and the *length of a path* is the sum of the edge weights in the path.[3] A more detailed discussion of graphs can be found in chapter 8.

Although the upper path in figure 6.3 is better than the bottom one, in the sense that the tourists will see more attractions, it is not immediately clear if there is an even better path in the grid. The Manhattan Tourist problem is to find the path with the maximum number of attractions,[4] that is, a *longest path*

3. We emphasize that the length of paths in the graph represent the overall number of attractions on this path and has nothing to do with the real length of the path (in miles), that is, the distance the tourists travel.

4. There are many interesting museums and architectural landmarks in Manhattan. However, it is impossible to please everyone, so one can change the relative importance of the types of attractions by modulating the weights on the edges in the graph. This flexibility in assigning weights will become important when we discuss *scoring matrices* for sequence comparison.

(a path of maximum overall weight) in the grid.

Manhattan Tourist Problem:
Find a longest path in a weighted grid.

Input: A weighted grid G with two distinguished vertices: a *source* and a *sink*.

Output: A longest path in G from *source* to *sink*.

Note that, since the tourists only move south and east, any grid positions west or north of the source are unusable. Similarly, any grid positions south or east of the sink are unusable, so we can simply say that the source vertex is at $(0, 0)$ and that the sink vertex at (n, m) defines the southeasternmost corner of the grid. In figure 6.4 $n = m = 4$, but n does not always have to equal m. We will use the grid shown in figure 6.4, rather than the one corresponding to the map of Manhattan in figure 6.3 so that you can see a nontrivial example of this problem.

The brute force approach to the Manhattan Tourist problem is to search among all paths in the grid for the longest path, but this is not an option for even a moderately large grid. Inspired by the previous chapter you may be tempted to use a greedy strategy. For example, a sensible greedy strategy would be to choose between two possible directions (south or east) by comparing how many attractions tourists would see if they moved one block south instead of moving one block east. This greedy strategy may provide rewarding sightseeing experience in the beginning but, a few blocks later, may bring you to an area of Manhattan you really do not want to be in. In fact, no known greedy strategy for the Manhattan Tourist problem provides an optimal solution to the problem. Had we followed the (obvious) greedy algorithm, we would have chosen the following path, corresponding to twenty three attractions.[5]

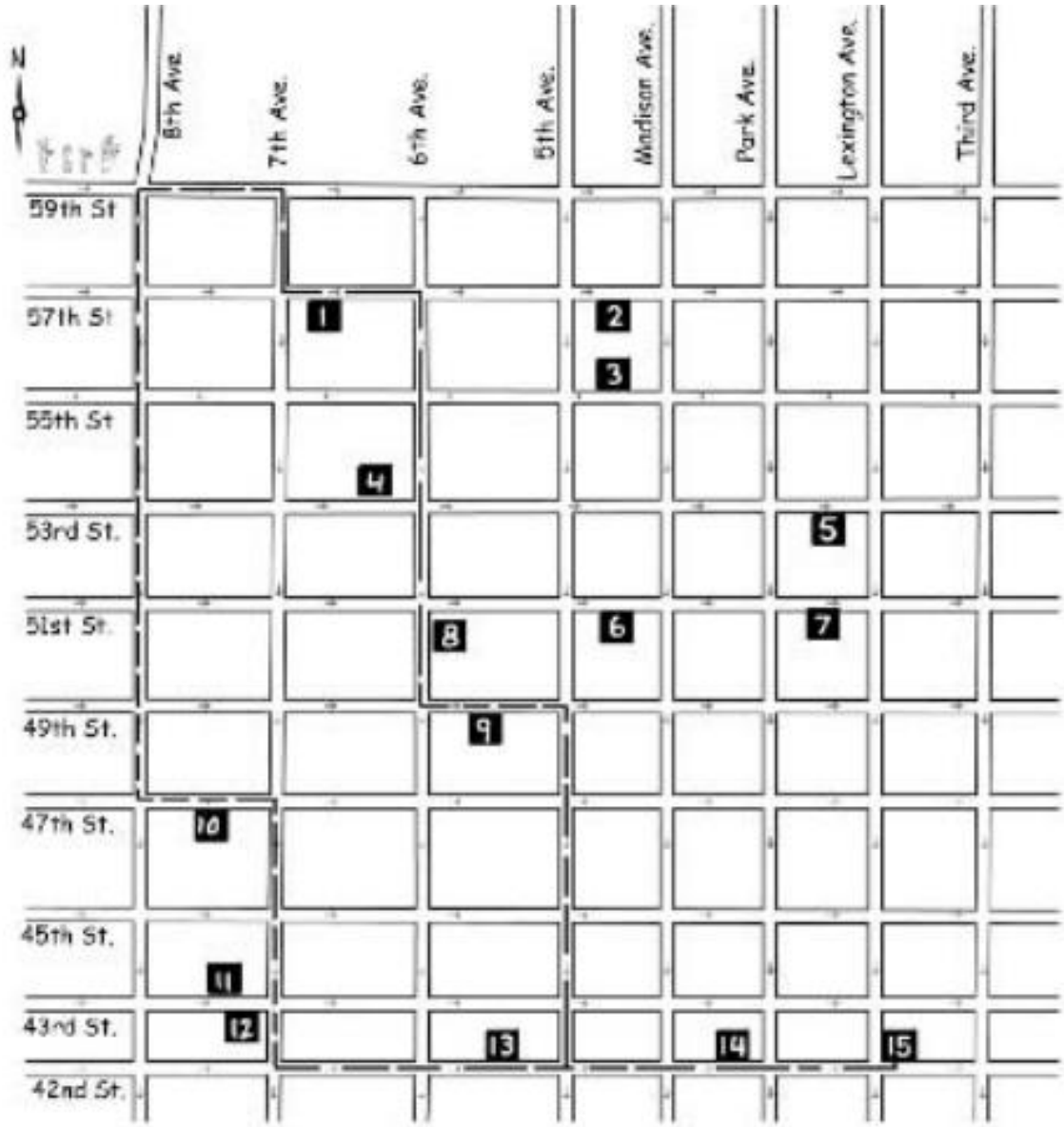


Figure 6.3 A city somewhat like Manhattan, laid out on a grid with one-way st
You may travel only to the east or to the south, and you are currently at the r
westernmost point (source) and need to travel to the southeasternmost point (
Your goal is to visit as many attractions as possible.

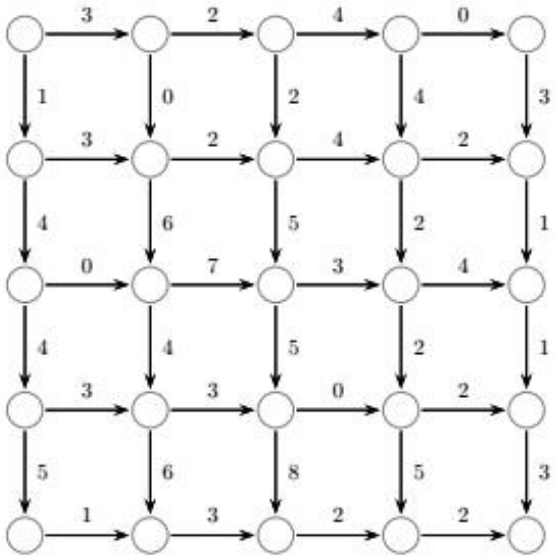
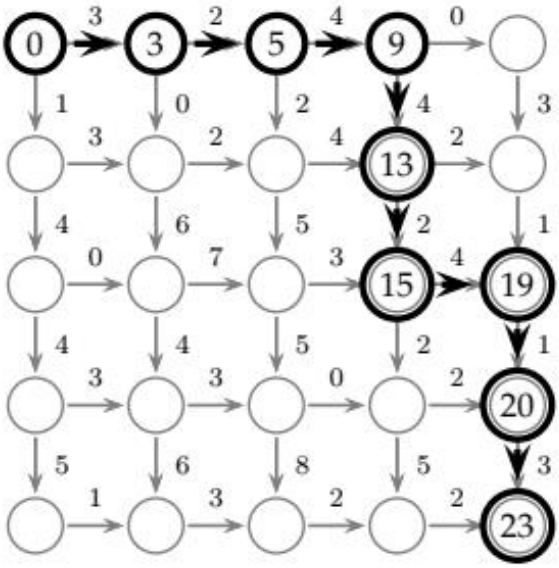


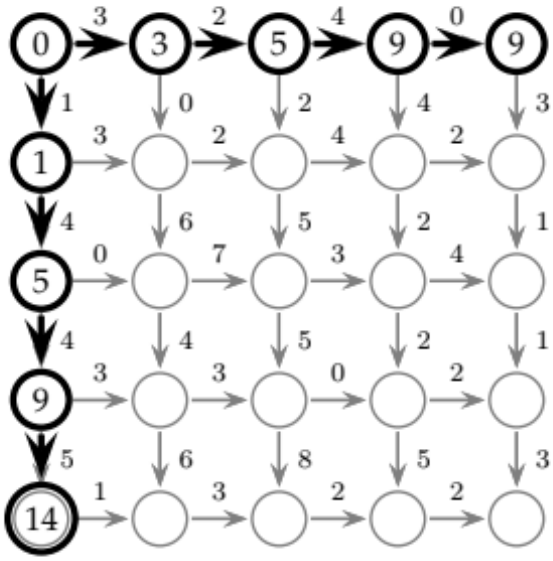
Figure 6.4 Manhattan represented as a graph with weighted edges.



Instead of solving the Manhattan Tourist problem directly, that is, finding the longest path from *source* $(0, 0)$ to *sink* (n, m), we solve a more general problem: find the longest path from *source* to an arbitrary vertex (i, j) with $0 \leq i \leq n, 0 \leq j \leq m$. We will denote the length of such a best path as $s_{i,j}$, noticing that $s_{n,m}$ is the weight of the path that represents the solution to the

Manhattan Tourist problem. If we only care about the longest path between $(0, 0)$ and (n, m)—the Manhattan Tourist problem—then we have to answer one question, namely, what is the best way to get from *source* to *sink*. If we solve the general problem, then we have to answer $n \times m$ questions: what is the best way to get from *source* to anywhere. At first glance it looks like we have just created $n \times m$ different problems (computing (i, j) with $0 \leq i \leq n$ and $0 \leq j \leq m$) instead of a single one (computing $s_{n,m}$), but the fact that solving the more general problem is as easy as solving the Manhattan Tourist problem is the basis of dynamic programming. Note that DPCHANGE also generalized the problems that it solves by finding the optimal number of coins for *all* values less than or equal to M.

Finding $s_{0,j}$ (for $0 \leq j \leq m$) is not hard, since in this case the tourists do not have any flexibility in their choice of path. By moving strictly to the east, the weight of the path $s_{0,j}$ is the sum of weights of the first j city blocks. Similarly, $s_{i,0}$ is also easy to compute for $0 \leq i \leq n$, since the tourists move only to the south.

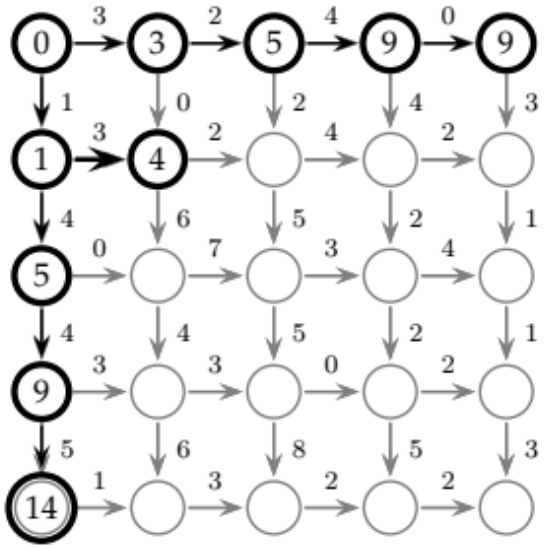


Now that we have figured out how to compute $s_{0,1}$ and $s_{1,0}$, we can compute $s_{1,1}$. The tourists can arrive at $(1, 1)$ in only two ways: either by traveling south from $(0, 1)$ or east from $(1, 0)$. The weight of each of these paths is

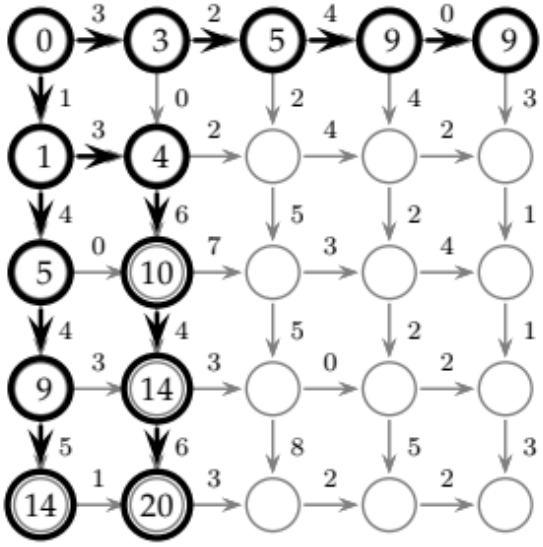
- $s_{0,1}$ + weight of the edge (block) between (0,1) and (1,1);

- $s_{1,0}$ + weight of the edge (block) between (1,0) and (1,1).

Since the goal is to find the longest path to, in this case, $(1, 1)$, we choose the larger of the above two quantities: $3 + 0$ and $1 + 3$. Note that since there are no other ways to get to grid position $(1, 1)$, we have found the longest path from $(0, 0)$ to $(1, 1)$.

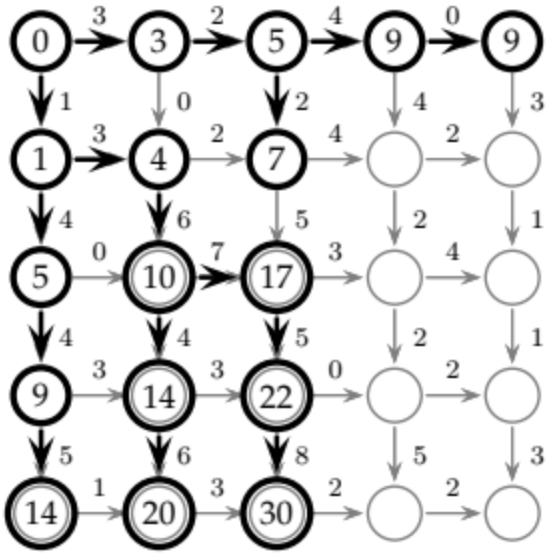


We have just found $s_{1,1}$. Similar logic applies to $s_{2,1}$, and then to $s_{3,1}$, and so on; once we have calculated $s_{i,0}$ for all i, we can calculate $s_{i,1}$ for all i.



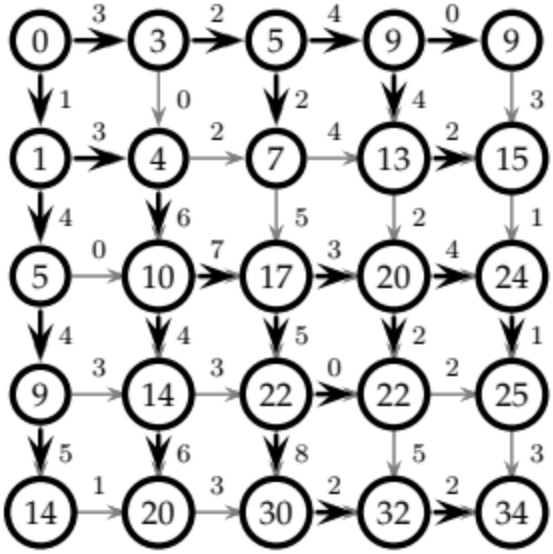
Once we have calculated $s_{i,1}$ for all i, we can use the same idea to calculate $s_{i,2}$ for all i, and so on. For example, we can calculate $s_{1,2}$ as follows.

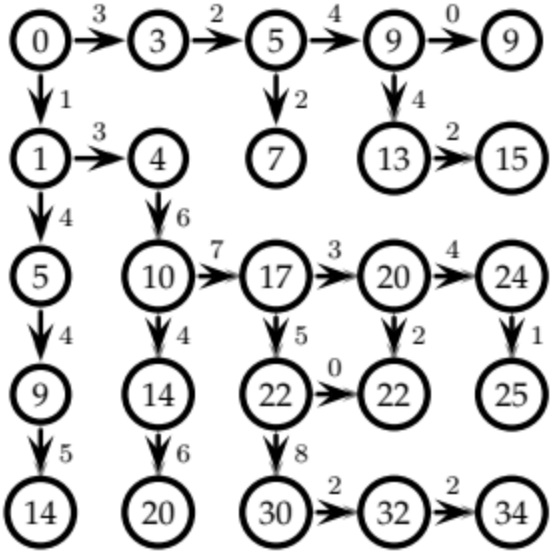
$$s_{1,2} = \max \begin{cases} s_{1,1} + \text{ weight of the edge between (1,1) and (1,2)} \\ s_{0,2} + \text{ weight of the edge between (0,2) and (1,2)} \end{cases}$$



In general, having the entire column $s_{*,j}$ allows us to compute the next whole column $s_{*,j+1}$. The observation that the only way to get to the intersection at (i, j) is either by moving south from intersection $(i-1, j)$ or by moving east from the intersection $(i, j-1)$ leads to the following recurrence:

$$s_{i,j} = \max \begin{cases} s_{i-1,j} + \text{ weight of the edge between } (i-1, j) \text{ and } (i, j) \\ s_{i,j-1} + \text{ weight of the edge between } (i, j-1) \text{ and } (i, j) \end{cases}$$





This recurrence allows us to compute every score $s_{i,j}$ in a single sweep of the grid. The algorithm MANHATTANTOURIST implements this procedure. Here, $\overset{\downarrow}{\mathbf{w}}$ is a two-dimensional array representing the weights of the grid's edges that run north to south, and $\overrightarrow{\mathbf{w}}$ is a two-dimensional array representing the weights of the grid's edges that run west to east. That is, $\overset{\downarrow}{w}_{i,j}$ is the weight of the edge between $(i, j-1)$ and (i, j); and $\overrightarrow{w}_{i,j}$ is the weight of the edge between $(i, j-1)$ and (i, j).

MANHATTANTOURIST($\overset{\downarrow}{\mathbf{w}}, \overrightarrow{\mathbf{w}}, n, m$)
1 $s_{0,0} \leftarrow 0$
2 **for** $i \leftarrow 1$ **to** n
3 $\quad s_{i,0} \leftarrow s_{i-1,0} + \overset{\downarrow}{w}_{i,0}$
4 **for** $j \leftarrow 1$ **to** m
5 $\quad s_{0,j} \leftarrow s_{0,j-1} + \overrightarrow{w}_{0,j}$
6 **for** $i \leftarrow 1$ **to** n
7 $\quad$ **for** $j \leftarrow 1$ **to** m
8 $\quad\quad s_{i,j} \leftarrow \max \begin{cases} s_{i-1,j} + \overset{\downarrow}{w}_{i,j} \\ s_{i,j-1} + \overrightarrow{w}_{i,j} \end{cases}$
9 **return** $s_{n,m}$

Lines 1 through 5 set up the *initial conditions* on the matrix s, and line 8 corresponds to the *recurrence* that allows us to fill in later table entries based on earlier ones. Most of the dynamic programming algorithms we will develop in the context of DNA sequence comparison will look just like MANHATTANTOURIST with only minor changes. We will generally just arrive at a recurrence like line 8 and call it an algorithm, with the understanding that the actual implementation will be similar to MANHATTANTOURIST.[6]

Many problems in bioinformatics can be solved efficiently by the application of the dynamic programming technique, once they are cast as traveling in a Manhattan-like grid. For example, development of new sequence comparison algorithms often amounts to building an appropriate "Manhattan" that adequately models the specifics of a particular biological problem, and by defining the block weights that reflect the costs of mutations from one DNA sequence to another.

6. MANHATTANTOURIST computes the length of the longest path in the grid, but does not give the path itself. In section 6.5 we will describe a minor modification to the algorithm that returns not only the optimal length, but also the optimal path.

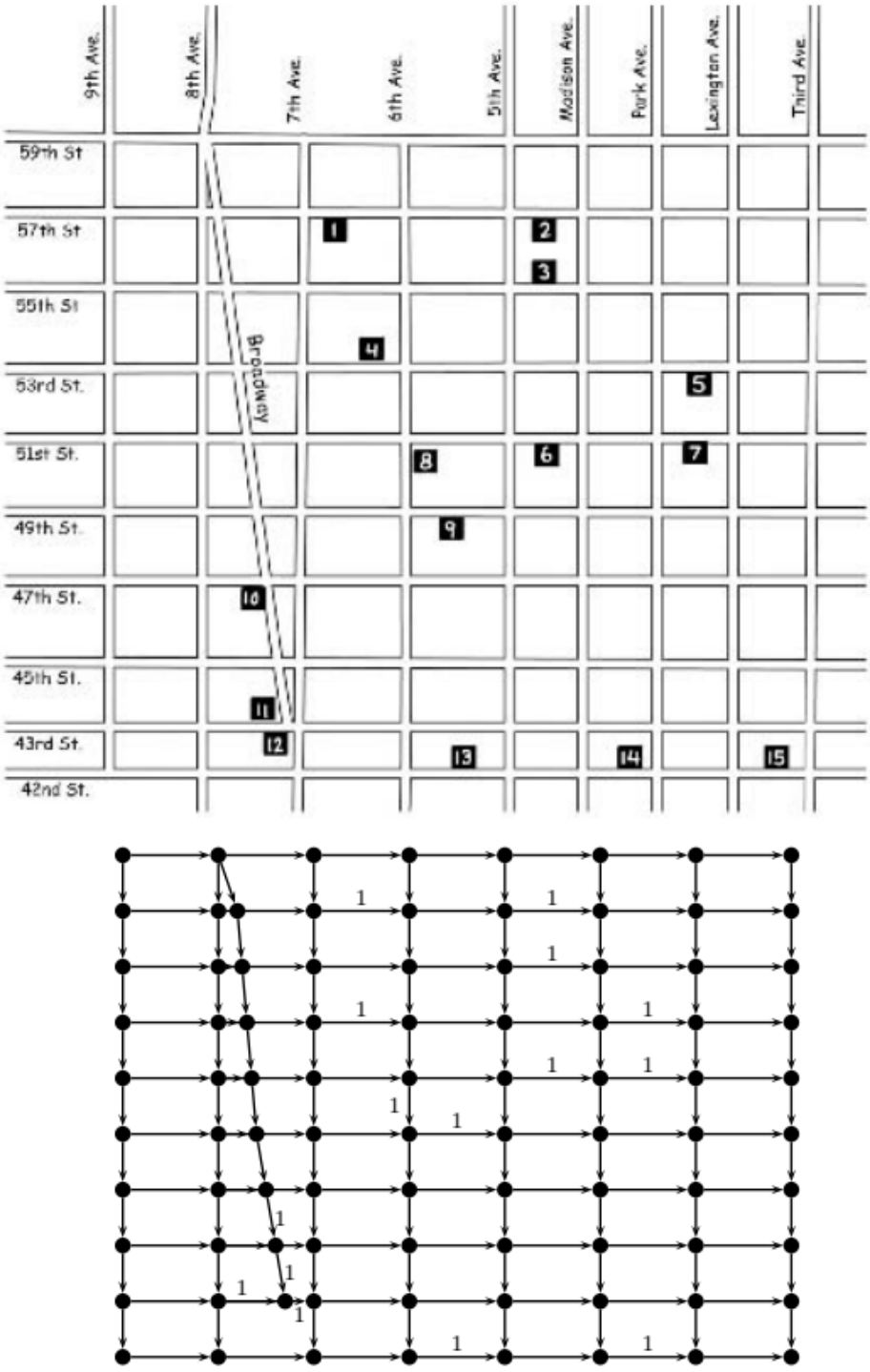


Figure 6.5 A city somewhat more like Manhattan than figure 6.4 with the complicating issue of a street that runs diagonally across the grid. Broadway cuts across several blocks. In the case of the Manhattan Tourist problem, it changes the optimal path (the optimal path in this new city has six attractions instead of five).

Unfortunately, Manhattan is not a perfectly regular grid. Broadway cuts across the borough (figure 6.5). We would like to solve a generalization of the Manhattan Tourist problem for the case in which the street map is not a regular rectangular grid. In this case, one can model any city map as a graph with vertices corresponding to the intersections of streets, and edges corresponding to the intervals of streets between the intersections. For the sake of simplicity we assume that the city blocks correspond to *directed* edges, so that the tourist can move only in the direction of the edge and that the resulting graph has no *directed cycles*.[7] Such graphs are called *directed acyclic graphs*, or *DAGs*. We assume that every edge has an associated weight (e.g., the number of attractions) and represent a graph G as a pair of two sets, V for vertices and E for edges: $G = (V, E)$. We number vertices from 1 to $|V|$ with a single integer, rather than a row-column pair as in the Manhattan problem. This does not change the generic dynamic programming algorithm other than in notation, but it allows us to represent imperfect grids. An edge from E can be specified in terms of its origin vertex u and its destination vertex v as (u, v). The following problem is simply a generalization of the Manhattan Tourist problem that is able to deal with arbitrary DAGs rather than with perfect grids.

Longest Path in a DAG Problem:
Find a longest path between two vertices in a weighted DAG.

Input: A weighted DAG G with *source* and *sink* vertices.

Output: A longest path in G from *source* to *sink*.

Not surprisingly, the Longest Path in a DAG problem can also be solved by dynamic programming. At every vertex, there may be multiple edges that "flow in" and multiple edges that "flow out." In the city analogy, any intersection may have multiple one-way streets leading in, and some other number of one-way streets exiting. We will call the number of edges entering a vertex (i.e., the number of inbound streets) the *indegree* of the vertex (i.e., intersection), and the number of edges leaving a vertex (i.e., the number of outbound streets) the *outdegree* of the vertex.

In the nicely regular case of the Manhattan problem, most vertices had

7. A directed cycle is a path from a vertex back to itself that respects the directions of edges. If the resulting graph contained a cycle, a tourist could start walking along this cycle revisiting the same attractions many times. In this case there is no "best" solution since a tourist may increase the number of visited attractions indefinitely.

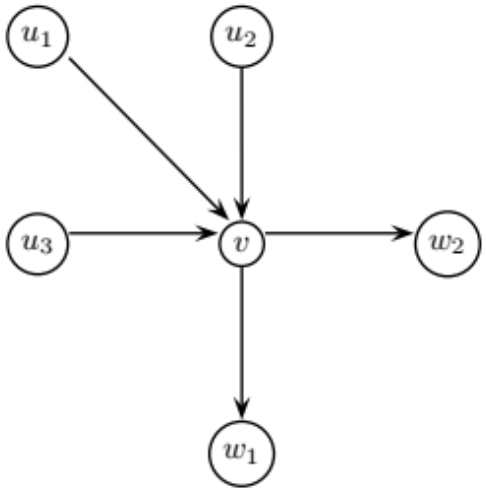


Figure 6.6 A graph with six vertices. The vertex v has indegree 3 and outdegree 2. The vertices u_1, u_2 and u_3 are all predecessors of v, and w_1 and w_2 are successors of v.

indegree 2 and outdegree 2, except for the vertices along the boundaries of the grid. In the more general DAG problem, a vertex can have an arbitrary indegree and outdegree. We will call u a *predecessor* to vertex v if $(u, v) \in E$— in other words, a predecessor of a vertex is any vertex that can be reached by traveling backwards along an inbound edge. Clearly, if v has indegree k, it has k predecessors.

Suppose a vertex v has indegree 3, and the set of predecessors of v is $\{u_1, u_2, u_3\}$ (figure 6.6). The longest path to v can be computed as follows:

$$s_v = \max \begin{cases} s_{u_1} + \text{ weight of edge from } u_1 \text{ to } v \\ s_{u_2} + \text{ weight of edge from } u_2 \text{ to } v \\ s_{u_3} + \text{ weight of edge from } u_3 \text{ to } v \end{cases}$$

In general, one can imagine a rather hectic city plan, but the recurrence relation remains simple, with the score s_v of the vertex v defined as follows.

$$s_v = \max_{u \in Predecessors(v)} (s_u + \text{ weight of edge from } u \text{ to } v)$$

Here, $Predecessors(v)$ is the set of all vertices u such that u is a predecessor of v. Since every edge participates in only a single recurrence, the running

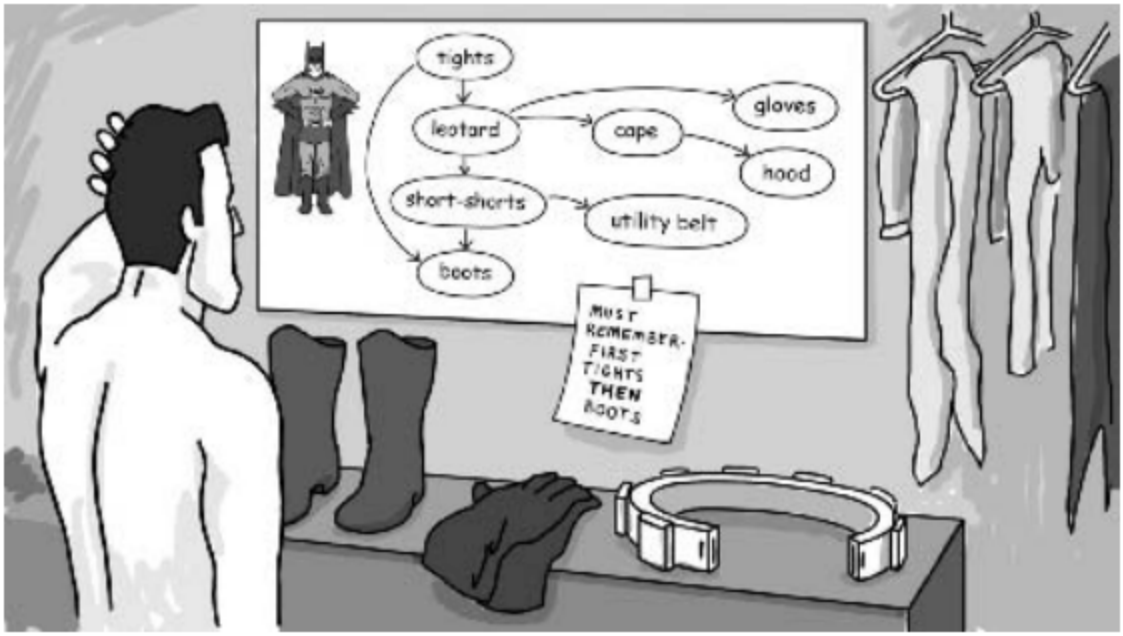


Figure 6.7 The "Dressing in the Morning problem" represented by a DAG. Some of us have more trouble than others.

time of the algorithm is defined by the number of edges in the graph.[8] The one hitch to this plan for solving the Longest Path problem in a DAG is that one must decide on the order in which to visit the vertices while computing s. This ordering is important, since by the time vertex v is analyzed, the values s_u for *all* its predecessors must have been computed. Three popular strategies for exploring the perfect grid are displayed in figure 6.9, column by column, row by row, and diagonal by diagonal. These exploration strategies correspond to different *topological orderings* of the DAG corresponding to the perfect grid. An ordering of vertices $v_1, \ldots, v_n$ of a DAG is called *topological* if every edge (v_i, v_j) of the DAG connects a vertex with a smaller index to a vertex with a larger index, that is, $i < j$. Figure 6.7 represents a DAG that corresponds to a problem that we each face every morning. Every DAG has a topological ordering (fig. 6.8); a problem at the end of this chapter asks you to prove this fact.

8. A graph with vertex set V can have at most $|V|^2$ edges, but graphs arising in sequence comparison are usually sparse, with many fewer edges.

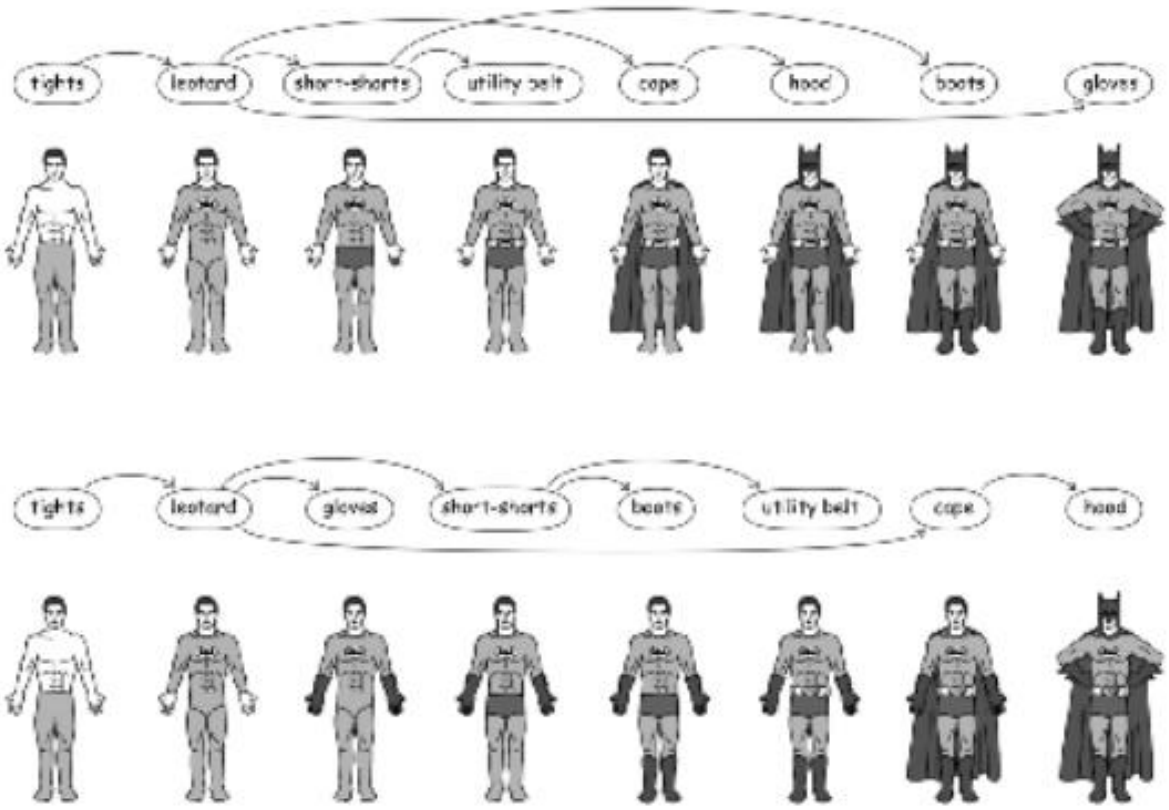
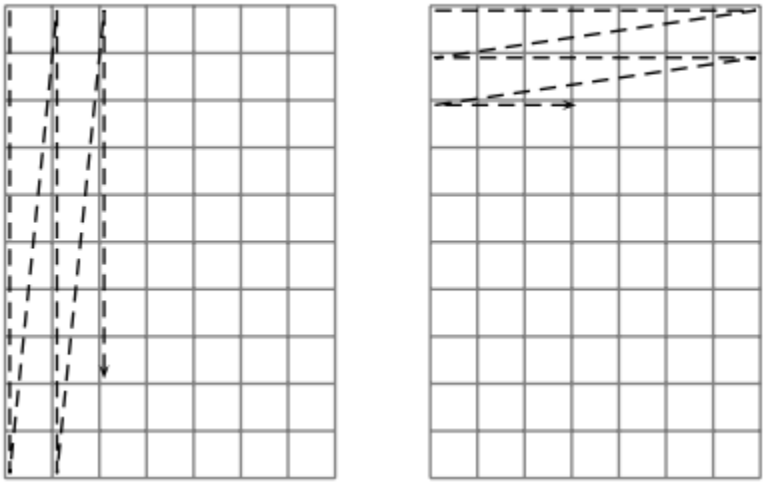


Figure 6.8 Two different ways of getting dressed in the morning corresponding to two different topological orderings of the graph in figure 6.7.



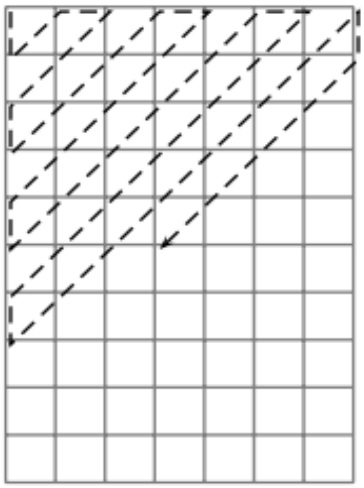


Figure 6.9 Three different strategies for filling in a dynamic programming array. The first fills in the array column by column: earlier columns are filled in before later ones. The second fills in the array row by row. The third method fills array entries along the diagonals and is useful in parallel computation.

Edit Distance and Alignments

So far, we have been vague about what we mean by "sequence similarity" or "distance" between DNA sequences. Hamming distance (introduced in chapter 4), while important in computer science, is not typically used to compare DNA or protein sequences. The Hamming distance calculation rigidly assumes that the ith symbol of one sequence is already *aligned* against the ith symbol of the other. However, it is often the case that the ith symbol in one sequence corresponds to a symbol at a different—and unknown—position in the other. For example, mutation in DNA is an evolutionary process: DNA replication errors cause substitutions, insertions, and deletions of nucleotides, leading to "edited" DNA texts. Since DNA sequences are subject to insertions and deletions, biologists rarely have the luxury of knowing in advance whether the ith symbol in one DNA sequence corresponds to the ith symbol in the other.

As figure 6.10 (a) shows, while strings ATATATAT and TATATATA are very different from the perspective of Hamming distance, they become very similar if one simply moves the second string over one place to align the $(i+1)$-st letter in ATATATAT against the ith letter in TATATATA for $1 \leq i \leq 7$. Strings ATATATAT and TATAAT present another example with more subtle similarities.

Figure 6.10 (b) reveals these similarities by aligning position 2 in ATATATAT against position 1 in TATAAT. Other pairs of aligned positions are 3 against 2, 4 against 3, 5 against 4, 7 against 5, and 8 against 6 (positions 1 and 6 in ATATATAT remain unaligned).

In 1966, Vladimir Levenshtein introduced the notion of the *edit distance* between two strings as the minimum number of editing operations needed to transform one string into another, where the edit operations are insertion of a symbol, deletion of a symbol, and substitution of one symbol for another. For example, TGCATAT can be transformed into ATCCGAT with five editing operations, shown in figure 6.11. This implies that the edit distance between TGCATAT and ATCCGAT is at most 5. Actually, the edit distance between them is 4 because you can transform one to the other with one move fewer, as in figure 6.12.

Unlike Hamming distance, edit distance allows one to compare strings of different lengths. Oddly, Levenshtein introduced the definition of edit distance but never described an algorithm for actually finding the edit distance between two strings. This algorithm has been discovered and rediscovered many times in applications ranging from automated speech recognition to, obviously, molecular biology. Although the details of the algorithms are

```
A T A T A T A T -
  : : : : : : :
- T A T A T A T A
```

(a) Alignment of ATATATAT against TATATATA.

```
A T A T A T A T
  : : : :   : :
- T A T A - A T
```

(b) Alignment of ATATATAT against TATAAT.

Figure 6.10 Alignment of ATATATAT against TATATATA and of ATATATAT against TATAAT.

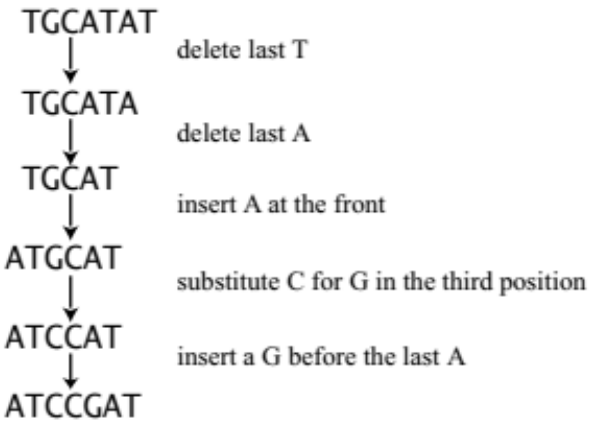


Figure 6.11 Five edit operations can take TGCATAT into ATCCGAT.

slightly different across the various applications, they are all based on dynamic programming.

The *alignment* of the strings **v** (of n characters) and **w** (of m characters, with m not necessarily the same as n) is a two-row matrix such that the first row contains the characters of **v** in order while the second row contains the characters of **w** in order, where spaces may be interspersed throughout the strings in different places. As a result, the characters in each string appear in order, though not necessarily adjacently. We also assume that no column

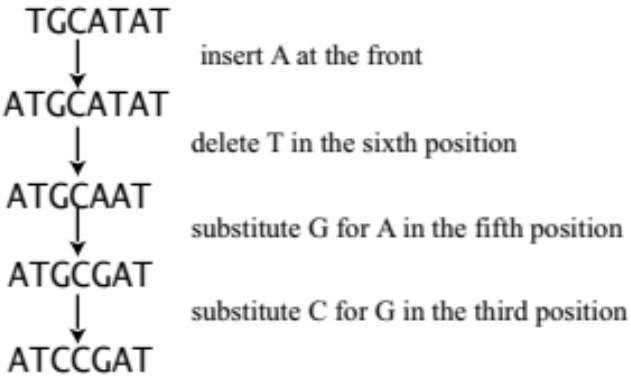


Figure 6.12 Four edit operations can also take TGCATAT into ATCCGAT.

of the alignment matrix contains spaces in both rows, so that the alignment may have at most $n + m$ columns.

A	T	-	G	T	T	A	T	-
A	T	C	G	T	-	A	-	C

Columns that contain the same letter in both rows are called *matches*, while columns containing different letters are called *mismatches*. The columns of the alignment containing one space are called *indels*, with the columns containing a space in the top row called *insertions* and the columns with a space in the bottom row *deletions*. The alignment shown in figure 6.13 (top) has five matches, zero mismatches, and four indels. The number of matches plus the number of mismatches plus the number of indels is equal to the length of the alignment matrix and must be smaller than $n + m$.

Each of the two rows in the alignment matrix is represented as a string interspersed by space symbols "−"; for example AT−GTTAT− is a representation of the row corresponding to $\mathbf{v}$ = ATGTTAT, while ATCGT−A−C is a representation of the row corresponding to $\mathbf{w}$ = ATCGTAC. Another way to represent the row AT−GTTAT− is 1 2 2 3 4 5 6 7 7, which shows the number of symbols of $\mathbf{v}$ present up to a given position. Similarly, ATCGT−A−C is represented as 1 2 3 4 5 5 6 6 7. When both rows of an alignment are represented in this way (fig. 6.13, top), the resulting matrix is

$$\begin{pmatrix} 0 \\ 0 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \end{pmatrix} \begin{pmatrix} 2 \\ 2 \end{pmatrix} \begin{pmatrix} 2 \\ 3 \end{pmatrix} \begin{pmatrix} 3 \\ 4 \end{pmatrix} \begin{pmatrix} 4 \\ 5 \end{pmatrix} \begin{pmatrix} 5 \\ 5 \end{pmatrix} \begin{pmatrix} 6 \\ 6 \end{pmatrix} \begin{pmatrix} 7 \\ 6 \end{pmatrix} \begin{pmatrix} 7 \\ 7 \end{pmatrix}$$

Each column in this matrix is a coordinate in a two-dimensional $n \times m$ grid;

the entire alignment is simply a path

$$(0,0) \rightarrow (1,1) \rightarrow (2,2) \rightarrow (2,3) \rightarrow (3,4) \rightarrow (4,5) \rightarrow (5,5) \rightarrow (6,6) \rightarrow (7,6) \rightarrow (7,7)$$

from $(0,0)$ to (n,m) in that grid (again, see figure 6.13). This grid is similar to the Manhattan grid that we introduced earlier, where each entry in the grid looks like a city block. The main difference is that here we can move along the diagonal. We can construct a graph, this time called the *edit graph*, by introducing a vertex for every intersection of streets in the grid, shown in figure 6.13. The edit graph will aid us in calculating the edit distance.

Every alignment corresponds to a path in the edit graph, and every path in the edit graph corresponds to an alignment where every edge in the path corresponds to one column in the alignment (fig. 6.13). Diagonal edges in the path that end at vertex (i, j) in the graph correspond to the column $\begin{pmatrix} v_i \\ w_j \end{pmatrix}$, horizontal edges correspond to $\begin{pmatrix} - \\ w_j \end{pmatrix}$, and vertical edges correspond to $\begin{pmatrix} v_i \\ - \end{pmatrix}$. The alignment above can be drawn as follows.

$$\begin{array}{cccccccccccc} & \mathsf{A} & & \mathsf{T} & & - & & \mathsf{G} & & \mathsf{T} & & \mathsf{T} & & \mathsf{A} & & \mathsf{T} & & - \\ \begin{pmatrix} 0 \\ 0 \end{pmatrix} & & \begin{pmatrix} 1 \\ 1 \end{pmatrix} & & \begin{pmatrix} 2 \\ 2 \end{pmatrix} & & \begin{pmatrix} 2 \\ 3 \end{pmatrix} & & \begin{pmatrix} 3 \\ 4 \end{pmatrix} & & \begin{pmatrix} 4 \\ 5 \end{pmatrix} & & \begin{pmatrix} 5 \\ 5 \end{pmatrix} & & \begin{pmatrix} 6 \\ 6 \end{pmatrix} & & \begin{pmatrix} 7 \\ 6 \end{pmatrix} & & \begin{pmatrix} 7 \\ 7 \end{pmatrix} \\ & \mathsf{A} & & \mathsf{T} & & \mathsf{G} & & \mathsf{C} & & \mathsf{T} & & - & & \mathsf{A} & & - & & \mathsf{C} \end{array}$$

Analyzing the merit of an alignment is equivalent to analyzing the merit of the corresponding path in the edit graph. Given any two strings, there are a large number of different alignment matrices and corresponding paths in the edit graph. Some of these have a surplus of mismatches and indels and a small number of matches, while others have many matches and few indels and mismatches. To determine the relative merits of one alignment over another, we rely on the notion of a scoring function, which takes as input an alignment matrix (or, equivalently, a path in the edit graph) and produces a score that determines the "goodness" of the alignment. There are a variety of scoring functions that we could use, but we want one that gives higher scores to alignments with more matches. The simplest functions score a column as a positive number if both letters are the same, and as a negative number if the two letters are different. The score for the whole alignment is the sum of the individual column scores. This scoring scheme amounts to

$$\begin{array}{cccccccccccc} & & 0 & 1 & 2 & 2 & 3 & 4 & 5 & 6 & 7 & 7 \\ \mathbf{v} & = & & \mathsf{A} & \mathsf{T} & - & \mathsf{G} & \mathsf{T} & \mathsf{T} & \mathsf{A} & \mathsf{T} & - \\ & & & | & | & & | & | & & | & & \\ \mathbf{w} & = & & \mathsf{A} & \mathsf{T} & \mathsf{C} & \mathsf{G} & \mathsf{T} & - & \mathsf{A} & - & \mathsf{C} \\ & & 0 & 1 & 2 & 3 & 4 & 5 & 5 & 6 & 6 & 7 \end{array}$$

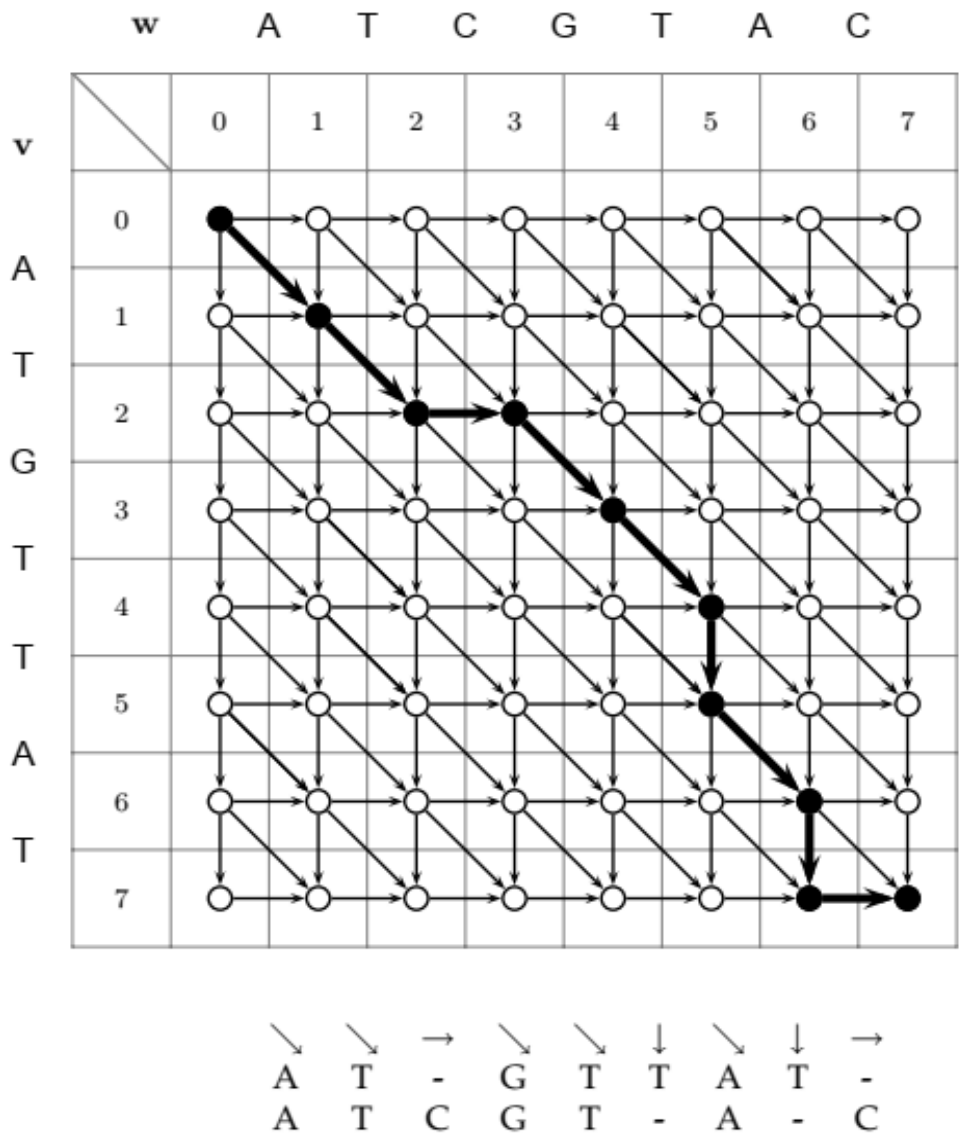


Figure 6.13 An alignment grid for $\mathbf{v} = \text{ATGTTAT}$ and $\mathbf{w} = \text{ATCGTAC}$. Every alignment corresponds to a path in the alignment grid from $(0, 0)$ to (n, m), and every path from $(0, 0)$ to (n, m) in the alignment grid corresponds to an alignment.

assigning weights to the edges in the edit graph.

By choosing different scoring functions, we can solve different string comparison problems. If we choose the very simple scoring function of "+1 for a match, 0 otherwise," then the problem becomes that of finding the longest common subsequence between two strings, which is discussed below. Before describing how to calculate Levenshtein's edit distance, we develop the Longest Common Subsequence problem as a warm-up.