Chapter 6
Dynamic programming

In the preceding chapters we have seen some elegant design principles—such as divide-and-conquer, graph exploration, and greedy choice—that yield definitive algorithms for a variety of important computational tasks. The drawback of these tools is that they can only be used on very specific types of problems. We now turn to the two *sledgehammers* of the algorithms craft, *dynamic programming* and *linear programming*, techniques of very broad applicability that can be invoked when more specialized methods fail. Predictably, this generality often comes with a cost in efficiency.

6.1 Shortest paths in dags, revisited

At the conclusion of our study of shortest paths (Chapter 4), we observed that the problem is especially easy in directed acyclic graphs (dags). Let's recapitulate this case, because it lies at the heart of dynamic programming.

The special distinguishing feature of a dag is that its nodes can be *linearized*; that is, they can be arranged on a line so that all edges go from left to right (Figure 6.1). To see why this helps with shortest paths, suppose we want to figure out distances from node S to the other nodes. For concreteness, let's focus on node D. The only way to get to it is through its predecessors, B or C; so to find the shortest path to D, we need only compare these two routes:

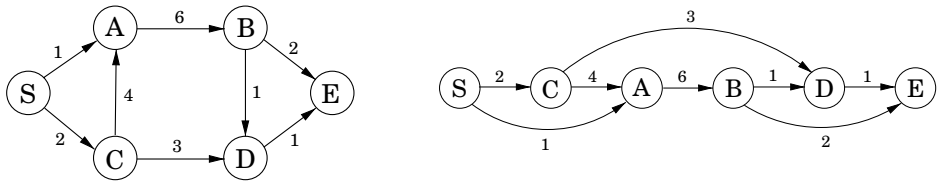
$$\texttt{dist}(D) = \min\{\texttt{dist}(B) + 1, \texttt{dist}(C) + 3\}.$$

A similar relation can be written for every node. If we compute these `dist` values in the left-to-right order of Figure 6.1, we can always be sure that by the time we get to a node v, we already have all the information we need to compute $\texttt{dist}(v)$. We are therefore able to compute all distances in a single pass:

```
initialize all dist(·) values to ∞
dist(s) = 0
for each v ∈ V\{s}, in linearized order:
    dist(v) = min_{(u,v)∈E}{dist(u) + l(u, v)}
```

Notice that this algorithm is solving a collection of *subproblems*, $\{\texttt{dist}(u) : u \in V\}$. We start with the smallest of them, $\texttt{dist}(s)$, since we immediately know its answer

Figure 6.1 A dag and its linearization (topological ordering).



to be 0. We then proceed with progressively "larger" subproblems—distances to vertices that are further and further along in the linearization—where we are thinking of a subproblem as large if we need to have solved a lot of other subproblems before we can get to it.

This is a very general technique. At each node, we compute some function of the values of the node's predecessors. It so happens that our particular function is a minimum of sums, but we could just as well make it a *maximum*, in which case we would get *longest* paths in the dag. Or we could use a product instead of a sum inside the brackets, in which case we would end up computing the path with the smallest product of edge lengths.

Dynamic programming is a very powerful algorithmic paradigm in which a problem is solved by identifying a collection of subproblems and tackling them one by one, smallest first, using the answers to small problems to help figure out larger ones, until the whole lot of them is solved. In dynamic programming we are not given a dag; the dag is *implicit*. Its nodes are the subproblems we define, and its edges are the dependencies between the subproblems: if to solve subproblem B we need the answer to subproblem A, then there is a (conceptual) edge from A to B. In this case, A is thought of as a smaller subproblem than B—and it will always be smaller, in an obvious sense.

But it's time we saw an example.

6.2 Longest increasing subsequences

In the *longest increasing subsequence* problem, the input is a sequence of numbers $a_1, \ldots, a_n$. A *subsequence* is any subset of these numbers taken in order, of the form $a_{i_1}, a_{i_2}, \ldots, a_{i_k}$ where $1 \leq i_1 < i_2 < \cdots < i_k \leq n$, and an *increasing* subsequence is one in which the numbers are getting strictly larger. The task is to find the increasing subsequence of greatest length. For instance, the longest increasing subsequence of 5, 2, 8, 6, 3, 6, 9, 7 is 2, 3, 6, 9:

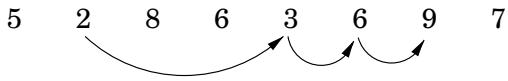
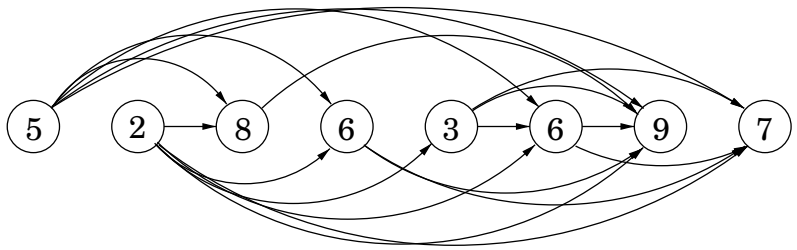


Figure 6.2 The dag of increasing subsequences.



In this example, the arrows denote transitions between consecutive elements of the optimal solution. More generally, to better understand the solution space, let's create a graph of *all* permissible transitions: establish a node i for each element a_i, and add directed edges (i, j) whenever it is possible for a_i and a_j to be consecutive elements in an increasing subsequence, that is, whenever $i < j$ and $a_i < a_j$ (Figure 6.2).

Notice that (1) this graph $G = (V, E)$ is a dag, since all edges (i, j) have $i < j$, and (2) there is a one-to-one correspondence between increasing subsequences and paths in this dag. Therefore, our goal is simply to find the longest path in the dag!

Here is the algorithm:

```
for j = 1, 2, ..., n:
    L(j) = 1 + max{L(i) : (i, j) ∈ E}
return max_j L(j)
```

$L(j)$ is the length of the longest path—the longest increasing subsequence—ending at j (plus 1, since strictly speaking we need to count nodes on the path, not edges). By reasoning in the same way as we did for shortest paths, we see that any path to node j must pass through one of its predecessors, and therefore $L(j)$ is 1 plus the maximum $L(\cdot)$ value of these predecessors. If there are no edges into j, we take the maximum over the empty set, zero. And the final answer is the *largest* $L(j)$, since any ending position is allowed.

This is dynamic programming. In order to solve our original problem, we have defined a collection of subproblems $\{L(j) : 1 \leq j \leq n\}$ with the following key property that allows them to be solved in a single pass:

> (*) *There is an ordering on the subproblems, and a relation that shows how to solve a subproblem given the answers to "smaller" subproblems, that is, subproblems that appear earlier in the ordering.*

In our case, each subproblem is solved using the relation

$$L(j) = 1 + \max\{L(i) : (i, j) \in E\},$$

an expression which involves only smaller subproblems. How long does this step take? It requires the predecessors of j to be known; for this the adjacency list of the reverse graph G^R, constructible in linear time (recall Exercise 3.5), is handy. The computation of $L(j)$ then takes time proportional to the indegree of j, giving an overall running time linear in $|E|$. This is at most $O(n^2)$, the maximum being when the input array is sorted in increasing order. Thus the dynamic programming solution is both simple and efficient.

There is one last issue to be cleared up: the L-values only tell us the *length* of the optimal subsequence, so how do we recover the subsequence itself? This is easily managed with the same bookkeeping device we used for shortest paths in Chapter 4. While computing $L(j)$, we should also note down $\texttt{prev}(j)$, the next-to-last node on the longest path to j. The optimal subsequence can then be reconstructed by following these backpointers.

6.3 Edit distance

When a spell checker encounters a possible misspelling, it looks in its dictionary for other words that are close by. What is the appropriate notion of closeness in this case?

A natural measure of the distance between two strings is the extent to which they can be *aligned*, or matched up. Technically, an alignment is simply a way of writing the strings one above the other. For instance, here are two possible alignments of SNOWY and SUNNY:

```
S - N O W Y        - S N O W - Y
S U N N - Y        S U N - - N Y
   Cost: 3            Cost: 5
```

The "−" indicates a "gap"; any number of these can be placed in either string. The *cost* of an alignment is the number of columns in which the letters differ. And the *edit distance* between two strings is the cost of their best possible alignment. Do you see that there is no better alignment of SNOWY and SUNNY than the one shown here with a cost of 3?

Edit distance is so named because it can also be thought of as the minimum number of *edits*—insertions, deletions, and substitutions of characters—needed to transform the first string into the second. For instance, the alignment shown on the left corresponds to three edits: insert U, substitute O → N, and delete W.

In general, there are so many possible alignments between two strings that it would be terribly inefficient to search through all of them for the best one. Instead we turn to dynamic programming.

A dynamic programming solution

When solving a problem by dynamic programming, the most crucial question is, *What are the subproblems*? As long as they are chosen so as to have the property

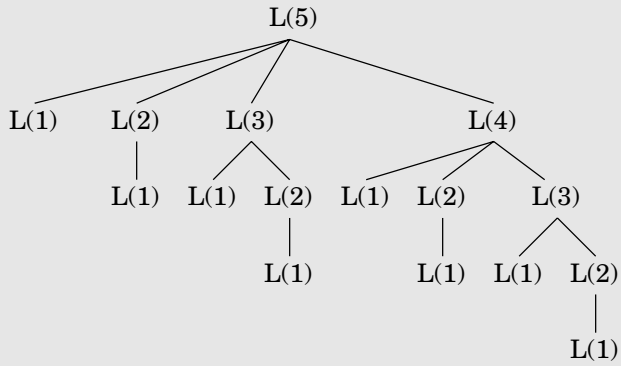
Recursion? No, thanks.

Returning to our discussion of longest increasing subsequences: the formula for $L(j)$ also suggests an alternative, recursive algorithm. Wouldn't that be even simpler?

Actually, recursion is a very bad idea: the resulting procedure would require exponential time! To see why, suppose that the dag contains edges (i, j) for *all* $i < j$—that is, the given sequence of numbers $a_1, a_2, \ldots, a_n$ is sorted. In that case, the formula for subproblem $L(j)$ becomes

$$L(j) = 1 + \max\{L(1), L(2), \ldots, L(j-1)\}.$$

The following figure unravels the recursion for $L(5)$. Notice that the same subproblems get solved over and over again!



For $L(n)$ this tree has exponentially many nodes (can you bound it?), and so a recursive solution is disastrous.

Then why did recursion work so well with divide-and-conquer? The key point is that in divide-and-conquer, a problem is expressed in terms of subproblems that are *substantially smaller*, say half the size. For instance, `mergesort` sorts an array of size n by recursively sorting two subarrays of size $n/2$. Because of this sharp drop in problem size, the full recursion tree has only logarithmic depth and a polynomial number of nodes.

In contrast, in a typical dynamic programming formulation, a problem is reduced to subproblems that are only slightly smaller—for instance, $L(j)$ relies on $L(j-1)$. Thus the full recursion tree generally has polynomial depth and an exponential number of nodes. However, it turns out that most of these nodes are repeats, that there are not too many *distinct* subproblems among them. Efficiency is therefore obtained by explicitly enumerating the distinct subproblems and solving them in the right order.

Programming?

The origin of the term *dynamic programming* has very little to do with writing code. It was first coined by Richard Bellman in the 1950s, a time when computer programming was an esoteric activity practiced by so few people as to not even merit a name. Back then programming meant "planning," and "dynamic programming" was conceived to optimally plan multistage processes. The dag of Figure 6.2 can be thought of as describing the possible ways in which such a process can evolve: each node denotes a state, the leftmost node is the starting point, and the edges leaving a state represent possible actions, leading to different states in the next unit of time.

The etymology of *linear programming*, the subject of Chapter 7, is similar.

(*) from page 158. it is an easy matter to write down the algorithm: iteratively solve one subproblem after the other, in order of increasing size.

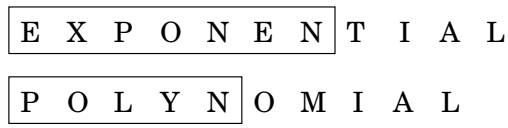
Our goal is to find the edit distance between two strings $x[1 \cdots m]$ and $y[1 \cdots n]$. What is a good subproblem? Well, it should go part of the way toward solving the whole problem; so how about looking at the edit distance between some *prefix* of the first string, $x[1 \cdots i]$, and some *prefix* of the second, $y[1 \cdots j]$? Call this subproblem $E(i, j)$ (see Figure 6.3). Our final objective, then, is to compute $E(m, n)$.

For this to work, we need to somehow express $E(i, j)$ in terms of smaller subproblems. Let's see—what do we know about the best alignment between $x[1 \cdots i]$ and $y[1 \cdots j]$? Well, its rightmost column can only be one of three things:

$$\begin{matrix} x[i] \\ - \end{matrix} \quad \text{or} \quad \begin{matrix} - \\ y[j] \end{matrix} \quad \text{or} \quad \begin{matrix} x[i] \\ y[j] \end{matrix}$$

The first case incurs a cost of 1 for this particular column, and it remains to align $x[1 \cdots i-1]$ with $y[1 \cdots j]$. But this is exactly the subproblem $E(i-1, j)$! We seem to be getting somewhere. In the second case, also with cost 1, we still need to align $x[1 \cdots i]$ with $y[1 \cdots j-1]$. This is again another subproblem, $E(i, j-1)$. And in the final case, which either costs 1 (if $x[i] \neq y[j]$) or 0 (if $x[i] = y[j]$), what's left is the subproblem $E(i-1, j-1)$. In short, we have expressed $E(i, j)$ in terms of

Figure 6.3 The subproblem $E(7, 5)$.



three *smaller* subproblems $E(i-1, j)$, $E(i, j-1)$, $E(i-1, j-1)$. We have no idea which of them is the right one, so we need to try them all and pick the best:

$$E(i, j) = \min\{1 + E(i-1, j),\ 1 + E(i, j-1),\ \text{diff}(i, j) + E(i-1, j-1)\}$$

where for convenience diff(i, j) is defined to be 0 if $x[i] = y[j]$ and 1 otherwise.

For instance, in computing the edit distance between EXPONENTIAL and POLYNOMIAL, subproblem $E(4, 3)$ corresponds to the prefixes EXPO and POL. The rightmost column of their best alignment must be one of the following:

$$\begin{matrix}\texttt{O}\\ -\end{matrix} \quad \texttt{or} \quad \begin{matrix}-\\ \texttt{L}\end{matrix} \quad \texttt{or} \quad \begin{matrix}\texttt{O}\\ \texttt{L}\end{matrix}$$

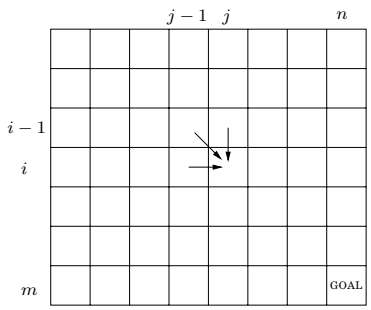
Thus, $E(4, 3) = \min\{1 + E(3, 3),\ 1 + E(4, 2),\ 1 + E(3, 2)\}$.

The answers to all the subproblems $E(i, j)$ form a two-dimensional table, as in Figure 6.4. In what order should these subproblems be solved? Any order is fine, as long as $E(i-1, j)$, $E(i, j-1)$, and $E(i-1, j-1)$ are handled *before* $E(i, j)$. For instance, we could fill in the table one row at a time, from top row to bottom row, and moving left to right across each row. Or alternatively, we could fill it in column by column. Both methods would ensure that by the time we get around to computing a particular table entry, all the other entries we need are already filled in.

With both the subproblems and the ordering specified, we are almost done. There just remain the "base cases" of the dynamic programming, the very smallest subproblems. In the present situation, these are $E(0, \cdot)$ and $E(\cdot, 0)$, both of which are easily solved. $E(0, j)$ is the edit distance between the 0-length prefix of x,

Figure 6.4 (a) The table of subproblems. Entries $E(i-1, j-1)$, $E(i-1, j)$, and $E(i, j-1)$ are needed to fill in $E(i, j)$. (b) The final table of values found by dynamic programming.

(a)



(b)

		P	O	L	Y	N	O	M	I	A	L
	0	1	2	3	4	5	6	7	8	9	10
E	1	1	2	3	4	5	6	7	8	9	10
X	2	2	2	3	4	5	6	7	8	9	10
P	3	2	3	3	4	5	6	7	8	9	10
O	4	3	2	3	4	5	5	6	7	8	9
N	5	4	3	3	4	4	5	6	7	8	9
E	6	5	4	4	4	5	5	6	7	8	9
N	7	6	5	5	5	4	5	6	7	8	9
T	8	7	6	6	6	5	5	6	7	8	9
I	9	8	7	7	7	6	6	6	6	7	8
A	10	9	8	8	8	7	7	7	7	6	7
L	11	10	9	8	9	8	8	8	8	7	6

namely the empty string, and the first j letters of y: clearly, j. And similarly, $E(i, 0) = i$.

At this point, the algorithm for edit distance basically writes itself.

```
for i = 0, 1, 2, ..., m:
    E(i, 0) = i
for j = 1, 2, ..., n:
    E(0, j) = j
for i = 1, 2, ..., m:
    for j = 1, 2, ..., n:
        E(i, j) = min{E(i − 1, j) + 1, E(i, j − 1) + 1, E(i − 1, j − 1) + diff(i, j)}
return E(m, n)
```

This procedure fills in the table row by row, and left to right within each row. Each entry takes constant time to fill in, so the overall running time is just the size of the table, $O(mn)$.

And in our example, the edit distance turns out to be 6:

```
E  X  P  O  N  E  N  −  T  I  A  L
−  −  P  O  L  Y  N  O  M  I  A  L
```

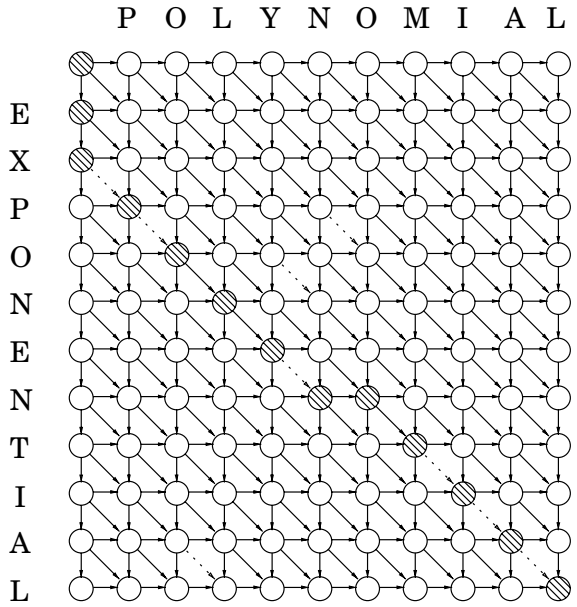
The underlying dag

Every dynamic program has an underlying dag structure: think of each node as representing a subproblem, and each edge as a precedence constraint on the order in which the subproblems can be tackled. Having nodes $u_1, \ldots, u_k$ point to v means "subproblem v can only be solved once the answers to $u_1, \ldots, u_k$ are known."

In our present edit distance application, the nodes of the underlying dag correspond to subproblems, or equivalently, to positions (i, j) in the table. Its edges are the precedence constraints, of the form $(i - 1, j) \to (i, j)$, $(i, j - 1) \to (i, j)$, and $(i - 1, j - 1) \to (i, j)$ (Figure 6.5). In fact, we can take things a little further and put weights on the edges so that the edit distances are given by shortest paths in the dag! To see this, set all edge lengths to 1, except for $\{(i - 1, j - 1) \to (i, j) : x[i] = y[j]\}$ (shown dotted in the figure), whose length is 0. The final answer is then simply the distance between nodes $s = (0, 0)$ and $t = (m, n)$. One possible shortest path is shown, the one that yields the alignment we found earlier. On this path, each move down is a deletion, each move right is an insertion, and each diagonal move is either a match or a substitution.

By altering the weights on this dag, we can allow generalized forms of edit distance, in which insertions, deletions, and substitutions have different associated costs.

Figure 6.5 The underlying dag, and a path of length 6.



6.4 Knapsack

During a robbery, a burglar finds much more loot than he had expected and has to decide what to take. His bag (or "knapsack") will hold a total weight of at most W pounds. There are n items to pick from, of weight $w_1, \ldots, w_n$ and dollar value $v_1, \ldots, v_n$. What's the most valuable combination of items he can fit into his bag?[1]

For instance, take $W = 10$ and

Item	Weight	Value
1	6	$30
2	3	$14
3	4	$16
4	2	$9

[1] If this application seems frivolous, replace "weight" with "CPU time" and "only W pounds can be taken" with "only W units of CPU time are available." Or use "bandwidth" in place of "CPU time," etc. The knapsack problem generalizes a wide variety of resource-constrained selection tasks.

Common subproblems

Finding the right subproblem takes creativity and experimentation. But there are a few standard choices that seem to arise repeatedly in dynamic programming.

i. The input is $x_1, x_2, \ldots, x_n$ and a subproblem is $x_1, x_2, \ldots, x_i$.

$$\boxed{x_1 \quad x_2 \quad x_3 \quad x_4 \quad x_5 \quad x_6} \quad x_7 \quad x_8 \quad x_9 \quad x_{10}$$

The number of subproblems is therefore linear.

ii. The input is $x_1, \ldots, x_n$, and $y_1, \ldots, y_m$. A subproblem is $x_1, \ldots, x_i$ and $y_1, \ldots, y_j$.

$$\boxed{x_1 \quad x_2 \quad x_3 \quad x_4 \quad x_5 \quad x_6} \quad x_7 \quad x_8 \quad x_9 \quad x_{10}$$

$$\boxed{y_1 \quad y_2 \quad y_3 \quad y_4 \quad y_5} \quad y_6 \quad y_7 \quad y_8$$

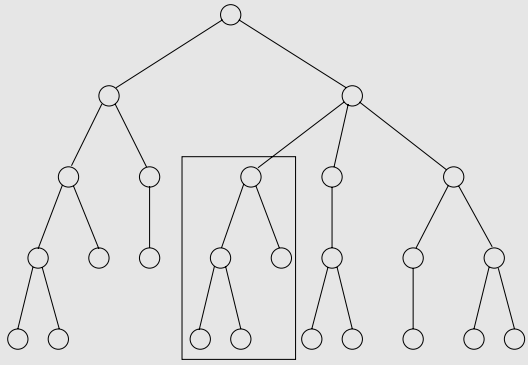
The number of subproblems is $O(mn)$.

iii. The input is $x_1, \ldots, x_n$ and a subproblem is $x_i, x_{i+1}, \ldots, x_j$.

$$x_1 \quad x_2 \quad \boxed{x_3 \quad x_4 \quad x_5 \quad x_6} \quad x_7 \quad x_8 \quad x_9 \quad x_{10}$$

The number of subproblems is $O(n^2)$.

iv. The input is a rooted tree. A subproblem is a rooted subtree.



If the tree has n nodes, how many subproblems are there?

We've already encountered the first two cases, and the others are coming up shortly.

Of mice and men

Our bodies are extraordinary machines: flexible in function, adaptive to new environments, and able to interact and reproduce. All these capabilities are specified by a program unique to each of us, a string that is 3 billion characters long over the alphabet $\{A, C, G, T\}$—our DNA.

The DNA sequences of any two people differ by only about 0.1%. However, this still leaves 3 million positions on which they vary, more than enough to explain the vast range of human diversity. These differences are of great scientific and medical interest—for instance, they might help predict which people are prone to certain diseases.

DNA is a vast and seemingly inscrutable program, but it can be broken down into smaller units that are more specific in their role, rather like subroutines. These are called *genes*. Computers have become a crucial tool in understanding the genes of humans and other organisms, to the extent that *computational genomics* is now a field in its own right. Here are examples of typical questions that arise.

1. When a new gene is discovered, one way to gain insight into its function is to find known genes that match it closely. This is particularly helpful in transferring knowledge from well-studied species, such as mice, to human beings.

   A basic primitive in this search problem is to define an efficiently computable notion of when two strings approximately match. The biology suggests a generalization of edit distance, and dynamic programming can be used to compute it.

   Then there's the problem of searching through the vast thicket of known genes: the database GenBank already has a total length of over 10^{10}, and this number is growing rapidly. The current method of choice is BLAST, a clever combination of algorithmic tricks and biological intuitions that has made it the most widely used software in computational biology.

2. Methods for *sequencing* DNA (that is, determining the string of characters that constitute it) typically only find fragments of 500–700 characters. Billions of these randomly scattered fragments can be generated, but how can they be assembled into a coherent DNA sequence? For one thing, the position of any one fragment in the final sequence is unknown and must be inferred by piecing together overlapping fragments.

   A showpiece of these efforts is the draft of human DNA completed in 2001 by two groups simultaneously: the publicly funded Human Genome Consortium and the private Celera Genomics.

3. When a particular gene has been sequenced in each of several species, can this information be used to reconstruct the evolutionary history of these species?

We will explore these problems in the exercises at the end of this chapter. Dynamic programming has turned out to be an invaluable tool for some of them and for computational biology in general.

There are two versions of this problem. If there are unlimited quantities of each item available, the optimal choice is to pick item 1 and two of item 4 (total: \$48). On the other hand, if there is one of each item (the burglar has broken into an art gallery, say), then the optimal knapsack contains items 1 and 3 (total: \$46).

As we shall see in Chapter 8, neither version of this problem is likely to have a polynomial-time algorithm. However, using dynamic programming they can both be solved in $O(nW)$ time, which is reasonable when W is small, but is not polynomial since the input size is proportional to $\log W$ rather than W.

Knapsack with repetition

Let's start with the version that allows repetition. As always, the main question in dynamic programming is, what are the subproblems? In this case we can shrink the original problem in two ways: we can either look at smaller knapsack capacities $w \leq W$, or we can look at fewer items (for instance, items $1, 2, \ldots, j$, for $j \leq n$). It usually takes a little experimentation to figure out exactly what works.

The first restriction calls for smaller capacities. Accordingly, define

$$K(w) = \text{maximum value achievable with a knapsack of capacity } w.$$

Can we express this in terms of smaller subproblems? Well, if the optimal solution to $K(w)$ includes item i, then removing this item from the knapsack leaves an optimal solution to $K(w - w_i)$. In other words, $K(w)$ is simply $K(w - w_i) + v_i$, for some i. We don't know which i, so we need to try all possibilities.

$$K(w) = \max_{i:w_i \leq w} \{K(w - w_i) + v_i\},$$

where as usual our convention is that the maximum over an empty set is 0. We're done! The algorithm now writes itself, and it is characteristically simple and elegant.

```
K(0) = 0
for w = 1 to W:
    K(w) = max{K(w − w_i) + v_i : w_i ≤ w}
return K(W)
```

This algorithm fills in a one-dimensional table of length $W + 1$, in left-to-right order. Each entry can take up to $O(n)$ time to compute, so the overall running time is $O(nW)$.

As always, there is an underlying dag. Try constructing it, and you will be rewarded with a startling insight: this particular variant of knapsack boils down to finding the longest path in a dag!

Knapsack without repetition

On to the second variant: what if repetitions are not allowed? Our earlier subproblems now become completely useless. For instance, knowing that the value $K(w - w_n)$ is very high doesn't help us, because we don't know whether or not item n already got used up in this partial solution. We must therefore refine our

concept of a subproblem to carry additional information about the items being used. We add a second parameter, $0 \leq j \leq n$:

$K(w, j)$ = maximum value achievable using a knapsack of capacity w and items $1, \ldots, j$.

The answer we seek is $K(W, n)$.

How can we express a subproblem $K(w, j)$ in terms of smaller subproblems? Quite simple: either item j is needed to achieve the optimal value, or it isn't needed:

$$K(w, j) = \max\{K(w - w_j, j - 1) + v_j, K(w, j - 1)\}.$$

(The first case is invoked only if $w_j \leq w$.) In other words, we can express $K(w, j)$ in terms of subproblems $K(\cdot, j - 1)$.

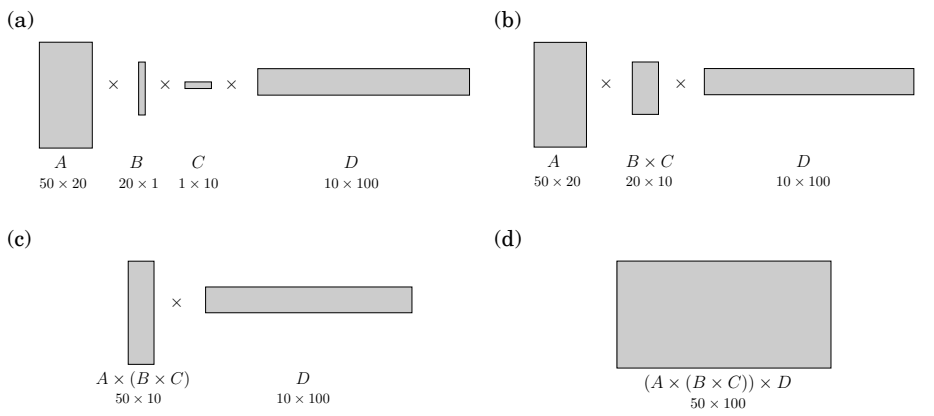
The algorithm then consists of filling out a two-dimensional table, with $W + 1$ rows and $n + 1$ columns. Each table entry takes just constant time, so even though the table is much larger than in the previous case, the running time remains the same, $O(nW)$. Here's the code.

```
Initialize all K(0, j) = 0 and all K(w, 0) = 0
for j = 1 to n:
   for w = 1 to W:
      if w_j > w:  K(w, j) = K(w, j − 1)
      else:  K(w, j) = max{K(w, j − 1), K(w − w_j, j − 1) + v_j}
return K(W, n)
```

6.5 Chain matrix multiplication

Suppose that we want to multiply four matrices, $A \times B \times C \times D$, of dimensions 50×20, 20×1, 1×10, and 10×100, respectively (Figure 6.6). This will involve iteratively multiplying two matrices at a time. Matrix multiplication is not

Figure 6.6 $A \times B \times C \times D = (A \times (B \times C)) \times D$.



Memoization

In dynamic programming, we write out a recursive formula that expresses large problems in terms of smaller ones and then use it to fill out a table of solution values in a bottom-up manner, from smallest subproblem to largest.

The formula also suggests a recursive algorithm, but we saw earlier that naive recursion can be terribly inefficient, because it solves the same subproblems over and over again. What about a more intelligent recursive implementation, one that remembers its previous invocations and thereby avoids repeating them?

On the knapsack problem (with repetitions), such an algorithm would use a hash table (recall Section 1.5) to store the values of $K(\cdot)$ that had already been computed. At each recursive call requesting some $K(w)$, the algorithm would first check if the answer was already in the table and then would proceed to its calculation only if it wasn't. This trick is called *memoization*:

```
A hash table, initially empty, holds values of K(w) indexed by w

function knapsack(w)
if w is in hash table: return K(w)
K(w) = max{knapsack(w − w_i) + v_i : w_i ≤ w}
insert K(w) into hash table, with key w
return K(w)
```

Since this algorithm never repeats a subproblem, its running time is $O(nW)$, just like the dynamic program. However, the constant factor in this big-O notation is substantially larger because of the overhead of recursion.

In some cases, though, memoization pays off. Here's why: dynamic programming automatically solves every subproblem that could conceivably be needed, while memoization only ends up solving the ones that are actually used. For instance, suppose that W and all the weights w_i are multiples of 100. Then a subproblem $K(w)$ is useless if 100 does not divide w. The memoized recursive algorithm will never look at these extraneous table entries.

commutative (in general, $A \times B \neq B \times A$), but it is *associative*, which means for instance that $A \times (B \times C) = (A \times B) \times C$. Thus we can compute our product of four matrices in many different ways, depending on how we parenthesize it. Are some of these better than others?

Multiplying an $m \times n$ matrix by an $n \times p$ matrix takes mnp multiplications, to a good enough approximation. Using this formula, let's compare several different ways of evaluating $A \times B \times C \times D$:

Parenthesization	Cost computation	Cost
$A \times ((B \times C) \times D)$	$20 \cdot 1 \cdot 10 + 20 \cdot 10 \cdot 100 + 50 \cdot 20 \cdot 100$	120, 200
$(A \times (B \times C)) \times D$	$20 \cdot 1 \cdot 10 + 50 \cdot 20 \cdot 10 + 50 \cdot 10 \cdot 100$	60, 200
$(A \times B) \times (C \times D)$	$50 \cdot 20 \cdot 1 + 1 \cdot 10 \cdot 100 + 50 \cdot 1 \cdot 100$	7, 000

As you can see, the order of multiplications makes a big difference in the final running time! Moreover, the natural *greedy* approach, to always perform the cheapest matrix multiplication available, leads to the second parenthesization shown here and is therefore a failure.

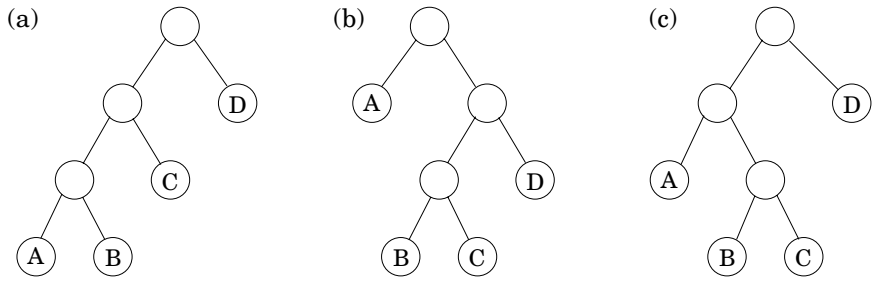
How do we determine the optimal order, if we want to compute $A_1 \times A_2 \times \cdots \times A_n$, where the A_i's are matrices with dimensions $m_0 \times m_1, m_1 \times m_2, \ldots, m_{n-1} \times m_n$, respectively? The first thing to notice is that a particular parenthesization can be represented very naturally by a binary tree in which the individual matrices correspond to the leaves, the root is the final product, and interior nodes are intermediate products (Figure 6.7). The possible orders in which to do the multiplication correspond to the various full binary trees with n leaves, whose number is exponential in n (Exercise 2.13). We certainly cannot try each tree, and with brute force thus ruled out, we turn to dynamic programming.

The binary trees of Figure 6.7 are suggestive: for a tree to be optimal, its subtrees must also be optimal. What are the subproblems corresponding to the subtrees? They are products of the form $A_i \times A_{i+1} \times \cdots \times A_j$. Let's see if this works: for $1 \leq i \leq j \leq n$, define

$$C(i, j) = \text{minimum cost of multiplying } A_i \times A_{i+1} \times \cdots \times A_j.$$

The size of this subproblem is the number of matrix multiplications, $|j - i|$. The smallest subproblem is when $i = j$, in which case there's nothing to multiply, so $C(i, i) = 0$. For $j > i$, consider the optimal subtree for $C(i, j)$. The first branch in this subtree, the one at the top, will split the product in two pieces, of the form $A_i \times \cdots \times A_k$ and $A_{k+1} \times \cdots \times A_j$, for some k between i and j. The cost of the subtree is then the cost of these two partial products, plus the cost of combining them: $C(i, k) + C(k + 1, j) + m_{i-1} \cdot m_k \cdot m_j$. And we just need to find the splitting

Figure 6.7 (a) $((A \times B) \times C) \times D$; (b) $A \times ((B \times C) \times D)$; (c) $(A \times (B \times C)) \times D$.



point k for which this is smallest:

$$C(i, j) = \min_{i \le k < j} \left\{ C(i, k) + C(k+1, j) + m_{i-1} \cdot m_k \cdot m_j \right\}.$$

We are ready to code! In the following, the variable s denotes subproblem size.

```
for i = 1 to n:  C(i, i) = 0
for s = 1 to n − 1:
  for i = 1 to n − s:
    j = i + s
    C(i, j) = min{C(i, k) + C(k + 1, j) + m_{i−1} · m_k · m_j : i ≤ k < j}
return C(1, n)
```

The subproblems constitute a two-dimensional table, each of whose entries takes $O(n)$ time to compute. The overall running time is thus $O(n^3)$.

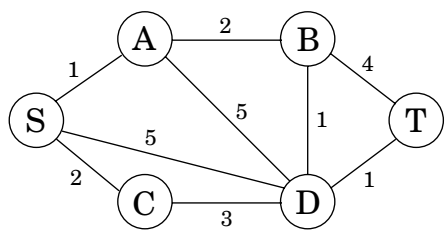
6.6 Shortest paths

We started this chapter with a dynamic programming algorithm for the elementary task of finding the shortest path in a dag. We now turn to more sophisticated shortest-path problems and see how these too can be accommodated by our powerful algorithmic technique.

Shortest reliable paths

Life is complicated, and abstractions such as graphs, edge lengths, and shortest paths rarely capture the whole truth. In a communications network, for example, even if edge lengths faithfully reflect transmission delays, there may be other considerations involved in choosing a path. For instance, each extra edge in the path might be an extra "hop" fraught with uncertainties and dangers of packet loss. In such cases, we would like to avoid paths with too many edges. Figure 6.8 illustrates this problem with a graph in which the shortest path from S to T has four edges, while there is another path that is a little longer but uses only two edges. If four edges translate to prohibitive unreliability, we may have to choose the latter path.

Figure 6.8 We want a path from s to t that is both short *and* has few edges.



Suppose then that we are given a graph G with lengths on the edges, along with two nodes s and t and an integer k, and we want the shortest path from s to t *that uses at most k edges*.

Is there a quick way to adapt Dijkstra's algorithm to this new task? Not quite: that algorithm focuses on the length of each shortest path without "remembering" the number of hops in the path, which is now a crucial piece of information.

In dynamic programming, the trick is to choose subproblems so that all vital information is remembered and carried forward. In this case, let us define, for each vertex v and each integer $i \leq k$, $\texttt{dist}(v, i)$ to be *the length of the shortest path from s to v that uses i edges*. The starting values $\texttt{dist}(v, 0)$ are ∞ for all vertices except s, for which it is 0. And the general update equation is, naturally enough,

$$\texttt{dist}(v, i) = \min_{(u,v)\in E} \{\texttt{dist}(u, i-1) + \ell(u, v)\}.$$

Need we say more?

All-pairs shortest paths

What if we want to find the shortest path not just between s and t but between *all* pairs of vertices? One approach would be to execute our general shortest-path algorithm from Section 4.6.1 (since there may be negative edges) $|V|$ times, once for each starting node. The total running time would then be $O(|V|^2|E|)$. We'll now see a better alternative, the $O(|V|^3)$ dynamic programming-based *Floyd-Warshall algorithm*.

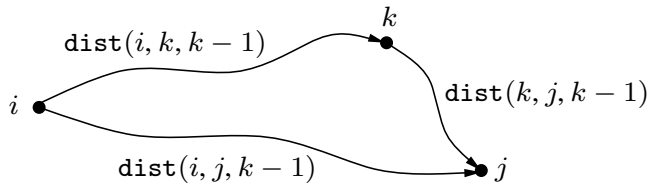
Is there a good subproblem for computing distances between all pairs of vertices in a graph? Simply solving the problem for more and more pairs or starting points is unhelpful, because it leads right back to the $O(|V|^2|E|)$ algorithm.

One idea comes to mind: the shortest path $u \to w_1 \to \cdots \to w_l \to v$ between u and v uses some number of intermediate nodes—possibly none. Suppose we disallow intermediate nodes altogether. Then we can solve all-pairs shortest paths at once: the shortest path from u to v is simply the direct edge (u, v), if it exists. What if we now gradually expand the *set of permissible intermediate nodes*? We can do this one node at a time, updating the shortest path lengths at each stage. Eventually this set grows to all of V, at which point all vertices are allowed to be on all paths, and we have found the true shortest paths between vertices of the graph!

More concretely, number the vertices in V as $\{1, 2, \ldots, n\}$, and let $\texttt{dist}(i, j, k)$ denote the length of the shortest path from i to j in which only nodes $\{1, 2, \ldots, k\}$ can be used as intermediates. Initially, $\texttt{dist}(i, j, 0)$ is the length of the direct edge between i and j, if it exists, and is ∞ otherwise.

What happens when we expand the intermediate set to include an extra node k? We must reexamine all pairs i, j and check whether using k as an intermediate point gives us a shorter path from i to j. But this is easy: a shortest path from i to j that uses k along with possibly other lower-numbered intermediate nodes goes through k just once (why? because we assume that there are no negative cycles). And we

have already calculated the length of the shortest path from i to k and from k to j using only lower-numbered vertices:



Thus, using k gives us a shorter path from i to j if and only if

$$\texttt{dist}(i, k, k-1) + \texttt{dist}(k, j, k-1) \;<\; \texttt{dist}(i, j, k-1),$$

in which case $\texttt{dist}(i, j, k)$ should be updated accordingly.

Here is the Floyd-Warshall algorithm—and as you can see, it takes $O(|V|^3)$ time.

```
for i = 1 to n:
    for j = 1 to n:
        dist(i, j, 0) = ∞
for all (i, j) ∈ E:
    dist(i, j, 0) = ℓ(i, j)
for k = 1 to n:
    for i = 1 to n:
        for j = 1 to n:
            dist(i, j, k) = min{dist(i, k, k − 1) + dist(k, j, k − 1),  dist(i, j, k − 1)}
```

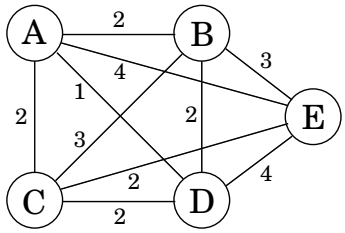
The traveling salesman problem

A traveling salesman is getting ready for a big sales tour. Starting at his hometown, suitcase in hand, he will conduct a journey in which each of his target cities is visited exactly once before he returns home. Given the pairwise distances between cities, what is the best order in which to visit them, so as to minimize the overall distance traveled?

Denote the cities by $1, \ldots, n$, the salesman's hometown being 1, and let $D = (d_{ij})$ be the matrix of intercity distances. The goal is to design a tour that starts and ends at 1, includes all other cities exactly once, and has minimum total length. Figure 6.9 shows an example involving five cities. Can you spot the optimal tour? Even in this tiny example, it is tricky for a human to find the solution; imagine what happens when hundreds of cities are involved.

It turns out this problem is also difficult for computers. In fact, the traveling salesman problem (TSP) is one of the most notorious computational tasks. There is a long history of attempts at solving it, a long saga of failures and partial successes, and along the way, major advances in algorithms and complexity theory. The most basic

Figure 6.9 The optimal traveling salesman tour has length 10.



piece of bad news about the TSP, which we will better understand in Chapter 8, is that it is highly unlikely to be solvable in polynomial time.

How long does it take, then? Well, the brute-force approach is to evaluate every possible tour and return the best one. Since there are $(n-1)!$ possibilities, this strategy takes $O(n!)$ time. We will now see that dynamic programming yields a much faster solution, though not a polynomial one.

What is the appropriate subproblem for the TSP? Subproblems refer to partial solutions, and in this case the most obvious partial solution is the initial portion of a tour. Suppose we have started at city 1 as required, have visited a few cities, and are now in city j. What information do we need in order to extend this partial tour? We certainly need to know j, since this will determine which cities are most convenient to visit next. And we also need to know all the cities visited so far, so that we don't repeat any of them. Here, then, is an appropriate subproblem.

> *For a subset of cities $S \subseteq \{1, 2, \ldots, n\}$ that includes 1, and $j \in S$, let $C(S, j)$ be the length of the shortest path visiting each node in S exactly once, starting at 1 and ending at j.*

When $|S| > 1$, we define $C(S, 1) = \infty$ since the path cannot both start and end at 1.

Now, let's express $C(S, j)$ in terms of smaller subproblems. We need to start at 1 and end at j; what should we pick as the second-to-last city? It has to be some $i \in S$, so the overall path length is the distance from 1 to i, namely, $C(S - \{j\}, i)$, plus the length of the final edge, d_{ij}. We must pick the best such i:

$$C(S, j) = \min_{i \in S : i \neq j} C(S - \{j\}, i) + d_{ij}.$$

On time and memory

The amount of time it takes to run a dynamic programming algorithm is easy to discern from the dag of subproblems: in many cases *it is just the total number of edges in the dag*! All we are really doing is visiting the nodes in linearized order, examining each node's inedges, and, most often, doing a constant amount of work per edge. By the end, each edge of the dag has been examined once.

But how much computer *memory* is required? There is no simple parameter of the dag characterizing this. It is certainly possible to do the job with an amount of memory proportional to the number of vertices (subproblems), but we can usually get away with much less. The reason is that the value of a particular subproblem only needs to be remembered until the larger subproblems depending on it have been solved. Thereafter, the memory it takes up can be released for reuse.

For example, in the Floyd-Warshall algorithm the value of $\texttt{dist}(i, j, k)$ is not needed once the $\texttt{dist}(\cdot, \cdot, k+1)$ values have been computed. Therefore, we only need two $|V| \times |V|$ arrays to store the `dist` values, one for odd values of k and one for even values: when computing $\texttt{dist}(i, j, k)$, we overwrite $\texttt{dist}(i, j, k-2)$.

(And let us not forget that, as always in dynamic programming, we also need one more array, $\texttt{prev}(i, j)$, storing the next to last vertex in the current shortest path from i to j, a value that must be updated with $\texttt{dist}(i, j, k)$. We omit this mundane but crucial bookkeeping step from our dynamic programming algorithms.)

Can you see why the edit distance dag in Figure 6.5 only needs memory proportional to the length of the shorter string?

The subproblems are ordered by $|S|$. Here's the code.

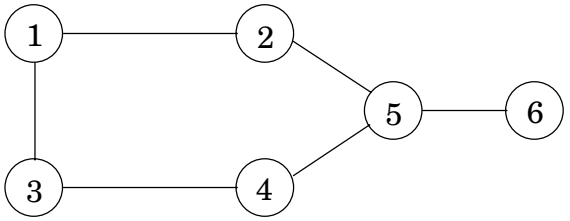
```
C({1}, 1) = 0
for s = 2 to n:
    for all subsets S ⊆ {1, 2, ..., n} of size s and containing 1:
        C(S, 1) = ∞
        for all j ∈ S, j ≠ 1:
            C(S, j) = min{C(S − {j}, i) + d_ij : i ∈ S, i ≠ j}
return min_j C({1, ..., n}, j) + d_j1
```

There are at most $2^n \cdot n$ subproblems, and each one takes linear time to solve. The total running time is therefore $O(n^2 2^n)$.

6.7 Independent sets in trees

A subset of nodes $S \subset V$ is an *independent set* of graph $G = (V, E)$ if there are no edges between them. For instance, in Figure 6.10 the nodes $\{1, 5\}$ form an independent set, but nodes $\{1, 4, 5\}$ do not, because of the edge between 4 and 5. The largest independent set is $\{2, 3, 6\}$.

Figure 6.10 The largest independent set in this graph has size 3.



Like several other problems we have seen in this chapter (knapsack, traveling salesman), finding the largest independent set in a graph is believed to be intractable. However, when the graph happens to be a *tree*, the problem can be solved in linear time, using dynamic programming. And what are the appropriate subproblems? Already in the chain matrix multiplication problem we noticed that the layered structure of a tree provides a natural definition of a subproblem—as long as one node of the tree has been identified as a root.

So here's the algorithm: Start by rooting the tree at any node r. Now, each node defines a subtree—the one hanging from it. This immediately suggests subproblems:

$$I(u) = \text{size of largest independent set of subtree hanging from } u.$$

Our final goal is $I(r)$.

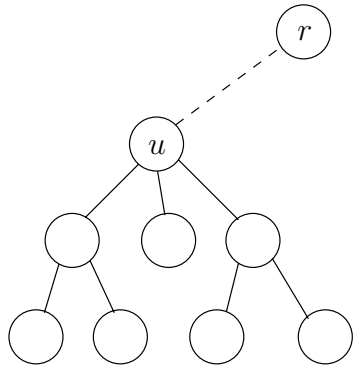
Dynamic programming proceeds as always from smaller subproblems to larger ones, that is to say, bottom-up in the rooted tree. Suppose we know the largest independent sets for all subtrees below a certain node u; in other words, suppose we know $I(w)$ for all descendants w of u. How can we compute $I(u)$? Let's split the computation into two cases: any independent set either includes u or it doesn't (Figure 6.11).

$$I(u) = \max\left\{1 + \sum_{\text{grandchildren } w \text{ of } u} I(w), \sum_{\text{children } w \text{ of } u} I(w)\right\}.$$

If the independent set includes u, then we get one point for it, but we aren't allowed to include the children of u—therefore we move on to the grandchildren. This is the first case in the formula. On the other hand, if we don't include u, then we don't get a point for it, but we can move on to its children.

The number of subproblems is exactly the number of vertices. With a little care, the running time can be made linear, $O(|V| + |E|)$.

Figure 6.11 $I(u)$ is the size of the largest independent set of the subtree rooted at u. Two cases: either u is in this independent set, or it isn't.



Exercises

6.1. A *contiguous subsequence* of a list S is a subsequence made up of consecutive elements of S. For instance, if S is

$$5, 15, -30, 10, -5, 40, 10,$$

then $15, -30, 10$ is a contiguous subsequence but $5, 15, 40$ is not. Give a linear-time algorithm for the following task:

Input: A list of numbers, $a_1, a_2, \ldots, a_n$.
Output: The contiguous subsequence of maximum sum (a subsequence of length zero has sum zero).

For the preceding example, the answer would be $10, -5, 40, 10$, with a sum of 55.

(*Hint:* For each $j \in \{1, 2, \ldots, n\}$, consider contiguous subsequences ending exactly at position j.)

6.2. You are going on a long trip. You start on the road at mile post 0. Along the way there are n hotels, at mile posts $a_1 < a_2 < \cdots < a_n$, where each a_i is measured from the starting point. The only places you are allowed to stop are at these hotels, but you can choose which of the hotels you stop at. You must stop at the final hotel (at distance a_n), which is your destination.

You'd ideally like to travel 200 miles a day, but this may not be possible (depending on the spacing of the hotels). If you travel x miles during a day, the *penalty* for that day is $(200 - x)^2$. You want to plan your trip so as to minimize the total penalty—that is, the sum, over all travel days, of the daily penalties. Give an efficient algorithm that determines the optimal sequence of hotels at which to stop.

6.3. Yuckdonald's is considering opening a series of restaurants along Quaint Valley Highway (QVH). The n possible locations are along a straight line, and the distances of these locations from the start of QVH are, in miles and in increasing order, $m_1, m_2, \ldots, m_n$. The constraints are as follows:

- At each location, Yuckdonald's may open at most one restaurant. The expected profit from opening a restaurant at location i is p_i, where $p_i > 0$ and $i = 1, 2, \ldots, n$.
- Any two restaurants should be at least k miles apart, where k is a positive integer.

Give an efficient algorithm to compute the maximum expected total profit subject to the given constraints.

6.4. You are given a string of n characters $s[1 \ldots n]$, which you believe to be a corrupted text document in which all punctuation has vanished (so that it looks something like "itwasthebestoftimes..."). You wish to reconstruct the document using a dictionary, which is available in the form of a Boolean function `dict`$(\cdot)$: for any string w,

$$\texttt{dict}(w) = \begin{cases} \texttt{true} & \text{if } w \text{ is a valid word} \\ \texttt{false} & \text{otherwise.} \end{cases}$$

(a) Give a dynamic programming algorithm that determines whether the string $s[\cdot]$ can be reconstituted as a sequence of valid words. The running time should be at most $O(n^2)$, assuming calls to `dict` take unit time.

(b) In the event that the string is valid, make your algorithm output the corresponding sequence of words.

6.5. *Pebbling a checkerboard.* We are given a checkerboard which has 4 rows and n columns, and has an integer written in each square. We are also given a set of $2n$ pebbles, and we want to place some or all of these on the checkerboard (each pebble can be placed on exactly one square) so as to maximize the sum of the integers in the squares that are covered by pebbles. There is one constraint: for a placement of pebbles to be legal, no two of them can be on horizontally or vertically adjacent squares (diagonal adjacency is fine).

(a) Determine the number of legal *patterns* that can occur in any column (in isolation, ignoring the pebbles in adjacent columns) and describe these patterns.

Call two patterns *compatible* if they can be placed on adjacent columns to form a legal placement. Let us consider subproblems consisting of the first k columns $1 \leq k \leq n$. Each subproblem can be assigned a *type*, which is the pattern occurring in the last column.

(b) Using the notions of compatibility and type, give an $O(n)$-time dynamic programming algorithm for computing an optimal placement.

6.6. Let us define a multiplication operation on three symbols a, b, c according to the following table; thus $ab = b$, $ba = c$, and so on. Notice that the multiplication operation defined by the table is neither associative nor commutative.

	a	b	c
a	b	b	a
b	c	b	a
c	a	c	c

Find an efficient algorithm that examines a string of these symbols, say *bbbbac*, and decides whether or not it is possible to parenthesize the string in such a way that the value of the resulting expression is a. For example, on input *bbbbac* your algorithm should return *yes* because $((b(bb))(ba))c = a$.

6.7. A subsequence is *palindromic* if it is the same whether read left to right or right to left. For instance, the sequence

$$A, C, G, T, G, T, C, A, A, A, A, T, C, G$$

has many palindromic subsequences, including A, C, G, C, A and A, A, A, A (on the other hand, the subsequence A, C, T is *not* palindromic). Devise an algorithm that takes a sequence $x[1 \ldots n]$ and returns the (length of the) longest palindromic subsequence. Its running time should be $O(n^2)$.

6.8. Given two strings $x = x_1 x_2 \cdots x_n$ and $y = y_1 y_2 \cdots y_m$, we wish to find the length of their *longest common substring*, that is, the largest k for which there are indices i and j with $x_i x_{i+1} \cdots x_{i+k-1} = y_j y_{j+1} \cdots y_{j+k-1}$. Show how to do this in time $O(mn)$.

6.9. A certain string-processing language offers a primitive operation which splits a string into two pieces. Since this operation involves copying the original string, it takes n units of time for a string of length n, regardless of the location of the cut. Suppose, now, that you want to break a string into many pieces. The order in which the breaks are made can affect the total running time. For example, if you want to cut a 20-character string at positions 3 and 10, then making the first cut at position 3 incurs a total cost of $20 + 17 = 37$, while doing position 10 first has a better cost of $20 + 10 = 30$.

Give a dynamic programming algorithm that, given the locations of m cuts in a string of length n, finds the minimum cost of breaking the string into $m + 1$ pieces.

6.10. *Counting heads.* Given integers n and k, along with $p_1, \ldots, p_n \in [0, 1]$, you want to determine the probability of obtaining exactly k heads when n biased coins are tossed independently at random, where p_i is the probability that the ith coin

comes up heads. Give an $O(nk)$ algorithm for this task.[2] Assume you can multiply and add two numbers in $[0, 1]$ in $O(1)$ time.

6.11. Given two strings $x = x_1 x_2 \cdots x_n$ and $y = y_1 y_2 \cdots y_m$, we wish to find the length of their *longest common subsequence*, that is, the largest k for which there are indices $i_1 < i_2 < \cdots < i_k$ and $j_1 < j_2 < \cdots < j_k$ with $x_{i_1} x_{i_2} \cdots x_{i_k} = y_{j_1} y_{j_2} \cdots y_{j_k}$. Show how to do this in time $O(mn)$.

6.12. You are given a convex polygon P on n vertices in the plane (specified by their x and y coordinates). A *triangulation* of P is a collection of $n-3$ diagonals of P such that no two diagonals intersect (except possibly at their endpoints). Notice that a triangulation splits the polygon's interior into $n-2$ disjoint triangles. The cost of a triangulation is the sum of the lengths of the diagonals in it. Give an efficient algorithm for finding a triangulation of minimum cost. (*Hint:* Label the vertices of P by $1, \ldots, n$, starting from an arbitrary vertex and walking clockwise. For $1 \le i < j \le n$, let the subproblem $A(i, j)$ denote the minimum cost triangulation of the polygon spanned by vertices $i, i+1, \ldots, j$.)

6.13. Consider the following game. A "dealer" produces a sequence $s_1 \cdots s_n$ of "cards," face up, where each card s_i has a value v_i. Then two players take turns picking a card from the sequence, but can only pick the first or the last card of the (remaining) sequence. The goal is to collect cards of largest total value. (For example, you can think of the cards as bills of different denominations.) Assume n is even.

(a) Show a sequence of cards such that it is not optimal for the first player to start by picking up the available card of larger value. That is, the natural *greedy* strategy is suboptimal.

(b) Give an $O(n^2)$ algorithm to compute an optimal strategy for the first player. Given the initial sequence, your algorithm should precompute in $O(n^2)$ time some information, and then the first player should be able to make each move optimally in $O(1)$ time by looking up the precomputed information.

6.14. *Cutting cloth.* You are given a rectangular piece of cloth with dimensions $X \times Y$, where X and Y are positive integers, and a list of n products that can be made using the cloth. For each product $i \in [1, n]$ you know that a rectangle of cloth of dimensions $a_i \times b_i$ is needed and that the final selling price of the product is c_i. Assume the a_i, b_i, and c_i are all positive integers. You have a machine that can cut any rectangular piece of cloth into two pieces either horizontally or vertically. Design an algorithm that determines the best return on the $X \times Y$ piece of cloth, that is, a strategy for cutting the cloth so that the products made from the resulting pieces give the maximum sum of selling prices. You are free to make as many copies of a given product as you wish, or none if desired.

6.15. Suppose two teams, A and B, are playing a match to see who is the first to win n games (for some particular n). We can suppose that A and B are equally

[2] In fact, there is also a $O(n \log^2 n)$ algorithm within your reach.

competent, so each has a 50% chance of winning any particular game. Suppose they have already played $i + j$ games, of which A has won i and B has won j. Give an efficient algorithm to compute the probability that A will go on to win the match. For example, if $i = n - 1$ and $j = n - 3$ then the probability that A will win the match is $7/8$, since it must win any of the next three games.

6.16. The *garage sale problem*. On a given Sunday morning, there are n garage sales going on, $g_1, g_2, \ldots, g_n$. For each garage sale g_j, you have an estimate of its value to you, v_j. For any two garage sales you have an estimate of the transportation cost d_{ij} of getting from g_i to g_j. You are also given the costs d_{0j} and d_{j0} of going between your home and each garage sale. You want to find a tour of a *subset* of the given garage sales, starting and ending at home, that maximizes your total benefit minus your total transportation costs.

Give an algorithm that solves this problem in time $O(n^2 2^n)$. (*Hint:* This is closely related to the traveling salesman problem.)

6.17. Given an unlimited supply of coins of denominations $x_1, x_2, \ldots, x_n$, we wish to make change for a value v; that is, we wish to find a set of coins whose total value is v. This might not be possible: for instance, if the denominations are 5 and 10 then we can make change for 15 but not for 12. Give an $O(nv)$ dynamic-programming algorithm for the following problem.

Input: $x_1, \ldots, x_n; v$.
Question: Is it possible to make change for v using coins of denominations $x_1, \ldots, x_n$?

6.18. Consider the following variation on the change-making problem (Exercise 6.17): you are given denominations $x_1, x_2, \ldots, x_n$, and you want to make change for a value v, but you are allowed to use each denomination *at most once*. For instance, if the denominations are 1, 5, 10, 20, then you can make change for $16 = 1 + 15$ and for $31 = 1 + 10 + 20$ but not for 40 (because you can't use 20 twice).

Input: Positive integers $x_1, x_2, \ldots, x_n$; another integer v.
Output: Can you make change for v, using each denomination x_i at most once?

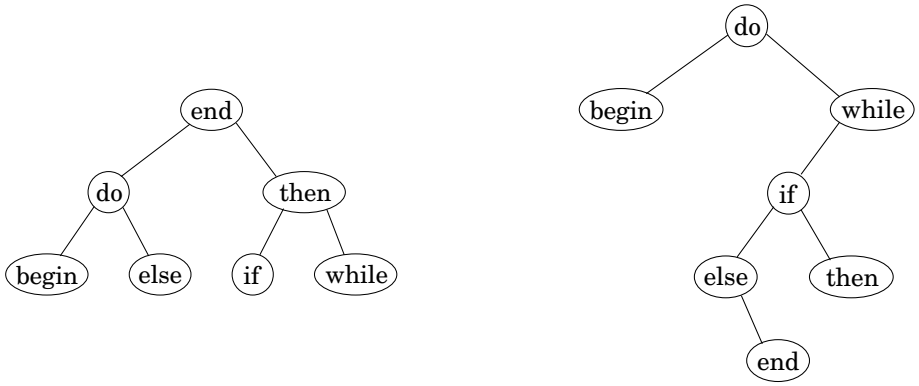
Show how to solve this problem in time $O(nv)$.

6.19. Here is yet another variation on the change-making problem (Exercise 6.17).

Given an unlimited supply of coins of denominations $x_1, x_2, \ldots, x_n$, we wish to make change for a value v using at most k coins; that is, we wish to find a set of $\leq k$ coins whose total value is v. This might not be possible: for instance, if the denominations are 5 and 10 and $k = 6$, then we can make change for 55 but not for 65. Give an efficient dynamic-programming algorithm for the following problem.

Input: $x_1, \ldots, x_n$; k; v.
Question: Is it possible to make change for v using at most k coins, of denominations $x_1, \ldots, x_n$?

Figure 6.12 Two binary search trees for the keywords of a programming language.



6.20. *Optimal binary search trees.* Suppose we know the frequency with which keywords occur in programs of a certain language, for instance:

begin	5%
do	40%
else	8%
end	4%
if	10%
then	10%
while	23%

We want to organize them in a *binary search tree*, so that the keyword in the root is alphabetically bigger than all the keywords in the left subtree and smaller than all the keywords in the right subtree (and this holds for all nodes).

Figure 6.12 has a nicely-balanced example on the left. In this case, when a keyword is being looked up, the number of comparisons needed is at most three: for instance, in finding "while", only the three nodes "end", "then", and "while" get examined. But since we know the frequency with which keywords are accessed, we can use an even more fine-tuned cost function, the *average number of comparisons* to look up a word. For the search tree on the left, it is

$$\text{cost} = 1(0.04) + 2(0.40 + 0.10) + 3(0.05 + 0.08 + 0.10 + 0.23) = 2.42.$$

By this measure, the best search tree is the one on the right, which has a cost of 2.18.

Give an efficient algorithm for the following task.

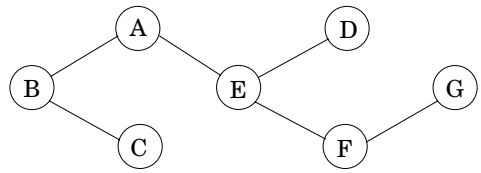
Input: n words (in sorted order); frequencies of these words*:* $p_1, p_2, \ldots, p_n$.

Output: The binary search tree of lowest cost (defined above as the expected number of comparisons in looking up a word).

6.21. A *vertex cover* of a graph $G = (V, E)$ is a subset of vertices $S \subseteq V$ that includes at least one endpoint of every edge in E. Give a linear-time algorithm for the following task.

Input: An undirected tree $T = (V, E)$.
Output: The size of the smallest vertex cover of T.

For instance, in the following tree, possible vertex covers include $\{A, B, C, D, E, F, G\}$ and $\{A, C, D, F\}$ but not $\{C, E, F\}$. The smallest vertex cover has size 3: $\{B, E, G\}$.



6.22. Give an $O(nt)$ algorithm for the following task.

Input: A list of n positive integers $a_1, a_2, \ldots, a_n$; a positive integer t.
Question: Does some subset of the a_i's add up to t? (You can use each a_i at most once.)

6.23. A mission-critical production system has n stages that have to be performed sequentially; stage i is performed by machine M_i. Each machine M_i has a probability r_i of functioning reliably and a probability $1 - r_i$ of failing (and the failures are independent). Therefore, if we implement each stage with a single machine, the probability that the whole system works is $r_1 \cdot r_2 \cdots r_n$. To improve this probability we add redundancy, by having m_i copies of the machine M_i that performs stage i. The probability that all m_i copies fail simultaneously is only $(1 - r_i)^{m_i}$, so the probability that stage i is completed correctly is $1 - (1 - r_i)^{m_i}$ and the probability that the whole system works is $\prod_{i=1}^{n}(1 - (1 - r_i)^{m_i})$. Each machine M_i has a cost c_i, and there is a total budget B to buy machines. (Assume that B and c_i are positive integers.)

Given the probabilities $r_1, \ldots, r_n$, the costs $c_1, \ldots, c_n$, and the budget B, find the redundancies $m_1, \ldots, m_n$ that are within the available budget and that maximize the probability that the system works correctly.

6.24. *Time and space complexity of dynamic programming.* Our dynamic programming algorithm for computing the edit distance between strings of length m and n creates a table of size $n \times m$ and therefore needs $O(mn)$ time and space. In practice, it will run out of space long before it runs out of time. How can this space requirement be reduced?

(a) Show that if we just want to compute the value of the edit distance (rather than the optimal sequence of edits), then only $O(n)$ space is

needed, because only a small portion of the table needs to be maintained at any given time.

(b) Now suppose that we also want the optimal sequence of edits. As we saw earlier, this problem can be recast in terms of a corresponding grid-shaped dag, in which the goal is to find the optimal path from node $(0, 0)$ to node (n, m). It will be convenient to work with this formulation, and while we're talking about convenience, we might as well also assume that m is a power of 2.

Let's start with a small addition to the edit distance algorithm that will turn out to be very useful. The optimal path in the dag must pass through an intermediate node $(k, m/2)$ for some k; show how the algorithm can be modified to also return this value k.

(c) Now consider a recursive scheme:

```
procedure find-path((0, 0) → (n, m))
compute the value k above
find-path((0, 0) → (k, m/2))
find-path((k, m/2) → (n, m))
concatenate these two paths, with k in the middle
```

Show that this scheme can be made to run in $O(mn)$ time and $O(n)$ space.

6.25. Consider the following 3-PARTITION problem. Given integers $a_1, \ldots, a_n$, we want to determine whether it is possible to partition $\{1, \ldots, n\}$ into three disjoint subsets I, J, K such that

$$\sum_{i \in I} a_i = \sum_{j \in J} a_j = \sum_{k \in K} a_k = \frac{1}{3} \sum_{i=1}^{n} a_i$$

For example, for input $(1, 2, 3, 4, 4, 5, 8)$ the answer is *yes*, because there is the partition $(1, 8)$, $(4, 5)$, $(2, 3, 4)$. On the other hand, for input $(2, 2, 3, 5)$ the answer is *no*.

Devise and analyze a dynamic programming algorithm for 3-PARTITION that runs in time polynomial in n and in $\sum_i a_i$.

6.26. *Sequence alignment.* When a new gene is discovered, a standard approach to understanding its function is to look through a database of known genes and find close matches. The closeness of two genes is measured by the extent to which they are *aligned*. To formalize this, think of a gene as being a long string over an alphabet $\Sigma = \{A, C, G, T\}$. Consider two genes (strings) $x = ATGCC$ and $y = TACGCA$. An alignment of x and y is a way of matching up these two strings by writing them in columns, for instance:

$$\begin{array}{cccccccc} - & A & T & - & G & C & C \\ T & A & - & C & G & C & A \end{array}$$

Here the "$-$" indicates a "gap." The characters of each string must appear in order, and each column must contain a character from at least one of the strings. The score of an alignment is specified by a scoring matrix δ of size $(|\Sigma| + 1) \times (|\Sigma| + 1)$, where the extra row and column are to accommodate gaps. For instance the preceding alignment has the following score:

$$\delta(-, T) + \delta(A, A) + \delta(T, -) + \delta(-, C) + \delta(G, G) + \delta(C, C) + \delta(C, A).$$

Give a dynamic programming algorithm that takes as input two strings $x[1 \ldots n]$ and $y[1 \ldots m]$ and a scoring matrix δ, and returns the highest-scoring alignment. The running time should be $O(mn)$.

6.27. *Alignment with gap penalties.* The alignment algorithm of Exercise 6.26 helps to identify DNA sequences that are close to one another. The discrepancies between these closely matched sequences are often caused by errors in DNA replication. However, a closer look at the biological replication process reveals that the scoring function we considered earlier has a qualitative problem: nature often inserts or removes entire substrings of nucleotides (creating long gaps), rather than editing just one position at a time. Therefore, the penalty for a gap of length 10 should not be 10 times the penalty for a gap of length 1, but something significantly smaller.

Repeat Exercise 6.26, but this time use a modified scoring function in which the penalty for a gap of length k is $c_0 + c_1 k$, where c_0 and c_1 are given constants (and c_0 is larger than c_1).

6.28. *Local sequence alignment.* Often two DNA sequences are significantly different, but contain regions that are very similar and are *highly conserved*. Design an algorithm that takes an input two strings $x[1 \ldots n]$ and $y[1 \ldots m]$ and a scoring matrix δ (as defined in Exercise 6.26), and outputs substrings x' and y' of x and y, respectively, that have the highest-scoring alignment over all pairs of such substrings. Your algorithm should take time $O(mn)$.

6.29. *Exon chaining.* Each gene corresponds to a subregion of the overall genome (the DNA sequence); however, part of this region might be "junk DNA." Frequently, a gene consists of several pieces called exons, which are separated by junk fragments called introns. This complicates the process of identifying genes in a newly sequenced genome.

Suppose we have a new DNA sequence and we want to check whether a certain gene (a string) is present in it. Because we cannot hope that the gene will be a contiguous subsequence, we look for partial matches—fragments of the DNA that are also present in the gene (actually, even these partial matches will be approximate, not perfect). We then attempt to assemble these fragments.

Let $x[1 \ldots n]$ denote the DNA sequence. Each partial match can be represented by a triple (l_i, r_i, w_i), where $x[l_i \ldots r_i]$ is the fragment and w_i is a weight

representing the strength of the match (it might be a local alignment score or some other statistical quantity). Many of these potential matches could be false, so the goal is to find a subset of the triples that are consistent (nonoverlapping) and have a maximum total weight.

Show how to do this efficiently.

6.30. *Reconstructing evolutionary trees by maximum parsimony.* Suppose we manage to sequence a particular gene across a whole bunch of different species. For concreteness, say there are n species, and the sequences are strings of length k over alphabet $\Sigma = \{A, C, G, T\}$. How can we use this information to reconstruct the evolutionary history of these species?

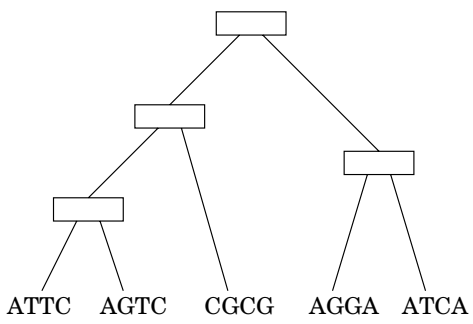
Evolutionary history is commonly represented by a tree whose leaves are the different species, whose root is their common ancestor, and whose internal branches represent speciation events (that is, moments when a new species broke off from an existing one). Thus we need to find the following:

- A (binary) evolutionary tree with the given species at the leaves.
- For each internal node, a string of length k: the gene sequence for that particular ancestor.

For each possible tree T, annotated with sequences $s(u) \in \Sigma^k$ at each of its nodes u, we can assign a score based on the principle of *parsimony*: fewer mutations are more likely.

$$\text{score}(T) = \sum_{(u,v)\in E(T)} (\text{number of positions on which } s(u) \text{ and } s(v) \text{ agree}).$$

Finding the highest-score tree is a difficult problem. Here we will consider just a small part of it: suppose we know the structure of the tree, and we want to fill in the sequences $s(u)$ of the internal nodes u. Here's an example with $k = 4$ and $n = 5$:



(a) In this particular example, there are several maximum parsimony reconstructions of the internal node sequences. Find one of them.

(b) Give an efficient (in terms of n and k) algorithm for this task. (*Hint:* Even though the sequences might be long, you can do just one position at a time.)

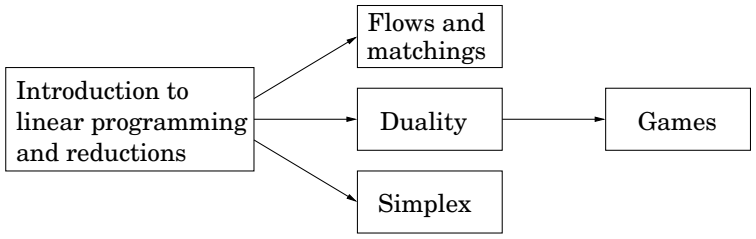
Chapter 7

Linear programming and reductions

Many of the problems for which we want algorithms are *optimization* tasks: the *shortest* path, the *cheapest* spanning tree, the *longest* increasing subsequence, and so on. In such cases, we seek a solution that (1) satisfies certain constraints (for instance, the path must use edges of the graph and lead from s to t, the tree must touch all nodes, the subsequence must be increasing); and (2) is the best possible, with respect to some well-defined criterion, among all solutions that satisfy these constraints.

Linear programming describes a broad class of optimization tasks in which both the constraints and the optimization criterion are *linear functions*. It turns out an enormous number of problems can be expressed in this way.

Given the vastness of its topic, this chapter is divided into several parts, which can be read separately subject to the following dependencies.



7.1 An introduction to linear programming

In a linear programming problem we are given a set of variables, and we want to assign real values to them so as to (1) satisfy a set of linear equations and/or linear inequalities involving these variables and (2) maximize or minimize a given linear objective function.

7.1.1 Example: profit maximization

A boutique chocolatier has two products: its flagship assortment of triangular chocolates, called *Pyramide*, and the more decadent and deluxe *Pyramide Nuit*. How much of each should it produce to maximize profits? Let's say it makes x_1 boxes of Pyramide per day, at a profit of \$1 each, and x_2 boxes of Nuit, at a more substantial profit of \$6 apiece; x_1 and x_2 are unknown values that we wish to determine. But this is not all; there are also some constraints on x_1 and x_2 that must be accommodated (besides the obvious one, $x_1, x_2 \geq 0$). First, the daily demand for these exclusive chocolates is limited to at most 200 boxes of Pyramide and 300 boxes of Nuit. Also, the current workforce can produce a total of at most 400 boxes of chocolate per day. What are the optimal levels of production?

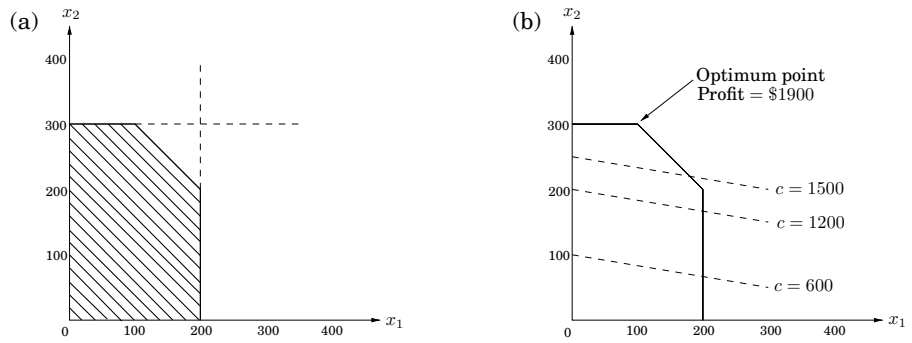
We represent the situation by a *linear program*, as follows.

$$\begin{array}{ll} \text{Objective function} & \max x_1 + 6x_2 \\ \text{Constraints} & x_1 \leq 200 \\ & x_2 \leq 300 \\ & x_1 + x_2 \leq 400 \\ & x_1, x_2 \geq 0 \end{array}$$

A linear equation in x_1 and x_2 defines a line in the two-dimensional (2D) plane, and a linear inequality designates a *half-space*, the region on one side of the line. Thus the set of all *feasible solutions* of this linear program, that is, the points (x_1, x_2) which satisfy all constraints, is the intersection of five half-spaces. It is a convex polygon, shown in Figure 7.1.

We want to find the point in this polygon at which the objective function—the profit—is maximized. The points with a profit of c dollars lie on the line $x_1 + 6x_2 = c$, which has a slope of $-1/6$ and is shown in Figure 7.1 for selected values of c. As c increases, this "profit line" moves parallel to itself, up and to the right. Since the goal

Figure 7.1 (a) The feasible region for a linear program. (b) Contour lines of the objective function: $x_1 + 6x_2 = c$ for different values of the profit c.



is to maximize c, we must move the line as far up as possible, while still touching the feasible region. The optimum solution will be the very last feasible point that the profit line sees and must therefore be a vertex of the polygon, as shown in the figure. If the slope of the profit line were different, then its last contact with the polygon could be an entire edge rather than a single vertex. In this case, the optimum solution would not be unique, but there would certainly be an optimum vertex.

It is a general rule of linear programs that the optimum is achieved at a vertex of the feasible region. The only exceptions are cases in which there is no optimum; this can happen in two ways:

1. The linear program is *infeasible*; that is, the constraints are so tight that it is impossible to satisfy all of them. For instance,

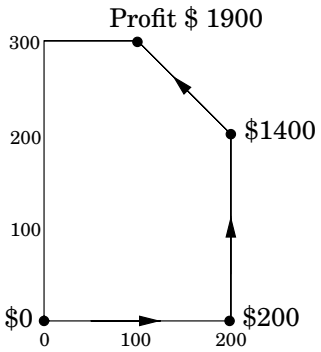
$$x \leq 1, \quad x \geq 2.$$

2. The constraints are so loose that the feasible region is *unbounded*, and it is possible to achieve arbitrarily high objective values. For instance,

$$\max \; x_1 + x_2$$
$$x_1, x_2 \geq 0.$$

Solving linear programs

Linear programs (LPs) can be solved by the *simplex method*, devised by George Dantzig in 1947. We shall explain it in more detail in Section 7.6, but briefly, this algorithm starts at a vertex, in our case perhaps $(0, 0)$, and repeatedly looks for an adjacent vertex (connected by an edge of the feasible region) of better objective value. In this way it does *hill-climbing* on the vertices of the polygon, walking from neighbor to neighbor so as to steadily increase profit along the way. Here's a possible trajectory.



Upon reaching a vertex that has no better neighbor, simplex declares it to be optimal and halts. Why does this *local* test imply *global* optimality? By simple geometry—think of the profit line passing through this vertex. Since all the vertex's neighbors lie below the line, the rest of the feasible polygon must also lie below this line.

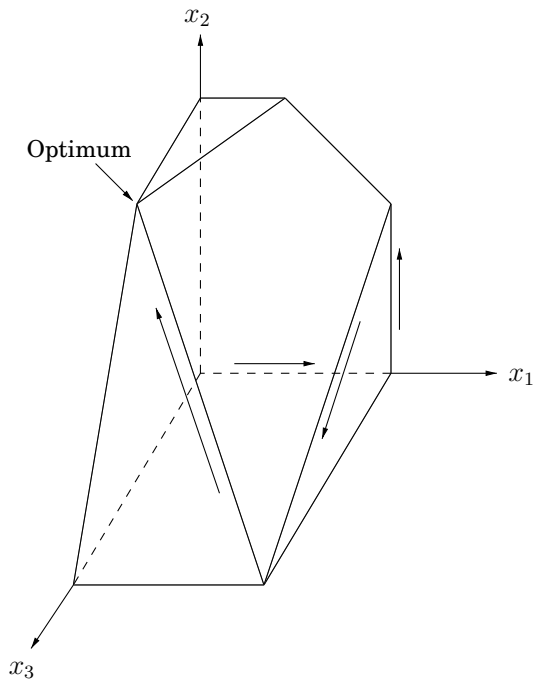
More products

Encouraged by consumer demand, the chocolatier decides to introduce a third and even more exclusive line of chocolates, called *Pyramide Luxe*. One box of these will bring in a profit of \$13. Let x_1, x_2, x_3 denote the number of boxes of each chocolate produced daily, with x_3 referring to Luxe. The old constraints on x_1 and x_2 persist, although the labor restriction now extends to x_3 as well: the sum of all three variables can be at most 400. What's more, it turns out that Nuit and Luxe require the same packaging machinery, except that Luxe uses it three times as much, which imposes another constraint $x_2 + 3x_3 \leq 600$. What are the best possible levels of production?

Here is the updated linear program.

$$\begin{aligned} \max \quad & x_1 + 6x_2 + 13x_3 \\ & x_1 \leq 200 \\ & x_2 \leq 300 \\ & x_1 + x_2 + x_3 \leq 400 \\ & x_2 + 3x_3 \leq 600 \\ & x_1, x_2, x_3 \geq 0 \end{aligned}$$

Figure 7.2 The feasible polyhedron for a three-variable linear program.



The space of solutions is now three-dimensional. Each linear equation defines a 3D plane, and each inequality a half-space on one side of the plane. The feasible region is an intersection of seven half-spaces, a polyhedron (Figure 7.2). Looking at the figure, can you decipher which inequality corresponds to each face of the polyhedron?

A profit of c corresponds to the plane $x_1 + 6x_2 + 13x_3 = c$. As c increases, this profit-plane moves parallel to itself, further and further into the positive orthant until it no longer touches the feasible region. The point of final contact is the optimal vertex: $(0, 300, 100)$, with total profit \$3100.

How would the simplex algorithm behave on this modified problem? As before, it would move from vertex to vertex, along edges of the polyhedron, increasing profit steadily. A possible trajectory is shown in Figure 7.2, corresponding to the following sequence of vertices and profits:

$$\underset{\$0}{(0,0,0)} \longrightarrow \underset{\$200}{(200,0,0)} \longrightarrow \underset{\$1400}{(200,200,0)} \longrightarrow \underset{\$2800}{(200,0,200)} \longrightarrow \underset{\$3100}{(0,300,100)}$$

Finally, upon reaching a vertex with no better neighbor, it would stop and declare this to be the optimal point. Once again by basic geometry, if all the vertex's neighbors lie on one side of the profit-plane, then so must the entire polyhedron.

A magic trick called duality

Here is why you should believe that $(0, 300, 100)$, with a total profit of \$3100, is the optimum: Look back at the linear program. Add the second inequality to the third, and add to them the fourth multiplied by 4. The result is the inequality $x_1 + 6x_2 + 13x_3 \leq 3100$.

Do you see? This inequality says that no feasible solution (values x_1, x_2, x_3 satisfying the constraints) can possibly have a profit greater than 3100. So we must indeed have found the optimum! The only question is, where did we get these mysterious multipliers $(0, 1, 1, 4)$ for the four inequalities?

In Section 7.4 we'll see that it is always possible to come up with such multipliers by solving another LP! Except that (it gets even better) we do not even need to solve this other LP, because it is in fact so intimately connected to the original one—it is called the *dual*—that solving the original LP solves the dual as well! But we are getting far ahead of our story.

What if we add a fourth line of chocolates, or hundreds more of them? Then the problem becomes high-dimensional, and hard to visualize. Simplex continues to work in this general setting, although we can no longer rely upon simple geometric intuitions for its description and justification. We will study the full-fledged simplex algorithm in Section 7.6.

In the meantime, we can rest assured in the knowledge that there are many professional, industrial-strength packages that implement simplex and take care of all the tricky details like numeric precision. In a typical application, the main task is therefore to correctly express the problem as a linear program. The package then takes care of the rest.

With this in mind, let's look at a high-dimensional application.

7.1.2 Example: production planning

This time, our company makes handwoven carpets, a product for which the demand is extremely seasonal. Our analyst has just obtained demand estimates for all months of the next calendar year: $d_1, d_2, \ldots, d_{12}$. As feared, they are very uneven, ranging from 440 to 920.

Here's a quick snapshot of the company. We currently have 30 employees, each of whom makes 20 carpets per month and gets a monthly salary of \$2,000. We have no initial surplus of carpets.

How can we handle the fluctuations in demand? There are three ways:

1. *Overtime*, but this is expensive since overtime pay is 80% more than regular pay. Also, workers can put in at most 30% overtime.
2. *Hiring and firing*, but these cost \$320 and \$400, respectively, per worker.
3. *Storing surplus production*, but this costs \$8 per carpet per month. We currently have no stored carpets on hand, and we must end the year without any carpets stored.

This rather involved problem can be formulated and solved as a linear program!

A crucial first step is defining the variables.

w_i = number of workers during ith month; $w_0 = 30$.
x_i = number of carpets made during ith month.
o_i = number of carpets made by overtime in month i.
h_i, f_i = number of workers hired and fired, respectively, at beginning of month i.
s_i = number of carpets stored at end of month i; $s_0 = 0$.

All in all, there are 72 variables (74 if you count w_0 and s_0).

We now write the constraints. First, all variables must be nonnegative:

$$w_i, x_i, o_i, h_i, f_i, s_i \geq 0, \quad i = 1, \ldots, 12.$$

The total number of carpets made per month consists of regular production plus overtime:

$$x_i = 20w_i + o_i$$

(one constraint for each $i = 1, \ldots, 12$). The number of workers can potentially change at the start of each month:

$$w_i = w_{i-1} + h_i - f_i.$$

The number of carpets stored at the end of each month is what we started with, plus the number we made, minus the demand for the month:

$$s_i = s_{i-1} + x_i - d_i.$$

And overtime is limited:

$$o_i \leq 6w_i.$$

Finally, what is the objective function? It is to minimize the total cost:

$$\min \ \ 2000 \sum_i w_i + 320 \sum_i h_i + 400 \sum_i f_i + 8 \sum_i s_i + 180 \sum_i o_i,$$

a linear function of the variables. Solving this linear program by simplex should take less than a second and will give us the optimum business strategy for our company.

Well, almost. The optimum solution might turn out to be *fractional*; for instance, it might involve hiring 10.6 workers in the month of March. This number would have to be rounded to either 10 or 11 in order to make sense, and the overall cost would then increase correspondingly. In the present example, most of the variables take on fairly large (double-digit) values, and thus rounding is unlikely to affect things too much. There are other LPs, however, in which rounding decisions have to be made very carefully in order to end up with an integer solution of reasonable quality.

In general, there is a tension in linear programming between the ease of obtaining fractional solutions and the desirability of integer ones. As we shall see in Chapter 8, finding the optimum integer solution of an LP is an important but very hard problem, called *integer linear programming*.

7.1.3 Example: optimum bandwidth allocation

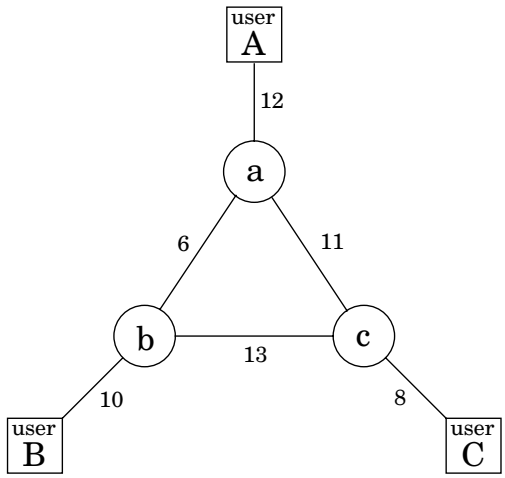
Next we turn to a miniaturized version of the kind of problem a network service provider might face.

Suppose we are managing a network whose lines have the bandwidths shown in Figure 7.3, and we need to establish three connections: between users A and B, between B and C, and between A and C. Each connection requires at least two units of bandwidth, but can be assigned more. Connection A–B pays \$3 per unit of bandwidth, and connections B–C and A–C pay \$2 and \$4, respectively.

Each connection can be routed in two ways, a long path and a short path, or by a combination: for instance, two units of bandwidth via the short route, one via the long route. How do we route these connections to maximize our network's revenue?

This is a linear program. We have variables for each connection and each path (long or short); for example, x_{AB} is the short-path bandwidth allocated to the connection between A and B, and x'_{AB} the long-path bandwidth for this same connection. We demand that no edge's bandwidth is exceeded and that each connection gets a

Figure 7.3 A communications network between three users A, B, and C. Bandwidths are shown.



bandwidth of at least 2 units.

$$
\begin{aligned}
\max \quad & 3x_{AB} + 3x'_{AB} + 2x_{BC} + 2x'_{BC} + 4x_{AC} + 4x'_{AC} \\
& x_{AB} + x'_{AB} + x_{BC} + x'_{BC} \le 10 \qquad [\text{edge } (b, B)] \\
& x_{AB} + x'_{AB} + x_{AC} + x'_{AC} \le 12 \qquad [\text{edge } (a, A)] \\
& x_{BC} + x'_{BC} + x_{AC} + x'_{AC} \le 8 \qquad [\text{edge } (c, C)] \\
& x_{AB} + x'_{BC} + x'_{AC} \le 6 \qquad [\text{edge } (a, b)] \\
& x'_{AB} + x_{BC} + x'_{AC} \le 13 \qquad [\text{edge } (b, c)] \\
& x'_{AB} + x'_{BC} + x_{AC} \le 11 \qquad [\text{edge } (a, c)] \\
& x_{AB} + x'_{AB} \ge 2 \\
& x_{BC} + x'_{BC} \ge 2 \\
& x_{AC} + x'_{AC} \ge 2 \\
& x_{AB}, x'_{AB}, x_{BC}, x'_{BC}, x_{AC}, x'_{AC} \ge 0
\end{aligned}
$$

Even a tiny example like this one is hard to solve on one's own (try it!), and yet the optimal solution is obtained instantaneously via simplex:

$$x_{AB} = 0,\ x'_{AB} = 7,\ x_{BC} = x'_{BC} = 1.5,\ x_{AC} = 0.5,\ x'_{AC} = 4.5.$$

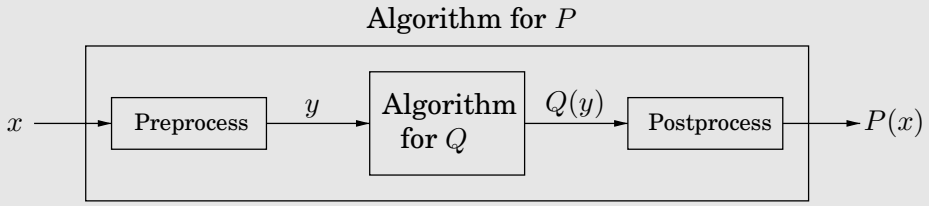
This solution is not integral, but in the present application we don't need it to be, and thus no rounding is required. Looking back at the original network, we see that every edge except a–c is used at full capacity.

Reductions

Sometimes a computational task is sufficiently general that any subroutine for it can also be used to solve a variety of other tasks, which at first glance might seem unrelated. For instance, we saw in Chapter 6 how an algorithm for finding the longest path in a dag can, surprisingly, also be used for finding longest increasing subsequences. We describe this phenomenon by saying that the longest increasing subsequence problem *reduces to* the longest path problem in a dag. In turn, the longest path in a dag reduces to the shortest path in a dag; here's how a subroutine for the latter can be used to solve the former:

```
function LONGEST PATH(G)
  negate all edge weights of G
  return SHORTEST PATH(G)
```

Let's step back and take a slightly more formal view of reductions. If any subroutine for task Q can also be used to solve P, we say P *reduces to* Q. Often, P is solvable by a single call to Q's subroutine, which means any instance x of P can be transformed into an instance y of Q such that $P(x)$ can be deduced from $Q(y)$:



(Do you see that the reduction from $P =$ LONGEST PATH to $Q =$ SHORTEST PATH follows this schema?) If the pre- and postprocessing procedures are efficiently computable then this creates an efficient algorithm for P out of *any* efficient algorithm for Q!

Reductions enhance the power of algorithms: Once we have an algorithm for problem Q (which could be shortest path, for example) we can use it to solve other problems. In fact, most of the computational tasks we study in this book are considered core computer science problems precisely because they arise in so many different applications, which is another way of saying that many problems reduce to them. This is especially true of linear programming.

One cautionary observation: our LP has one variable for every possible path between the users. In a larger network, there could easily be exponentially many such paths, and therefore this particular way of translating the network problem into an LP will not scale well. We will see a cleverer and more scalable formulation in Section 7.2.

Here's a parting question for you to consider. Suppose we removed the constraint that each connection should receive at least two units of bandwidth. Would the optimum change?

7.1.4 Variants of linear programming

As evidenced in our examples, a general linear program has many degrees of freedom.

1. It can be either a maximization or a minimization problem.
2. Its constraints can be equations and/or inequalities.
3. The variables are often restricted to be nonnegative, but they can also be unrestricted in sign.

We will now show that these various LP options *can all be reduced to one another* via simple transformations. Here's how.

1. To turn a maximization problem into a minimization (or vice versa), just multiply the coefficients of the objective function by -1.

2a. To turn an inequality constraint like $\sum_{i=1}^{n} a_i x_i \leq b$ into an equation, introduce a new variable s and use

$$\sum_{i=1}^{n} a_i x_i + s = b$$
$$s \geq 0.$$

This s is called the *slack variable* for the inequality. As justification, observe that a vector $(x_1, \ldots, x_n)$ satisfies the original inequality constraint if and only if there is some $s \geq 0$ for which it satisfies the new equality constraint.

2b. To change an equality constraint into inequalities is easy: rewrite $ax = b$ as the equivalent pair of constraints $ax \leq b$ and $ax \geq b$.

3. Finally, to deal with a variable x that is unrestricted in sign, do the following:
   - Introduce two nonnegative variables, $x^+, x^- \geq 0$.
   - Replace x, wherever it occurs in the constraints or the objective function, by $x^+ - x^-$.

   This way, x can take on any real value by appropriately adjusting the new variables. More precisely, any feasible solution to the original LP involving x can be mapped to a feasible solution of the new LP involving x^+, x^-, and vice versa.

By applying these transformations we can reduce any LP (maximization or minimization, with both inequalities and equations, and with both nonnegative and unrestricted variables) into an LP of a much more constrained kind that we call the *standard form*, in which the variables are all nonnegative, the constraints are all equations, and the objective function is to be minimized.

Matrix-vector notation

A linear function like $x_1 + 6x_2$ can be written as the dot product of two vectors

$$\mathbf{c} = \begin{pmatrix} 1 \\ 6 \end{pmatrix} \text{ and } \mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix},$$

denoted $\mathbf{c} \cdot \mathbf{x}$ or $\mathbf{c}^T\mathbf{x}$. Similarly, linear constraints can be compiled into matrix-vector form:

$$\begin{array}{r} x_1 \le 200 \\ x_2 \le 300 \\ x_1 + x_2 \le 400 \end{array} \quad \Longrightarrow \quad \underbrace{\begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{pmatrix}}_{\mathbf{A}} \underbrace{\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}}_{\mathbf{x}} \le \underbrace{\begin{pmatrix} 200 \\ 300 \\ 400 \end{pmatrix}}_{\mathbf{b}}.$$

Here each row of matrix $\mathbf{A}$ corresponds to one constraint: its dot product with $\mathbf{x}$ is at most the value in the corresponding row of $\mathbf{b}$. In other words, if the rows of $\mathbf{A}$ are the vectors $\mathbf{a}_1, \ldots, \mathbf{a}_m$, then the statement $\mathbf{Ax} \le \mathbf{b}$ is equivalent to

$$\mathbf{a}_i \cdot \mathbf{x} \le b_i \text{ for all } i = 1, \ldots, m.$$

With these notational conveniences, a generic LP can be expressed simply as

$$\begin{array}{c} \max \ \mathbf{c}^T\mathbf{x} \\ \mathbf{Ax} \le \mathbf{b} \\ \mathbf{x} \ge 0. \end{array}$$

For example, our first linear program gets rewritten thus:

$$\begin{array}{rl} \max & x_1 + 6x_2 \\ & x_1 \le 200 \\ & x_2 \le 300 \\ & x_1 + x_2 \le 400 \\ & x_1, x_2 \ge 0 \end{array} \quad \Longrightarrow \quad \begin{array}{rl} \min & -x_1 - 6x_2 \\ & x_1 + s_1 = 200 \\ & x_2 + s_2 = 300 \\ & x_1 + x_2 + s_3 = 400 \\ & x_1, x_2, s_1, s_2, s_3 \ge 0 \end{array}$$

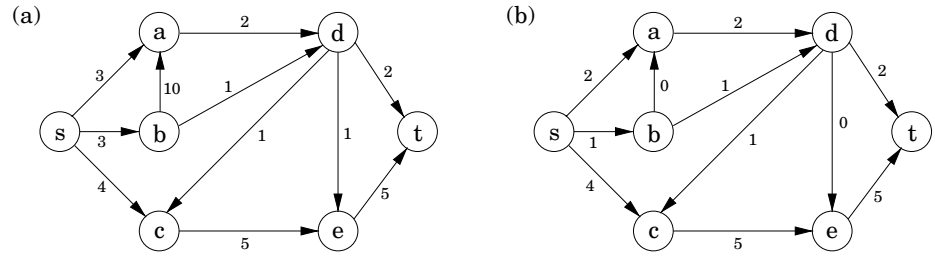
The original was also in a useful form: maximize an objective subject to certain inequalities. Any LP can likewise be recast in this way, using the reductions given earlier.

7.2 Flows in networks

7.2.1 Shipping oil

Figure 7.4(a) shows a directed graph representing a network of pipelines along which oil can be sent. The goal is to ship as much oil as possible from the *source* s to the *sink* t. Each pipeline has a maximum *capacity* it can handle, and there are

Figure 7.4 (a) A network with edge capacities. (b) A *flow* in the network.



no opportunities for storing oil en route. Figure 7.4(b) shows a possible *flow* from s to t, which ships 7 units in all. Is this the best that can be done?

7.2.2 Maximizing flow

The networks we are dealing with consist of a directed graph $G = (V, E)$; two special nodes $s, t \in V$, which are, respectively, a source and sink of G; and *capacities* $c_e > 0$ on the edges.

We would like to send as much oil as possible from s to t without exceeding the capacities of any of the edges. A particular shipping scheme is called a *flow* and consists of a variable f_e for each edge e of the network, satisfying the following two properties:

1. It doesn't violate edge capacities: $0 \leq f_e \leq c_e$ for all $e \in E$.
2. For all nodes u except s and t, the amount of flow entering u equals the amount leaving u:

$$\sum_{(w,u)\in E} f_{wu} = \sum_{(u,z)\in E} f_{uz}.$$

   In other words, flow is *conserved*.

The *size* of a flow is the total quantity sent from s to t and, by the conservation principle, is equal to the quantity leaving s:

$$\text{size}(f) = \sum_{(s,u)\in E} f_{su}.$$

In short, our goal is to assign values to $\{f_e : e \in E\}$ that will satisfy a set of linear constraints and maximize a linear objective function. But this is a linear program! *The maximum-flow problem reduces to linear programming.*

For example, for the network of Figure 7.4 the LP has 11 variables, one per edge. It seeks to maximize $f_{sa} + f_{sb} + f_{sc}$ subject to a total of 27 constraints: 11 for nonnegativity (such as $f_{sa} \geq 0$), 11 for capacity (such as $f_{sa} \leq 3$), and 5 for flow conservation (one for each node of the graph other than s and t, such as $f_{sc} + f_{dc} = f_{ce}$). Simplex

would take no time at all to correctly solve the problem and to confirm that, in our example, a flow of 7 is in fact optimal.

7.2.3 A closer look at the algorithm

All we know so far of the simplex algorithm is the vague geometric intuition that it keeps making local moves on the surface of a convex feasible region, successively improving the objective function until it finally reaches the optimal solution. Once we have studied it in more detail (Section 7.6), we will be in a position to understand exactly how it handles flow LPs, which is useful as a source of inspiration for designing *direct* max-flow algorithms.

It turns out that in fact the behavior of simplex has an elementary interpretation:

> Start with zero flow.
>
> *Repeat:* choose an appropriate path from s to t, and increase flow along the edges of this path as much as possible.

Figure 7.5(a)–(d) shows a small example in which simplex halts after two iterations. The final flow has size 2, which is easily seen to be optimal.

There is just one complication. What if we had initially chosen a different path, the one in Figure 7.5(e)? This gives only one unit of flow and yet seems to block all other paths. Simplex gets around this problem by also allowing paths to *cancel existing flow*. In this particular case, it would subsequently choose the path of Figure 7.5(f). Edge (b, a) of this path isn't in the original network and has the effect of canceling flow previously assigned to edge (a, b).

To summarize, in each iteration simplex looks for an $s - t$ path whose edges (u, v) can be of two types:

1. (u, v) is in the original network, and is not yet at full capacity.
2. The reverse edge (v, u) is in the original network, and there is some flow along it.

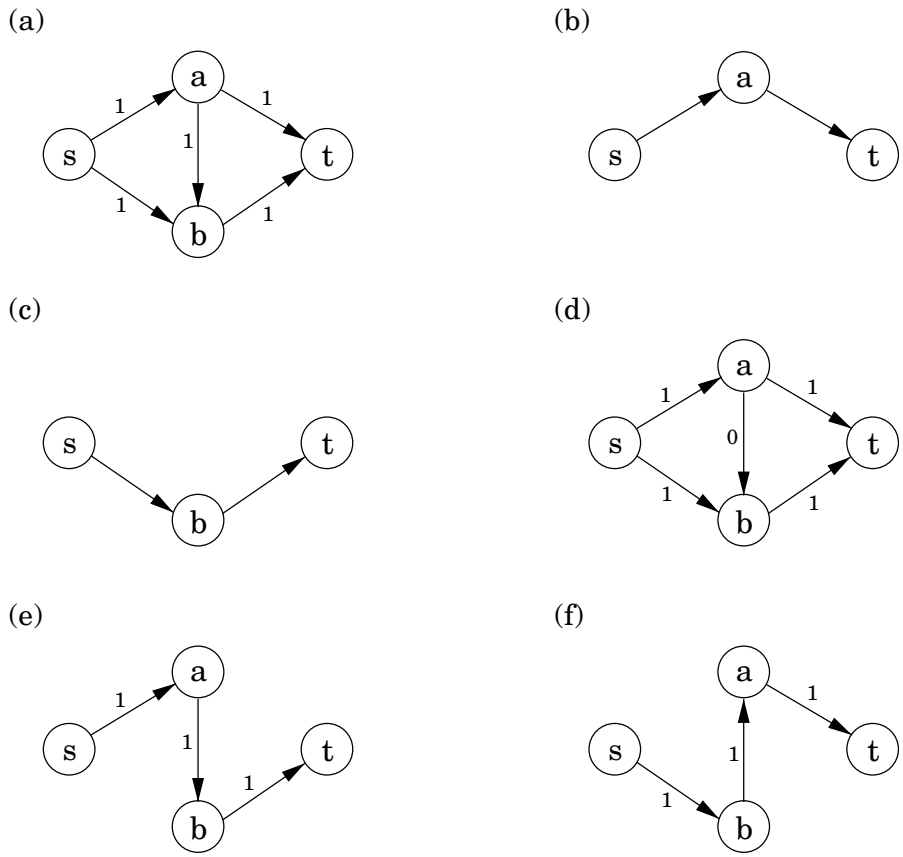
If the current flow is f, then in the first case, edge (u, v) can handle up to $c_{uv} - f_{uv}$ additional units of flow, and in the second case, up to f_{vu} additional units (canceling all or part of the existing flow on (v, u)). These flow-increasing opportunities can be captured in a *residual network* $G^f = (V, E^f)$, which has exactly the two types of edges listed, with residual capacities c^f:

$$\begin{cases} c_{uv} - f_{uv} & \text{if } (u, v) \in E \text{ and } f_{uv} < c_{uv} \\ f_{vu} & \text{if } (v, u) \in E \text{ and } f_{vu} > 0. \end{cases}$$

Thus we can equivalently think of simplex as choosing an $s - t$ path in the residual network.

By simulating the behavior of simplex, we get a direct algorithm for solving max-flow. It proceeds in iterations, each time explicitly constructing G^f, finding a suitable

Figure 7.5 An illustration of the max-flow algorithm. (a) A toy network. (b) The first path chosen. (c) The second path chosen. (d) The final flow. (e) We could have chosen this path first. (f) In which case, we would have to allow this second path.



$s - t$ path in G^f by using, say, a linear-time breadth-first search, and halting if there is no longer any such path along which flow can be increased.

Figure 7.6 illustrates the algorithm on our oil example.

7.2.4 A certificate of optimality

Now for a truly remarkable fact: not only does simplex correctly compute a maximum flow, but it also generates a short proof of the optimality of this flow!

Let's see an example of what this means. Partition the nodes of the oil network (Figure 7.4) into two groups, $L = \{s, a, b\}$ and $R = \{c, d, e, t\}$:

Figure 7.6 The max-flow algorithm applied to the network of Figure 7.4. At each iteration, the current flow is shown on the left and the residual network on the right. The paths chosen are shown in bold.

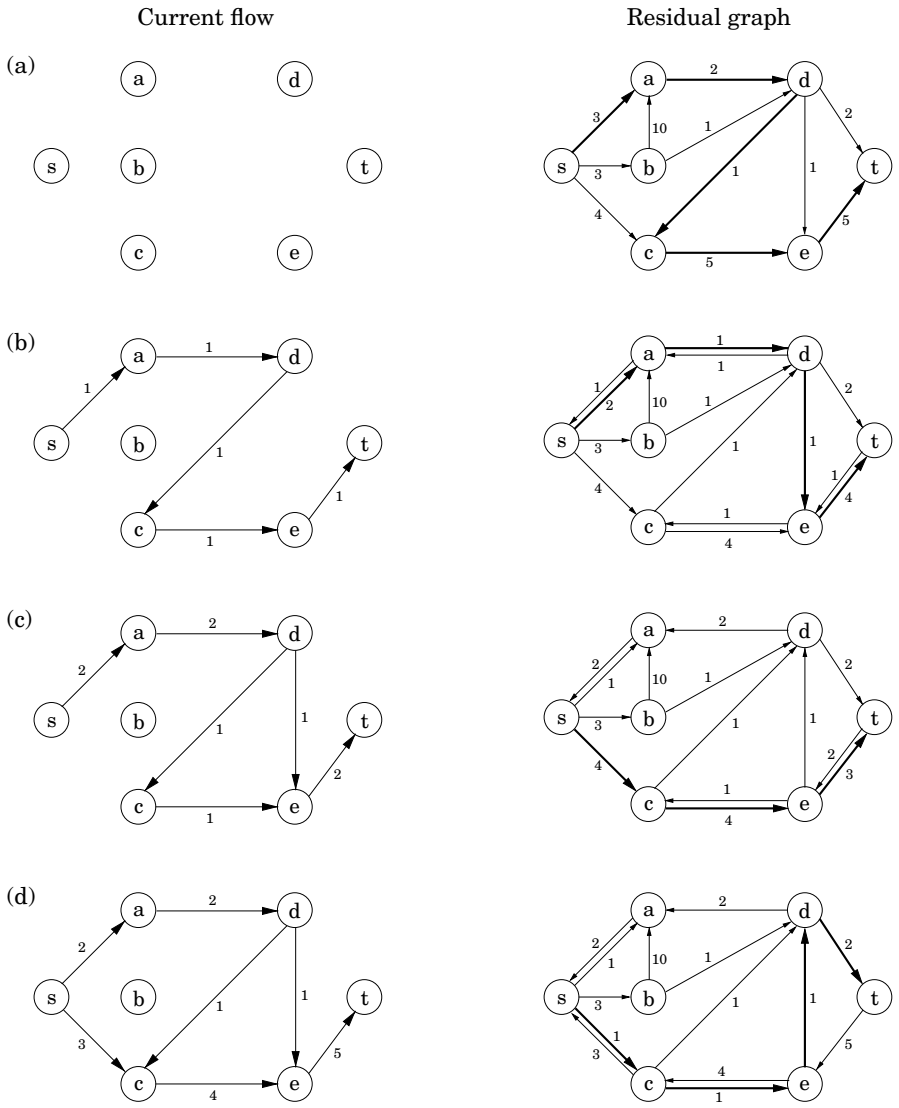
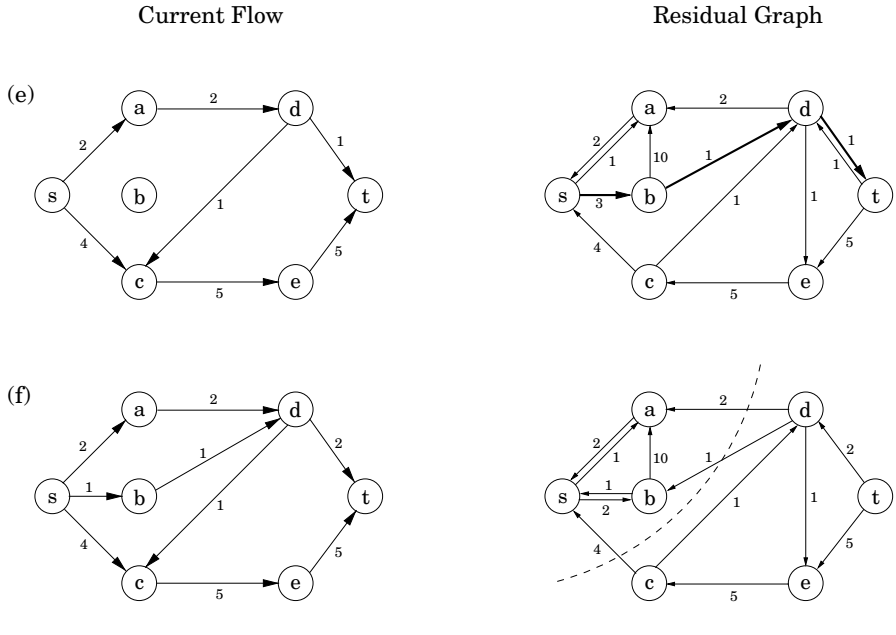
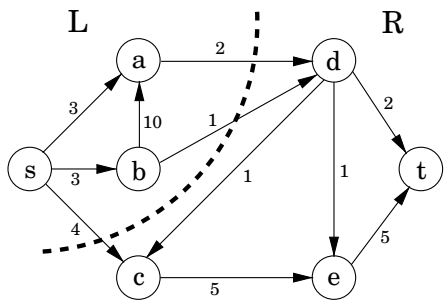


Figure 7.6 *Continued*





Any oil transmitted must pass from L to R. Therefore, no flow can possibly exceed the total capacity of the edges from L to R, which is 7. But this means that the flow we found earlier, of size 7, must be optimal!

More generally, an (s, t)-*cut* partitions the vertices into two disjoint groups L and R such that s is in L and t is in R. Its *capacity* is the total capacity of the edges from L to R, and as argued previously, is an upper bound on *any* flow:

Pick any flow f and any (s, t)-cut (L, R). Then size $(f) \leq$ capacity(L, R).

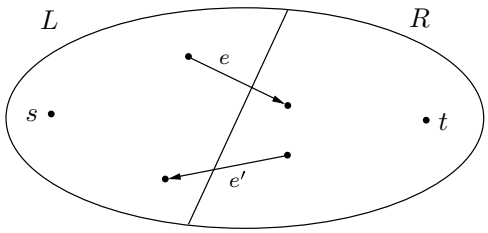
Some cuts are large and give loose upper bounds—cut $(\{s, b, c\}, \{a, d, e, t\})$ has a capacity of 19. But there is also a cut of capacity 7, which is effectively a *certificate of*

optimality of the maximum flow. This isn't just a lucky property of our oil network; such a cut *always* exists.

Max-flow min-cut theorem:
The size of the maximum flow in a network equals the capacity of the smallest (s, t)*-cut.*

Moreover, our algorithm automatically finds this cut as a by-product!

Let's see why this is true. Suppose f is the final flow when the algorithm terminates. We know that node t is no longer reachable from s in the residual network G^f. Let L be the nodes that *are* reachable from s in G^f, and let $R = V - L$ be the rest of the nodes. Then (L, R) is a cut in the graph G:



We claim that

$$\text{size}(f) \quad = \quad \text{capacity}(L, R).$$

To see this, observe that by the way L is defined, any edge going from L to R must be at full capacity (in the current flow f), and any edge from R to L must have zero flow. (So, in the figure, $f_e = c_e$ and $f_{e'} = 0$.) Therefore the net flow across (L, R) is exactly the capacity of the cut.

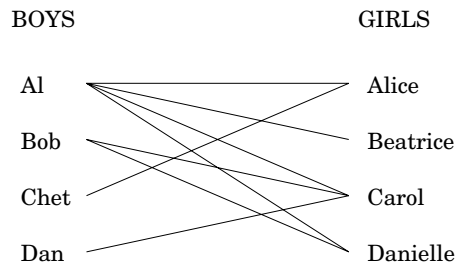
7.2.5 Efficiency

Each iteration of our maximum-flow algorithm is efficient, requiring $O(|E|)$ time if a depth-first or breadth-first search is used to find an $s - t$ path. But how many iterations are there?

Suppose all edges in the original network have *integer* capacities $\leq C$. Then an inductive argument shows that on each iteration of the algorithm, the flow is always an integer and increases by an integer amount. Therefore, since the maximum flow is at most $C|E|$ (why?), it follows that the number of iterations is at most this much. But this is hardly a reassuring bound: what if C is in the millions?

We examine this issue further in Exercise 7.31. It turns out that it is indeed possible to construct bad examples in which the number of iterations is proportional to C, *if* $s - t$ paths are not carefully chosen. However, if paths are chosen in a sensible manner—in particular, by using a breadth-first search, which finds the path with the fewest edges—then the number of iterations is at most $O(|V| \cdot |E|)$, no matter what the capacities are. This latter bound gives an overall running time of $O(|V| \cdot |E|^2)$ for maximum flow.

Figure 7.7 An edge between two people means they like each other. Is it possible to pair everyone up happily?

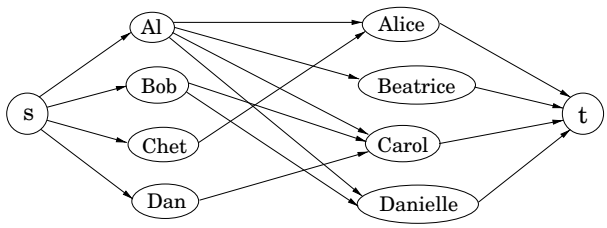


7.3 Bipartite matching

Figure 7.7 shows a graph with four nodes on the left representing boys and four nodes on the right representing girls.[1] There is an edge between a boy and girl if they like each other (for instance, Al likes all the girls). Is it possible to choose couples so that everyone has exactly one partner, and it is someone they like? In graph-theoretic jargon, is there a *perfect matching*?

This matchmaking game can be reduced to the maximum-flow problem, and thereby to linear programming! Create a new source node, s, with outgoing edges to all the boys; a new sink node, t, with incoming edges from all the girls; and direct all the edges in the original bipartite graph from boy to girl (Figure 7.8). Finally, give every edge a capacity of 1. Then there is a perfect matching if and only if this network has a flow whose size equals the number of couples. Can you find such a flow in the example?

Figure 7.8 A matchmaking network. Each edge has a capacity of one.



Actually, the situation is slightly more complicated than just stated: what is easy to see is that the optimum *integer-valued* flow corresponds to the optimum matching.

[1] This kind of graph, in which the nodes can be partitioned into two groups such that all edges are *between* the groups, is called *bipartite*.

We would be at a bit of a loss interpreting a flow that ships 0.7 units along the edge Al–Carol, for instance! Fortunately, the maximum-flow problem has the following

Property: *if all edge capacities are integers, then the optimal flow found by our algorithm is integral. We can see this directly from the algorithm, which in such cases would increment the flow by an integer amount on each iteration.*

Hence integrality comes for free in the maximum-flow problem. Unfortunately, this is the exception rather than the rule: as we will see in Chapter 8, it is a very difficult problem to find the optimum solution (or for that matter, *any* solution) of a general linear program, if we also demand that the variables be integers.

7.4 Duality

We have seen that in networks, flows are smaller than cuts, but the maximum flow and minimum cut exactly coincide and each is therefore a certificate of the other's optimality. Remarkable as this phenomenon is, we now generalize it from maximum flow to *any* problem that can be solved by linear programming! It turns out that every linear maximization problem has a *dual* minimization problem, and they relate to each other in much the same way as flows and cuts.

To understand what duality is about, recall our introductory LP with the two types of chocolate:

$$\begin{aligned} \max\ x_1 + 6x_2 & \\ x_1 &\leq 200 \\ x_2 &\leq 300 \\ x_1 + x_2 &\leq 400 \\ x_1, x_2 &\geq 0. \end{aligned}$$

Simplex declares the optimum solution to be $(x_1, x_2) = (100, 300)$, with objective value 1900. Can this answer be checked somehow? Let's see: suppose we take the first inequality and add it to six times the second inequality. We get

$$x_1 + 6x_2 \ \leq\ 2000.$$

This is interesting, because it tells us that it is impossible to achieve a profit of more than 2000. Can we add together some other combination of the LP constraints and bring this upper bound even closer to 1900? After a little experimentation, we find that multiplying the three inequalities by 0, 5, and 1, respectively, and adding them up yields

$$x_1 + 6x_2 \ \leq\ 1900.$$

So 1900 must indeed be the best possible value! The multipliers $(0, 5, 1)$ magically constitute a *certificate of optimality*! It is remarkable that such a certificate exists for this LP—and even if we knew there were one, how would we systematically go about finding it?

Let's investigate the issue by describing what we expect of these three multipliers, call them y_1, y_2, y_3.

Multiplier	Inequality
y_1	$x_1 \leq 200$
y_2	$x_2 \leq 300$
y_3	$x_1 + x_2 \leq 400$

To start with, these y_i's must be nonnegative, for otherwise they are unqualified to multiply inequalities (multiplying an inequality by a negative number would flip the $\leq$ to $\geq$). After the multiplication and addition steps, we get the bound:

$$(y_1 + y_3)x_1 + (y_2 + y_3)x_2 \leq 200y_1 + 300y_2 + 400y_3.$$

We want the left-hand side to look like our objective function $x_1 + 6x_2$ so that the right-hand side is an upper bound on the optimum solution. For this we need $y_1 + y_3$ to be 1 and $y_2 + y_3$ to be 6. Come to think of it, it would be fine if $y_1 + y_3$ were larger than 1—the resulting certificate would be all the more convincing. Thus, we get an upper bound

$$x_1 + 6x_2 \leq 200y_1 + 300y_2 + 400y_3 \quad \text{if} \quad \left\{ \begin{array}{c} y_1, y_2, y_3 \geq 0 \\ y_1 + y_3 \geq 1 \\ y_2 + y_3 \geq 6 \end{array} \right\}.$$

We can easily find y's that satisfy the inequalities on the right by simply making them large enough, for example $(y_1, y_2, y_3) = (5, 3, 6)$. But these particular multipliers would tell us that the optimum solution of the LP is at most $200 \cdot 5 + 300 \cdot 3 + 400 \cdot 6 = 4300$, a bound that is far too loose to be of interest. What we want is a bound that is as tight as possible, so we should minimize $200y_1 + 300y_2 + 400y_3$ subject to the preceding inequalities. *And this is a new linear program*!

Therefore, finding the set of multipliers that gives the best upper bound on our original LP is tantamount to solving a new LP:

$$\begin{aligned} \min \ 200y_1 + 300y_2 + 400y_3 \\ y_1 + y_3 \geq 1 \\ y_2 + y_3 \geq 6 \\ y_1, y_2, y_3 \geq 0 \end{aligned}$$

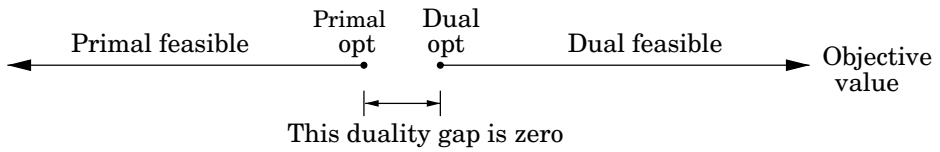
By design, any feasible value of this *dual* LP is an upper bound on the original *primal* LP. So if we somehow find a pair of primal and dual feasible values that are equal, then they must both be optimal. Here is just such a pair:

$$\text{Primal} : (x_1, x_2) = (100, 300); \quad \text{Dual} : (y_1, y_2, y_3) = (0, 5, 1).$$

They both have value 1900, and therefore they certify each other's optimality (Figure 7.9).

Amazingly, this is not just a lucky example, but a general phenomenon. To start with, the preceding construction—creating a multiplier for each primal constraint;

Figure 7.9 By design, dual feasible values $\geq$ primal feasible values. The duality theorem tells us that moreover their optima coincide.



writing a constraint in the dual for every variable of the primal, in which the sum is required to be above the objective coefficient of the corresponding primal variable; and optimizing the sum of the multipliers weighted by the primal right-hand sides—can be carried out for any LP, as shown in Figure 7.10, and in even greater generality in Figure 7.11. The second figure has one noteworthy addition: if the primal has an equality constraint, then the corresponding multiplier (or *dual variable*) need not be nonnegative, because the validity of equations is preserved when multiplied by negative numbers. So, the multipliers of equations are unrestricted variables. Notice also the simple symmetry between the two LPs, in that the matrix $A = (a_{ij})$ defines one primal constraint with each of its *rows*, and one dual constraint with each of its *columns*.

By construction, any feasible solution of the dual is an upper bound on any feasible solution of the primal. But moreover, their optima coincide!

Duality theorem: *If a linear program has a bounded optimum, then so does its dual, and the two optimum values coincide.*

When the primal is the LP that expresses the max-flow problem, it is possible to assign interpretations to the dual variables that show the dual to be none other than the minimum-cut problem (Exercise 7.25). The relation between flows and cuts is therefore just a specific instance of the duality theorem. And in fact, the proof of this theorem falls out of the simplex algorithm, in much the same way as the max-flow min-cut theorem fell out of the analysis of the max-flow algorithm.

Figure 7.10 A generic primal LP in matrix-vector form, and its dual.

Primal LP:

$$\begin{aligned} \max\ & \mathbf{c}^T\mathbf{x} \\ & \mathbf{A}\mathbf{x} \leq \mathbf{b} \\ & \mathbf{x} \geq 0 \end{aligned}$$

Dual LP:

$$\begin{aligned} \min\ & \mathbf{y}^T\mathbf{b} \\ & \mathbf{y}^T\mathbf{A} \geq \mathbf{c}^T \\ & \mathbf{y} \geq 0 \end{aligned}$$

Figure 7.11 In the most general case of linear programming, we have a set I of inequalities and a set E of equalities (a total of $m = |I| + |E|$ constraints) over n variables, of which a subset N are constrained to be nonnegative. The dual has $m = |I| + |E|$ variables, of which only those corresponding to I have nonnegativity constraints.

Primal LP:

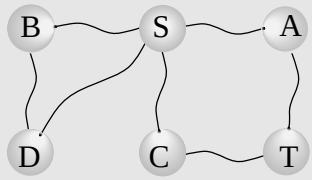
$$\begin{aligned} &\max\ c_1x_1 + \cdots + c_nx_n \\ &a_{i1}x_1 + \cdots + a_{in}x_n \le b_i \quad \text{for } i \in I \\ &a_{i1}x_1 + \cdots + a_{in}x_n = b_i \quad \text{for } i \in E \\ &x_j \ge 0 \quad \text{for } j \in N \end{aligned}$$

Dual LP:

$$\begin{aligned} &\min\ b_1y_1 + \cdots + b_my_m \\ &a_{1j}y_1 + \cdots + a_{mj}y_m \ge c_j \quad \text{for } j \in N \\ &a_{1j}y_1 + \cdots + a_{mj}y_m = c_j \quad \text{for } j \notin N \\ &y_i \ge 0 \quad \text{for } i \in I \end{aligned}$$

Visualizing duality

One can solve the shortest-path problem by the following "analog" device: Given a weighted undirected graph, build a *physical model* of it in which each edge is a string of length equal to the edge's weight, and each node is a knot at which the appropriate endpoints of strings are tied together. Then to find the shortest path from s to t, just *pull* s away from t until the gadget is taut. It is intuitively clear that this finds the shortest path from s to t.



There is nothing remarkable or surprising about all this until we notice the following: the shortest-path problem is a *minimization* problem, right? Then why are we *pulling* s away from t, an act whose purpose is, obviously, *maximization*? Answer: By pulling s away from t we solve *the dual* of the shortest-path problem! This dual has a very simple form (Exercise 7.28), with one variable x_u for each node u:

$$\max\ x_s - x_t$$
$$|x_u - x_v| \le w_{uv} \quad \text{for all edges } \{u, v\}.$$

In words, the dual problem is to stretch s and t as far apart as possible, subject to the constraint that the endpoints of any edge $\{u, v\}$ are separated by a distance of at most w_{uv}.

7.5 Zero-sum games

We can represent various conflict situations in life by *matrix games*. For example, the schoolyard *rock-paper-scissors* game is specified by the *payoff matrix* illustrated here. There are two players, called Row and Column, and they each pick a move

from $\{r, p, s\}$. They then look up the matrix entry corresponding to their moves, and Column pays Row this amount. It is Row's gain and Column's loss.

$G =$

		Column		
		r	p	s
Row	r	0	-1	1
	p	1	0	-1
	s	-1	1	0

Now suppose the two of them play this game repeatedly. If Row always makes the same move, Column will quickly catch on and will always play the countermove, winning every time. Therefore Row should mix things up: we can model this by allowing Row to have a *mixed strategy*, in which on each turn she plays r with probability x_1, p with probability x_2, and s with probability x_3. This strategy is specified by the vector $\mathbf{x} = (x_1, x_2, x_3)$, positive numbers that add up to 1. Similarly, Column's mixed strategy is some $\mathbf{y} = (y_1, y_2, y_3)$.[2]

On any given round of the game, there is an $x_i y_j$ chance that Row and Column will play the ith and jth moves, respectively. Therefore the *expected* (average) payoff is

$$\sum_{i,j} G_{ij} \cdot \text{Prob[Row plays } i\text{, Column plays } j] \;=\; \sum_{i,j} G_{ij} x_i y_j.$$

Row wants to *maximize* this, while Column wants to *minimize* it. What payoffs can they hope to achieve in rock-paper-scissors? Well, suppose for instance that Row plays the "completely random" strategy $\mathbf{x} = (1/3, 1/3, 1/3)$. If Column plays r, then the average payoff (reading the first column of the game matrix) will be

$$\frac{1}{3} \cdot 0 + \frac{1}{3} \cdot 1 + \frac{1}{3} \cdot -1 \;=\; 0.$$

This is also true if Column plays p, or s. And since the payoff of any mixed strategy (y_1, y_2, y_3) is just a weighted average of the individual payoffs for playing r, p, and s, it must also be zero. This can be seen directly from the preceding formula,

$$\sum_{i,j} G_{ij} x_i y_j \;=\; \sum_{i,j} G_{ij} \cdot \frac{1}{3} y_j \;=\; \sum_j y_j \left(\sum_i \frac{1}{3} G_{ij} \right) \;=\; \sum_j y_j \cdot 0 \;=\; 0,$$

where the second-to-last equality is the observation that every column of G adds up to zero. Thus by playing the "completely random" strategy, Row forces an expected payoff of zero, *no matter what Column does*. This means that Column cannot hope for a negative (expected) payoff (remember that he wants the payoff to be as small as possible). But symmetrically, if Column plays the completely random strategy, he also forces an expected payoff of zero, and thus Row cannot hope for a positive (expected) payoff. In short, the best each player can do is to play completely

[2] Also of interest are scenarios in which players alter their strategies from round to round, but these can get very complicated and are a vast subject unto themselves.

randomly, with an expected payoff of zero. We have mathematically confirmed what you knew all along about rock-paper-scissors!

Let's think about this in a slightly different way, by considering two scenarios:

1. First Row announces her strategy, and then Column picks his.
2. First Column announces his strategy, and then Row chooses hers.

We've seen that the average payoff is the same (zero) in either case if both parties play optimally. But this might well be due to the high level of symmetry in rock-paper-scissors. In general games, we'd expect the first option to favor Column, since he knows Row's strategy and can fully exploit it while choosing his own. Likewise, we'd expect the second option to favor Row. Amazingly, this is not the case: if both play optimally, then it doesn't hurt a player to announce his or her strategy in advance! What's more, this remarkable property is a consequence of—and in fact equivalent to—linear programming duality.

Let's investigate this with a nonsymmetric game. Imagine a *presidential election* scenario in which there are two candidates for office, and the moves they make correspond to campaign issues on which they can focus (the initials stand for *economy*, *society*, *morality*, and *tax cut*). The payoff entries are millions of votes lost by Column.

$$G = \begin{array}{|c|cc|} \hline & m & t \\ \hline e & 3 & -1 \\ s & -2 & 1 \\ \hline \end{array}$$

Suppose Row announces that she will play the mixed strategy $\mathrm{x} = (1/2, 1/2)$. What should Column do? Move m will incur an expected loss of $1/2$, while t will incur an expected loss of 0. The best response of Column is therefore the *pure* strategy $\mathrm{y} = (0, 1)$.

More generally, once Row's strategy $\mathrm{x} = (x_1, x_2)$ is fixed, there is always a *pure* strategy that is optimal for Column: either move m, with payoff $3x_1 - 2x_2$, or t, with payoff $-x_1 + x_2$, whichever is smaller. After all, any mixed strategy y is a weighted average of these two pure strategies and thus cannot beat the better of the two.

Therefore, if Row is forced to announce $\mathbf{x}$ before Column plays, she knows that his best response will achieve an expected payoff of $\min\{3x_1 - 2x_2, -x_1 + x_2\}$. She should choose $\mathbf{x}$ *defensively* to maximize her payoff against this best response:

$$\text{Pick } (x_1, x_2) \text{ that maximizes } \underbrace{\min\{3x_1 - 2x_2, -x_1 + x_2\}}_{\text{payoff from Column's best response to } \mathbf{x}}$$

This choice of x_i's gives Row the best possible *guarantee* about her expected payoff. And we will now see that it can be found by an LP! The main trick is to notice that

for *fixed* x_1 and x_2 the following are equivalent:

$$z = \min\{3x_1 - 2x_2, -x_1 + x_2\} \qquad \begin{aligned} &\max\ z \\ &z \le 3x_1 - 2x_2 \\ &z \le -x_1 + x_2 \end{aligned}$$

And Row needs to choose x_1 and x_2 to maximize this z.

$$\begin{array}{rcrcrcl} & & \max & & z & & \\ -3x_1 & + & 2x_2 & + & z & \le & 0 \\ x_1 & - & x_2 & + & z & \le & 0 \\ x_1 & + & x_2 & & & = & 1 \\ & & & & x_1, x_2 & \ge & 0 \end{array}$$

Symmetrically, if Column has to announce his strategy first, his best bet is to choose the mixed strategy y that minimizes his loss under Row's best response, in other words,

$$\text{Pick } (y_1, y_2) \text{ that minimizes} \qquad \underbrace{\max\{3y_1 - y_2, -2y_1 + y_2\}}_{\text{outcome of Row's best response to } \mathbf{y}}$$

In LP form, this is

$$\begin{array}{rcrcrcl} & & \min & & w & & \\ -3y_1 & + & y_2 & + & w & \ge & 0 \\ 2y_1 & - & y_2 & + & w & \ge & 0 \\ y_1 & + & y_2 & & & = & 1 \\ & & & & y_1, y_2 & \ge & 0 \end{array}$$

The crucial observation now is that *these two LPs are dual to each other* (see Figure 7.11)! Hence, they have the same optimum, call it V.

Let us summarize. By solving an LP, Row (the maximizer) can determine a strategy for herself that guarantees an expected outcome of at least V no matter what Column does. And by solving the dual LP, Column (the minimizer) can guarantee an expected outcome of at most V, no matter what Row does. It follows that this is the uniquely defined optimal play: a priori it wasn't even certain that such a play existed. V is known as the *value* of the game. In our example, it is $1/7$ and is realized when Row plays her optimum mixed strategy $(3/7, 4/7)$ and Column plays his optimum mixed strategy $(2/7, 5/7)$.

This example is easily generalized to arbitrary games and shows the existence of mixed strategies that are optimal for both players and achieve the same value—a fundamental result of game theory called the *min-max theorem*. It can be written in equation form as follows:

$$\max_{\mathbf{x}} \min_{\mathbf{y}} \sum_{i,j} G_{ij} x_i y_j \;=\; \min_{\mathbf{y}} \max_{\mathbf{x}} \sum_{i,j} G_{ij} x_i y_j.$$

This is surprising, because the left-hand side, in which Row has to announce her strategy first, should presumably be better for Column than the right-hand side, in which he has to go first. Duality equalizes the two, as it did with maximum flows and minimum cuts.

7.6 The simplex algorithm

The extraordinary power and expressiveness of linear programs would be little consolation if we did not have a way to solve them efficiently. This is the role of the simplex algorithm.

At a high level, the simplex algorithm takes a set of linear inequalities and a linear objective function and finds the optimal feasible point by the following strategy:

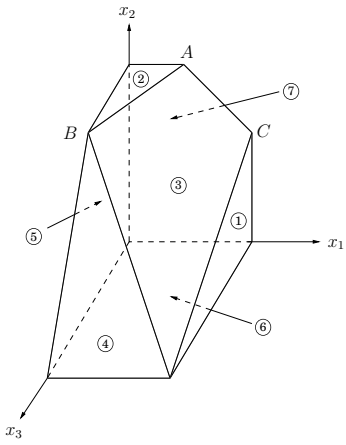
```
let v be any vertex of the feasible region
while there is a neighbor v' of v with better objective value:
  set v = v'
```

In our 2D and 3D examples (Figure 7.1 and Figure 7.2), this was simple to visualize and made intuitive sense. But what if there are n variables, $x_1, \ldots, x_n$?

Any setting of the x_i's can be represented by an n-tuple of real numbers and plotted in n-dimensional space. A linear equation involving the x_i's defines a *hyperplane* in this same space $\mathbb{R}^n$, and the corresponding linear inequality defines a *half-space*, all points that are either precisely on the hyperplane or lie on one particular side of it. Finally, the feasible region of the linear program is specified by a set of inequalities and is therefore the intersection of the corresponding half-spaces, a convex polyhedron.

But what do the concepts of *vertex* and *neighbor* mean in this general context?

Figure 7.12 A polyhedron defined by seven inequalities.



$$\begin{aligned} \max \quad & x_1 + 6x_2 + 13x_3 & \\ & x_1 \leq 200 & ① \\ & x_2 \leq 300 & ② \\ & x_1 + x_2 + x_3 \leq 400 & ③ \\ & x_2 + 3x_3 \leq 600 & ④ \\ & x_1 \geq 0 & ⑤ \\ & x_2 \geq 0 & ⑥ \\ & x_3 \geq 0 & ⑦ \end{aligned}$$

7.6.1 Vertices and neighbors in n-dimensional space

Figure 7.12 recalls an earlier example. Looking at it closely, we see that *each vertex is the unique point at which some subset of hyperplanes meet*. Vertex A, for instance, is the sole point at which constraints ②, ③, and ⑦ are satisfied with equality. On the other hand, the hyperplanes corresponding to inequalities ④ and ⑥ do not define a vertex, because their intersection is not just a single point but an entire line.

Let's make this definition precise.

> *Pick a subset of the inequalities. If there is a unique point that satisfies them with equality, and this point happens to be feasible, then it is a* vertex.

How many equations are needed to uniquely identify a point? When there are n variables, we need at least n linear equations if we want a unique solution. On the other hand, having more than n equations is redundant: at least one of them can be rewritten as a linear combination of the others and can therefore be disregarded. In short,

> Each vertex is specified by a set of n inequalities.[3]

A notion of *neighbor* now follows naturally.

> Two vertices are *neighbors* if they have $n-1$ defining inequalities in common.

In Figure 7.12, for instance, vertices A and C share the two defining inequalities {③, ⑦} and are thus neighbors.

7.6.2 The algorithm

On each iteration, simplex has two tasks:

1. Check whether the current vertex is optimal (and if so, halt).
2. Determine where to move next.

As we will see, both tasks are easy if the vertex happens to be at the origin. And if the vertex is elsewhere, we will transform the coordinate system to move it to the origin!

First let's see why the origin is so convenient. Suppose we have some generic LP

$$\begin{aligned} \max\ & \mathbf{c}^T\mathbf{x} \\ & \mathbf{A}\mathbf{x} \le \mathbf{b} \\ & \mathbf{x} \ge 0 \end{aligned}$$

where x is the vector of variables, $\mathrm{x} = (x_1, \ldots, x_n)$. Suppose the origin is feasible. Then it is certainly a vertex, since it is the unique point at which the n inequalities $\{x_1 \ge 0, \ldots, x_n \ge 0\}$ are *tight*. Now let's solve our two tasks. Task 1:

> The origin is optimal if and only if all $c_i \le 0$.

[3] There is one tricky issue here. It is possible that the same vertex might be generated by different subsets of inequalities. In Figure 7.12, vertex B is generated by {②, ③, ④}, but also by {②, ④, ⑤}. Such vertices are called *degenerate* and require special consideration. Let's assume for the time being that they don't exist, and we'll return to them later.

If all $c_i \leq 0$, then considering the constraints $\mathrm{x} \geq 0$, we can't hope for a better objective value. Conversely, if some $c_i > 0$, then the origin is not optimal, since we can increase the objective function by raising x_i.

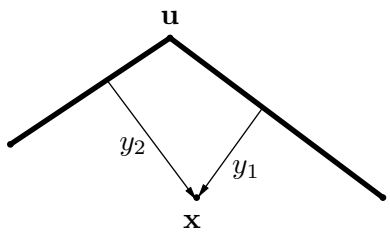
Thus, for task 2, we can move by increasing some x_i for which $c_i > 0$. How much can we increase it? *Until we hit some other constraint*. That is, we release the tight constraint $x_i \geq 0$ and increase x_i until some other inequality, previously loose, now becomes tight. At that point, we again have exactly n tight inequalities, so we are at a new vertex.

For instance, suppose we're dealing with the following linear program.

$$\begin{aligned} \max\ & 2x_1 + 5x_2 & \\ 2x_1 - x_2 &\leq 4 & \textcircled{1} \\ x_1 + 2x_2 &\leq 9 & \textcircled{2} \\ -x_1 + x_2 &\leq 3 & \textcircled{3} \\ x_1 &\geq 0 & \textcircled{4} \\ x_2 &\geq 0 & \textcircled{5} \end{aligned}$$

Simplex can be started at the origin, which is specified by constraints ④ and ⑤. To move, we release the tight constraint $x_2 \geq 0$. As x_2 is gradually increased, the first constraint it runs into is $-x_1 + x_2 \leq 3$, and thus it has to stop at $x_2 = 3$, at which point this new inequality is tight. The new vertex is thus given by ③ and ④.

So we know what to do if we are at the origin. But what if our current vertex **u** is elsewhere? The trick is to transform **u** into the origin, by shifting the coordinate system from the usual $(x_1, \ldots, x_n)$ to the "local view" from **u.** These local coordinates consist of (appropriately scaled) distances $y_1, \ldots, y_n$ to the n hyperplanes (inequalities) that define and enclose **u:**



Specifically, if one of these enclosing inequalities is $\mathbf{a}_i \cdot \mathrm{x} \leq b_i$, then the distance from a point **x** to that particular "wall" is

$$y_i = b_i - \mathbf{a}_i \cdot \mathrm{x}.$$

The n equations of this type, one per wall, define the y_i's as linear functions of the x_i's, and this relationship can be inverted to express the x_i's as a linear function of the y_i's. Thus we can rewrite the entire LP in terms of the y's. This doesn't fundamentally change it (for instance, the optimal value stays the same), but expresses it in a different coordinate frame. The revised "local" LP has the following three properties:

1. It includes the inequalities $\mathbf{y} \geq 0$, which are simply the transformed versions of the inequalities defining $\mathbf{u}$.
2. $\mathbf{u}$ itself is the origin in y-space.
3. The cost function becomes $\max c_{\mathbf{u}} + \tilde{\mathbf{c}}^T\mathbf{y}$, where $c_{\mathbf{u}}$ is the value of the objective function at $\mathbf{u}$ and $\tilde{\mathbf{c}}$ is a transformed cost vector.

In short, we are back to the situation we know how to handle! Figure 7.13 shows this algorithm in action, continuing with our earlier example.

The simplex algorithm is now fully defined. It moves from vertex to neighboring vertex, stopping when the objective function is locally optimal, that is, when the coordinates of the local cost vector are all zero or negative. As we've just seen, a vertex with this property must also be globally optimal. On the other hand, if the current vertex is not locally optimal, then its local coordinate system includes some dimension along which the objective function can be improved, so we move along this direction—along this edge of the polyhedron—until we reach a neighboring vertex. By the nondegeneracy assumption (see footnote 3 in Section 7.6.1), this edge has nonzero length, and so we strictly improve the objective value. Thus the process must eventually halt.

7.6.3 Loose ends

There are several important issues in the simplex algorithm that we haven't yet mentioned.

The starting vertex:
How do we find a vertex at which to start simplex? In our 2D and 3D examples we always started at the origin, which worked because the linear programs happened to have inequalities with positive right-hand sides. In a general LP we won't always be so fortunate. However, it turns out that finding a starting vertex *can be reduced to an LP* and solved by simplex!

To see how this is done, start with any linear program in standard form (recall Section 7.1.4), since we know LPs can always be rewritten this way.

$$\min\ \mathbf{c}^T\mathbf{x}\ \text{ such that }\ \mathbf{A}\mathbf{x} = \mathbf{b} \text{ and } \mathbf{x} \geq 0.$$

We first make sure that the right-hand sides of the equations are all nonnegative: if $b_i < 0$, just multiply both sides of the ith equation by -1.

Then we create a new LP as follows:

- Create m new *artificial variables* $z_1, \ldots, z_m \geq 0$, where m is the number of equations.
- Add z_i to the left-hand side of the ith equation.
- Let the objective, to be *minimized*, be $z_1 + z_2 + \cdots + z_m$.

For this new LP, it's easy to come up with a starting vertex, namely, the one with $z_i = b_i$ for all i and all other variables zero. Therefore we can solve it by simplex, to obtain the optimum solution.

Figure 7.13 Simplex in action.

Initial LP:

$$\begin{aligned} \max\ & 2x_1 + 5x_2 \\ 2x_1 - x_2 &\le 4 && ① \\ x_1 + 2x_2 &\le 9 && ② \\ -x_1 + x_2 &\le 3 && ③ \\ x_1 &\ge 0 && ④ \\ x_2 &\ge 0 && ⑤ \end{aligned}$$

Current vertex: {④, ⑤} (origin).
Objective value: 0.

Move: increase x_2.
⑤ is released, ③ becomes tight. Stop at $x_2 = 3$.

New vertex {④, ③} has local coordinates (y_1, y_2):

$$y_1 = x_1, \quad y_2 = 3 + x_1 - x_2$$

Rewritten LP:

$$\begin{aligned} \max\ & 15 + 7y_1 - 5y_2 \\ y_1 + y_2 &\le 7 && ① \\ 3y_1 - 2y_2 &\le 3 && ② \\ y_2 &\ge 0 && ③ \\ y_1 &\ge 0 && ④ \\ -y_1 + y_2 &\le 3 && ⑤ \end{aligned}$$

Current vertex: {④, ③}.
Objective value: 15.

Move: increase y_1.
④ is released, ② becomes tight. Stop at $y_1 = 1$.

New vertex {②, ③} has local coordinates (z_1, z_2):

$$z_1 = 3 - 3y_1 + 2y_2, \quad z_2 = y_2$$

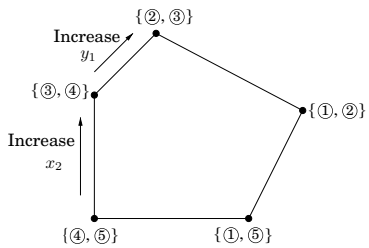
Rewritten LP:

$$\begin{aligned} \max\ & 22 - \tfrac{7}{3}z_1 - \tfrac{1}{3}z_2 \\ -\tfrac{1}{3}z_1 + \tfrac{5}{3}z_2 &\le 6 && ① \\ z_1 &\ge 0 && ② \\ z_2 &\ge 0 && ③ \\ \tfrac{1}{3}z_1 - \tfrac{2}{3}z_2 &\le 1 && ④ \\ \tfrac{1}{3}z_1 + \tfrac{1}{3}z_2 &\le 4 && ⑤ \end{aligned}$$

Current vertex: {②, ③}.
Objective value: 22.

Optimal: all $c_i < 0$.

Solve ②, ③ (in original LP) to get optimal solution $(x_1, x_2) = (1, 4)$.



There are two cases. If the optimum value of $z_1 + \cdots + z_m$ is zero, then all z_i's obtained by simplex are zero, and hence from the optimum vertex of the new LP we get a starting feasible vertex of the original LP, just by ignoring the z_i's. We can at last start simplex!

But what if the optimum objective turns out to be positive? Let us think. We tried to minimize the sum of the z_i's, but simplex decided that it cannot be zero. But this means that the original linear program is infeasible: it *needs* some nonzero z_i's to become feasible. This is how simplex discovers and reports that an LP is infeasible.

Degeneracy:
In the polyhedron of Figure 7.12 vertex B is *degenerate*. Geometrically, this means that it is the intersection of more than $n = 3$ faces of the polyhedron (in this case, ②, ③, ④, ⑤). Algebraically, it means that if we choose any one of four sets of three inequalities ({②, ③, ④}, {②, ③, ⑤}, {②, ④, ⑤}, and {③, ④, ⑤}) and solve the corresponding system of three linear equations in three unknowns, we'll get the same solution in all four cases: $(0, 300, 100)$. This is a serious problem: simplex may return a suboptimal degenerate vertex simply because all its neighbors are identical to it and thus have no better objective. And if we modify simplex so that it detects degeneracy and continues to hop from vertex to vertex despite lack of any improvement in the cost, it may end up looping forever.

One way to fix this is by a *perturbation*: change each b_i by a tiny random amount to $b_i \pm \epsilon_i$. This doesn't change the essence of the LP since the ϵ_i's are tiny, but it has the effect of differentiating between the solutions of the linear systems. To see why geometrically, imagine that the four planes ②, ③, ④, ⑤ were jolted a little. Wouldn't vertex B split into two vertices, very close to one another?

Unboundedness:
In some cases an LP is unbounded, in that its objective function can be made arbitrarily large (or small, if it's a minimization problem). If this is the case, simplex will discover it: in exploring the neighborhood of a vertex, it will notice that taking out an inequality and adding another leads to an underdetermined system of equations that has an infinity of solutions. And in fact (this is an easy test) the space of solutions contains a whole line across which the objective can become larger and larger, all the way to ∞. In this case simplex halts and complains.

7.6.4 The running time of simplex

What is the running time of simplex, for a generic linear program

$$\max \ \mathbf{c}^T\mathbf{x} \ \text{ such that } \ \mathbf{A}\mathbf{x} \leq 0 \text{ and } \mathbf{x} \geq 0,$$

where there are n variables and $\mathbf{A}$ contains m inequality constraints? Since it is an iterative algorithm that proceeds from vertex to vertex, let's start by computing the time taken for a single iteration. Suppose the current vertex is $\mathbf{u}$. By definition, it is the unique point at which n inequality constraints are satisfied with equality. Each of its neighbors shares $n - 1$ of these inequalities, so $\mathbf{u}$ can have at most $n \cdot m$ neighbors: choose which inequality to drop and which new one to add.

A naive way to perform an iteration would be to check each potential neighbor to see whether it really is a vertex of the polyhedron and to determine its cost.

Gaussian elimination

Under our algebraic definition, merely writing down the coordinates of a vertex involves solving a system of linear equations. How is this done?

We are given a system of n linear equations in n unknowns, say $n = 4$ and

$$\begin{aligned} x_1 \quad &\quad - 2x_3 \quad &\quad = 2 \\ x_2 + x_3 \quad &\quad = 3 \\ x_1 + x_2 \quad &\quad - x_4 = 4 \\ x_2 + 3x_3 + x_4 &= 5 \end{aligned}$$

The high school method for solving such systems is to repeatedly apply the following rule: *if we add a multiple of one equation to another equation, the overall system of equations remains equivalent.* For example, adding -1 times the first equation to the third one, we get the equivalent system

$$\begin{aligned} x_1 \quad &\quad - 2x_3 \quad &\quad = 2 \\ x_2 + x_3 \quad &\quad = 3 \\ x_2 + 2x_3 - x_4 &= 2 \\ x_2 + 3x_3 + x_4 &= 5 \end{aligned}$$

This transformation is clever in the following sense: it *eliminates* the variable x_1 from the third equation, leaving just one equation with x_1. In other words, ignoring the first equation, we have a system of *three* equations in *three* unknowns: we decreased n by 1! We can solve this smaller system to get x_2, x_3, x_4, and then plug these into the first equation to get x_1.

This suggests an algorithm—once more due to Gauss.

```
procedure gauss (E, X)
Input:  A system E = {e_1,...,e_n} of equations in n unknowns X = {x_1,...,x_n}:
        e_1 : a_11 x_1 + a_12 x_2 + ··· + a_1n x_n = b_1; ···; e_n : a_n1 x_1 + a_n2 x_2 + ··· + a_nn x_n = b_n
Output: A solution of the system, if one exists


If all coefficients a_i1 are zero:
   halt with message "either infeasible or not linearly independent"
if n = 1: return b_1/a_11


choose the coefficient a_p1 of largest magnitude, and swap equations e_1, e_p
for i = 2 to n:
    e_i = e_i − (a_i1/a_11) · e_i
(x_2,...,x_n) = gauss(E − {e_1}, X − {x_1})
x_1 = (b_1 − Σ_{j>1} a_1j x_j)/a_11
return (x_1,...,x_n)
```

(When choosing the equation to swap into first place, we pick the one with largest $|a_{p1}|$ for reasons of *numerical accuracy*; after all, we will be dividing by a_{p1}.)

Gaussian elimination uses $O(n^2)$ *arithmetic operations* to reduce the problem size from n to $n-1$, and thus uses $O(n^3)$ operations overall. To show that this is also a good estimate of the total *running time*, we need to argue that the numbers involved remain polynomially bounded—for instance, that the solution $(x_1, \ldots, x_n)$ does not require too much more precision to write down than the original coefficients a_{ij} and b_i. Do you see why this is true?

Finding the cost is quick, just a dot product, but checking whether it is a true vertex involves solving a system of n equations in n unknowns (that is, satisfying the n chosen inequalities exactly) and checking whether the result is feasible. By Gaussian elimination (see the following box) this takes $O(n^3)$ time, giving an unappetizing running time of $O(mn^4)$ per iteration.

Fortunately, there is a much better way, and this mn^4 factor can be improved to mn, making simplex a practical algorithm. Recall our earlier discussion (Section 7.6.2) about the *local view* from vertex **u.** It turns out that the per-iteration overhead of rewriting the LP in terms of the current local coordinates is just $O((m+n)n)$; this exploits the fact that the local view changes only slightly between iterations, in just one of its defining inequalities.

Next, to select the best neighbor, we recall that the (local view of) the objective function is of the form "max $c_{\mathbf{u}} + \tilde{\mathbf{c}} \cdot \mathbf{y}$" where $c_{\mathbf{u}}$ is the value of the objective function at **u.** This immediately identifies a promising direction to move: we pick any $\tilde{c}_i > 0$ (if there is none, then the current vertex is optimal and simplex halts). Since the rest of the LP has now been rewritten in terms of the **y**-coordinates, it is easy to determine how much y_i can be increased before some other inequality is violated. (And if we can increase y_i indefinitely, we know the LP is unbounded.)

It follows that the running time per iteration of simplex is just $O(mn)$. But how many iterations could there be? Naturally, there can't be more than $\binom{m+n}{n}$, which is an upper bound on the number of vertices. But this upper bound is exponential in n. And in fact, there are examples of LPs for which simplex does indeed take an exponential number of iterations. In other words, *simplex is an exponential-time algorithm*. However, such exponential examples do not occur in practice, and it is this fact that makes simplex so valuable and so widely used.

Linear programming in polynomial time

Simplex is not a polynomial time algorithm. Certain rare kinds of linear programs cause it to go from one corner of the feasible region to a better corner and then to a still better one, and so on for an exponential number of steps. For a long time, linear programming was considered a paradox, a problem that can be solved in practice, but not in theory!

Then, in 1979, a young Soviet mathematician called Leonid Khachiyan came up with the *ellipsoid algorithm*, one that is very different from simplex, extremely simple in its conception (but sophisticated in its proof) and yet one that solves any linear program in polynomial time. Instead of chasing the solution from one corner of the polyhedron to the next, Khachiyan's algorithm confines it to smaller and smaller ellipsoids (skewed high-dimensional balls). When this algorithm was announced, it became a kind of "mathematical Sputnik," a splashy achievement that had the U.S. establishment worried, in the height of the Cold War, about the possible scientific superiority of the Soviet Union. The ellipsoid algorithm turned out to be an important theoretical advance, but did not compete well with simplex in practice. The paradox of linear programming deepened: A problem with two algorithms, one that is efficient in theory, and one that is efficient in practice!

Linear programming in polynomial time (*Continued*)

A few years later Narendra Karmarkar, a graduate student at UC Berkeley, came up with a completely different idea, which led to another provably polynomial algorithm for linear programming. Karmarkar's algorithm is known as *the interior point method*, because it does just that: it dashes to the optimum corner not by hopping from corner to corner on the surface of the polyhedron like simplex does, but by cutting a clever path in the interior of the polyhedron. And it does perform well in practice.

But perhaps the greatest advance in linear programming algorithms was not Khachiyan's theoretical breakthrough or Karmarkar's novel approach, but an unexpected consequence of the latter: the fierce competition between the two approaches, simplex and interior point, resulted in the development of very fast code for linear programming.

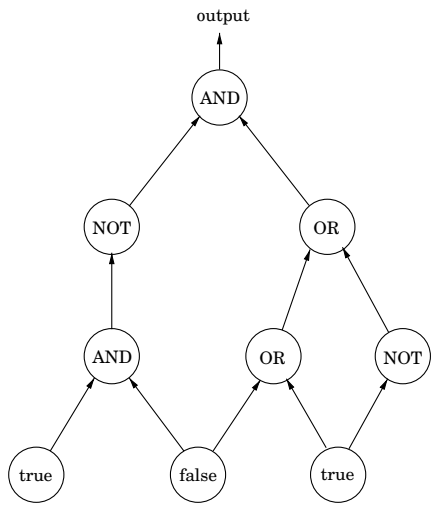
7.7 Postscript: circuit evaluation

The importance of linear programming stems from the astounding variety of problems that reduce to it and thereby bear witness to its expressive power. In a sense, this next one is the *ultimate* application.

We are given a *Boolean circuit*, that is, a dag of gates of the following types.

- *Input gates* have indegree zero, with value `true` or `false`.
- AND gates and OR gates have indegree 2.
- NOT gates have indegree 1.

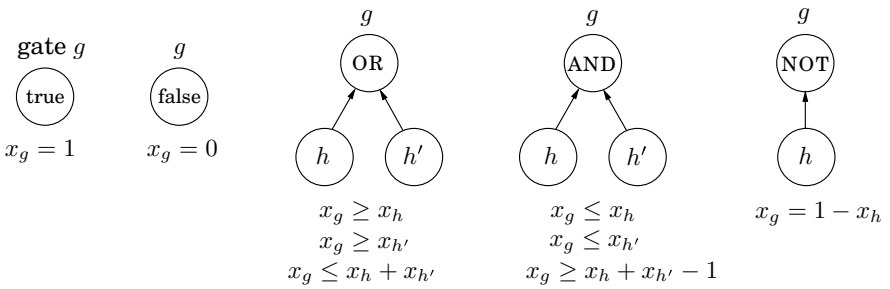
In addition, one of the gates is designated as the *output*. Here's an example.



The CIRCUIT VALUE problem is the following: when the laws of Boolean logic are applied to the gates in topological order, does the output evaluate to `true`?

There is a simple, automatic way of translating this problem into a linear program. Create a variable x_g for each gate g, with constraints $0 \leq x_g \leq 1$. Add additional

constraints for each type of gate:



These constraints force all the gates to take on exactly the right values—0 for `false`, and 1 for `true`. We don't need to maximize or minimize anything, and we can read the answer off from the variable x_o corresponding to the output gate.

This is a straightforward reduction to linear programming, from a problem that may not seem very interesting at first. However, the CIRCUIT VALUE problem is in a sense *the most general problem solvable in polynomial time*! After all, any algorithm will eventually run on a computer, and the computer is ultimately a Boolean combinational circuit implemented on a chip. If the algorithm runs in polynomial time, it can be rendered as a Boolean circuit consisting of polynomially many copies of the computer's circuit, one per unit of time, with the values of the gates in one layer used to compute the values for the next. Hence, the fact that CIRCUIT VALUE reduces to linear programming means that *all problems that can be solved in polynomial time do!*

In our next topic, **NP**-*completeness*, we shall see that many *hard* problems reduce, much the same way, to *integer programming*, linear programming's difficult twin.

Another parting thought: by what other means can the circuit evaluation problem be solved? Let's think—a circuit is a dag. And what algorithmic technique is most appropriate for solving problems on dags? That's right: dynamic programming! Together with linear programming, the world's two most general algorithmic techniques.

Exercises

7.1. Consider the following linear program.

$$\begin{aligned} \text{maximize} \quad & 5x + 3y \\ 5x - 2y &\geq 0 \\ x + y &\leq 7 \\ x &\leq 5 \\ x &\geq 0 \\ y &\geq 0 \end{aligned}$$

Plot the feasible region and identify the optimal solution.

7.2. Duckwheat is produced in Kansas and Mexico and consumed in New York and California. Kansas produces 15 shnupells of duckwheat and Mexico 8. Meanwhile, New York consumes 10 shnupells and California 13. The transportation costs per shnupell are $4 from Mexico to New York, $1 from Mexico to California, $2 from Kansas to New York, and $3 from Kansas to California.

Write a linear program that decides the amounts of duckwheat (in shnupells and fractions of a shnupell) to be transported from each producer to each consumer, so as to minimize the overall transportation cost.

7.3. A cargo plane can carry a maximum weight of 100 tons and a maximum volume of 60 cubic meters. There are three materials to be transported, and the cargo company may choose to carry any amount of each, up to the maximum available limits given below.

- Material 1 has density 2 tons/cubic meter, maximum available amount 40 cubic meters, and revenue $1,000 per cubic meter.
- Material 2 has density 1 ton/cubic meter, maximum available amount 30 cubic meters, and revenue $1,200 per cubic meter.
- Material 3 has density 3 tons/cubic meter, maximum available amount 20 cubic meters, and revenue $12,000 per cubic meter.

Write a linear program that optimizes revenue within the constraints.

7.4. Moe is deciding how much Regular Duff beer and how much Duff Strong beer to order each week. Regular Duff costs Moe $1 per pint and he sells it at $2 per pint; Duff Strong costs Moe $1.50 per pint and he sells it at $3 per pint. However, as part of a complicated marketing scam, the Duff company will only sell a pint of Duff Strong for each two pints or more of Regular Duff that Moe buys. Furthermore, due to past events that are better left untold, Duff will not sell Moe more than 3,000 pints per week. Moe knows that he can sell however much beer he has. Formulate a linear program for deciding how much Regular Duff and how much Duff Strong to buy, so as to maximize Moe's profit. Solve the program geometrically.

7.5. The Canine Products company offers two dog foods, Frisky Pup and Husky Hound, that are made from a blend of cereal and meat. A package of Frisky Pup requires 1 pound of cereal and 1.5 pounds of meat, and sells for $7. A package of Husky Hound uses 2 pounds of cereal and 1 pound of meat, and sells for $6. Raw cereal costs $1 per pound and raw meat costs $2 per pound. It also costs $1.40 to package the Frisky Pup and $0.60 to package the Husky Hound. A total of 240,000 pounds of cereal and 180,000 pounds of meat are available each month. The only production bottleneck is that the factory can only package 110,000 bags of Frisky Pup per month. Needless to say, management would like to maximize profit.

(a) Formulate the problem as a linear program in two variables.

(b) Graph the feasible region, give the coordinates of every vertex, and circle the vertex maximizing profit. What is the maximum profit possible?

7.6. Give an example of a linear program in two variables whose feasible region is infinite, but such that there is an optimum solution of bounded cost.

7.7. Find necessary and sufficient conditions on the reals a and b under which the linear program

$$\begin{aligned} \max\ & x + y \\ ax + by &\leq 1 \\ x, y &\geq 0 \end{aligned}$$

(a) Is infeasible.

(b) Is unbounded.

(c) Has a finite and unique optimal solution.

7.8. You are given the following points in the plane:

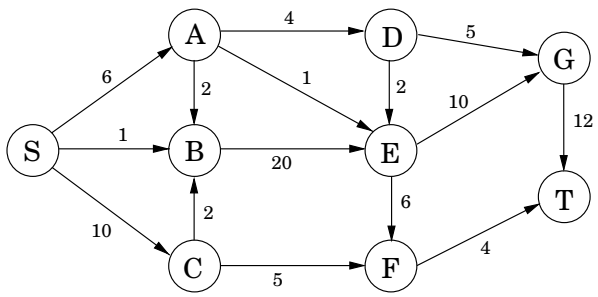
$$(1, 3),\ (2, 5),\ (3, 7),\ (5, 11),\ (7, 14),\ (8, 15),\ (10, 19).$$

You want to find a line $ax + by = c$ that approximately passes through these points (no line is a perfect fit). Write a linear program (you don't need to solve it) to find the line that minimizes the *maximum absolute error*,

$$\max_{1 \leq i \leq 7} |ax_i + by_i - c|.$$

7.9. A *quadratic programming* problem seeks to maximize a quadratic objective function (with terms like $3x_1^2$ or $5x_1x_2$) subject to a set of linear constraints. Give an example of a quadratic program in two variables x_1, x_2 such that the feasible region is nonempty and bounded, and yet none of the vertices of this region optimize the (quadratic) objective.

7.10. For the following network, with edge capacities as shown, find the maximum flow from S to T, along with a matching cut.



7.11. Write the dual to the following linear program.

$$\begin{aligned} \max\ & x + y \\ 2x + y &\leq 3 \\ x + 3y &\leq 5 \\ x, y &\geq 0 \end{aligned}$$

Find the optimal solutions to both primal and dual LPs.

7.12. For the linear program

$$\begin{aligned} \max\ & x_1 - 2x_3 \\ & x_1 - x_2 \le 1 \\ & 2x_2 - x_3 \le 1 \\ & x_1, x_2, x_3 \ge 0 \end{aligned}$$

prove that the solution $(x_1, x_2, x_3) = (3/2, 1/2, 0)$ is optimal.

7.13. *Matching pennies.* In this simple two-player game, the players (call them R and C) each choose an outcome, *heads* or *tails*. If both outcomes are equal, C gives a dollar to R; if the outcomes are different, R gives a dollar to C.

(a) Represent the payoffs by a 2×2 matrix.

(b) What is the value of this game, and what are the optimal strategies for the two players?

7.14. The pizza business in Little Town is split between two rivals, Tony and Joey. They are each investigating strategies to steal business away from the other. Joey is considering either lowering prices or cutting bigger slices. Tony is looking into starting up a line of gourmet pizzas, or offering outdoor seating, or giving free sodas at lunchtime. The effects of these various strategies are summarized in the following payoff matrix (entries are dozens of pizzas, Joey's gain and Tony's loss).

		Tony		
		Gourmet	Seating	Free soda
Joey	Lower price	+2	0	−3
	Bigger slices	−1	−2	+1

For instance, if Joey reduces prices and Tony goes with the gourmet option, then Tony will lose 2 dozen pizzas worth of business to Joey.

What is the value of this game, and what are the optimal strategies for Tony and Joey?

7.15. Find the value of the game specified by the following payoff matrix.

$$\begin{array}{rrrr} 0 & 0 & -1 & -1 \\ 0 & 1 & -2 & -1 \\ -1 & -1 & 1 & 1 \\ -1 & 0 & 0 & 1 \\ 1 & -2 & 0 & -3 \\ 0 & -3 & 2 & -1 \\ 0 & -2 & 1 & -1 \end{array}$$

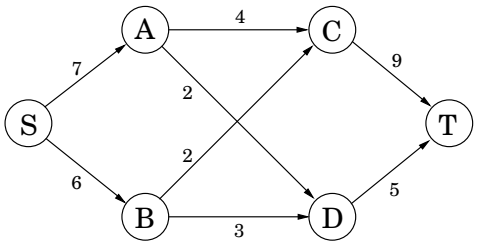
(*Hint:* Consider the mixed strategies $(1/3, 0, 0, 1/2, 1/6, 0, 0)$ and $(2/3, 0, 0, 1/3)$.)

7.16. A salad is any combination of the following ingredients: (1) tomato, (2) lettuce, (3) spinach, (4) carrot, and (5) oil. Each salad must contain: (A) at least 15 grams of protein, (B) at least 2 and at most 6 grams of fat, (C) at least 4 grams of carbohydrates, (D) at most 100 milligrams of sodium. Furthermore, (E) you do not want your salad to be more than 50% greens by mass. The nutritional contents of these ingredients (per 100 grams) are

ingredient	energy (kcal)	protein (grams)	fat (grams)	carbohydrate (grams)	sodium (milligrams)
tomato	21	0.85	0.33	4.64	9.00
lettuce	16	1.62	0.20	2.37	8.00
spinach	371	12.78	1.58	74.69	7.00
carrot	346	8.39	1.39	80.70	508.20
oil	884	0.00	100.00	0.00	0.00

Find a linear programming applet on the Web and use it to make the salad with the fewest calories under the nutritional constraints. Describe your linear programming formulation and the optimal solution (the quantity of each ingredient and the value). Cite the Web resources that you used.

7.17. Consider the following network (the numbers are edge capacities).



(a) Find the maximum flow f and a minimum cut.

(b) Draw the residual graph G_f (along with its edge capacities). In this residual network, mark the vertices reachable from S and the vertices from which T is reachable.

(c) An edge of a network is called a *bottleneck edge* if increasing its capacity results in an increase in the maximum flow. List all bottleneck edges in the above network.

(d) Give a very simple example (containing at most four nodes) of a network which has no bottleneck edges.

(e) Give an efficient algorithm to identify all bottleneck edges in a network. (*Hint:* Start by running the usual network flow algorithm, and then examine the residual graph.)

7.18. There are many common variations of the maximum flow problem. Here are four of them.

(a) There are many sources and many sinks, and we wish to maximize the total flow from all sources to all sinks.

(b) Each *vertex* also has a capacity on the maximum flow that can enter it.

(c) Each edge has not only a capacity, but also a *lower bound* on the flow it must carry.

(d) The outgoing flow from each node u is not the same as the incoming flow, but is smaller by a factor of $(1 - \epsilon_u)$, where ϵ_u is a loss coefficient associated with node u.

Each of these can be solved efficiently. Show this by reducing (a) and (b) to the original max-flow problem, and reducing (c) and (d) to linear programming.

7.19. Suppose someone presents you with a solution to a max-flow problem on some network. Give a *linear* time algorithm to determine whether the solution does indeed give a maximum flow.

7.20. Consider the following generalization of the maximum flow problem.

You are given a directed network $G = (V, E)$ with edge capacities $\{c_e\}$. Instead of a single (s, t) pair, you are given multiple pairs $(s_1, t_1), (s_2, t_2), \ldots, (s_k, t_k)$, where the s_i are sources of G and the t_i are sinks of G. You are also given k *demands* $d_1, \ldots, d_k$. The goal is to find k flows $f^{(1)}, \ldots, f^{(k)}$ with the following properties:

- $f^{(i)}$ is a valid flow from s_i to t_i.
- For each edge e, the total flow $f_e^{(1)} + f_e^{(2)} + \cdots + f_e^{(k)}$ does not exceed the capacity c_e.
- The size of each flow $f^{(i)}$ is at least the demand d_i.
- The size of the *total* flow (the sum of the flows) is as large as possible.

How would you solve this problem?

7.21. An edge of a flow network is called *critical* if decreasing the capacity of this edge results in a decrease in the maximum flow. Give an efficient algorithm that finds a critical edge in a network.

7.22. In a particular network $G = (V, E)$ whose edges have integer capacities c_e, we have already found the maximum flow f from node s to node t. However, we now find out that one of the capacity values we used was wrong: for edge (u, v) we used c_{uv} whereas it should have been $c_{uv} - 1$. This is unfortunate because the flow f uses that particular edge at full capacity: $f_{uv} = c_{uv}$.

We could redo the flow computation from scratch, but there's a faster way. Show how a new optimal flow can be computed in $O(|V| + |E|)$ time.

7.23. A *vertex cover* of an undirected graph $G = (V, E)$ is a subset of the vertices which touches every edge—that is, a subset $S \subset V$ such that for each edge $\{u, v\} \in E$, one or both of u, v are in S.

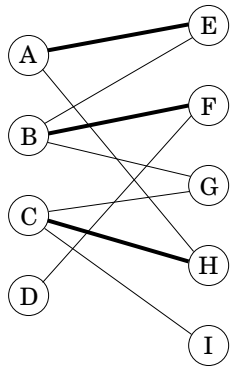
Show that the problem of finding the minimum vertex cover in a *bipartite* graph reduces to maximum flow. (*Hint:* Can you relate this problem to the minimum cut in an appropriate network?)

7.24. *Direct bipartite matching.* We've seen how to find a maximum matching in a bipartite graph via reduction to the maximum flow problem. We now develop a direct algorithm.

Let $G = (V_1 \cup V_2, E)$ be a bipartite graph (so each edge has one endpoint in V_1 and one endpoint in V_2), and let $M \in E$ be a matching in the graph (that is, a set of edges that don't touch). A vertex is said to be *covered* by M if it is the

endpoint of one of the edges in M. An *alternating path* is a path of odd length that starts and ends with a non-covered vertex, and whose edges alternate between M and $E - M$.

(a) In the bipartite graph below, a matching M is shown in bold. Find an alternating path.

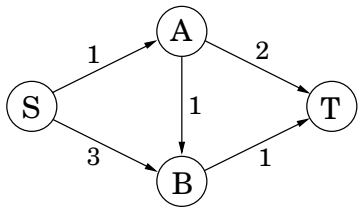


(b) Prove that a matching M is maximum if and only if there does not exist an alternating path with respect to it.

(c) Design an algorithm that finds an alternating path in $O(|V| + |E|)$ time using a variant of breadth-first search.

(d) Give a direct $O(|V| \cdot |E|)$ algorithm for finding a maximum matching in a bipartite graph.

7.25. *The dual of maximum flow.* Consider the following network with edge capacities.



(a) Write the problem of finding the maximum flow from S to T as a linear program.

(b) Write down the dual of this linear program. There should be a dual variable for each edge of the network and for each vertex other than S, T.

Now we'll solve the same problem in full generality. Recall the linear program for a general maximum flow problem (Section 7.2).

(c) Write down the dual of this general flow LP, using a variable y_e for each edge and x_u for each vertex $u \neq s, t$.

(d) Show that any solution to the general dual LP must satisfy the following property: for any directed path from s to t in the network, the sum of the y_e values along the path must be at least 1.

(e) What are the intuitive meanings of the dual variables? Show that any $s-t$ cut in the network can be translated into a dual feasible solution whose cost is exactly the capacity of that cut.

7.26. In a satisfiable system of linear inequalities

$$\begin{aligned} a_{11}x_1 + \cdots + a_{1n}x_n &\le b_1 \\ &\vdots \\ a_{m1}x_1 + \cdots + a_{mn}x_n &\le b_m \end{aligned}$$

we describe the jth inequality as *forced-equal* if it is satisfied with equality by *every* solution $\mathrm{x} = (x_1, \ldots, x_n)$ of the system. Equivalently, $\sum_i a_{ji}x_i \le b_j$ is *not* forced-equal if there exists an x that satisfies the whole system and such that $\sum_i a_{ji}x_i < b_j$.
For example, in

$$\begin{aligned} x_1 + x_2 &\le 2 \\ -x_1 - x_2 &\le -2 \\ x_1 &\le 1 \\ -x_2 &\le 0 \end{aligned}$$

the first two inequalities are forced-equal, while the third and fourth are not. A solution x to the system is called *characteristic* if, for every inequality I that is not forced-equal, x satisfies I without equality. In the instance above, such a solution is $(x_1, x_2) = (-1, 3)$, for which $x_1 < 1$ and $-x_2 < 0$ while $x_1 + x_2 = 2$ and $-x_1 - x_2 = -2$.

(a) Show that any satisfiable system has a characteristic solution.

(b) Given a satisfiable system of linear inequalities, show how to use linear programming to determine which inequalities are forced-equal, and to find a characteristic solution.

7.27. Show that the *change-making problem* (Exercise 6.17) can be formulated as an integer linear program. Can we solve this program as an LP, in the certainty that the solution will turn out to be integral (as in the case of bipartite matching)? Either prove it or give a counterexample.

7.28. *A linear program for shortest path.* Suppose we want to compute the shortest path from node s to node t in a directed graph with edge lengths $l_e > 0$.

(a) Show that this is equivalent to finding an $s-t$ flow f that minimizes $\sum_e l_e f_e$ subject to $\mathrm{size}(f) = 1$. There are no capacity constraints.

(b) Write the shortest path problem as a linear program.

(c) Show that the dual LP can be written as

$$\begin{aligned} &\max\ x_s - x_t \\ x_u - x_v &\le l_{uv} \text{ for all } (u, v) \in E \end{aligned}$$

(d) An interpretation for the dual is given in the box on page 209. Why isn't our dual LP identical to the one on that page?

7.29. *Hollywood.* A film producer is seeking actors and investors for his new movie. There are n available actors; actor i charges s_i dollars. For funding, there are m available investors. Investor j will provide p_j dollars, but only on the condition that certain actors $L_j \subseteq \{1, 2, \ldots, n\}$ are included in the cast (*all* of these actors L_j must be chosen in order to receive funding from investor j).

The producer's profit is the sum of the payments from investors minus the payments to actors. The goal is to maximize this profit.

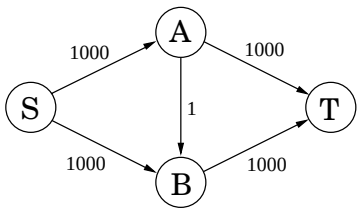
(a) Express this problem as an integer linear program in which the variables take on values $\{0, 1\}$.

(b) Now relax this to a linear program, and show that there must in fact be an *integral* optimal solution (as is the case, for example, with maximum flow and bipartite matching).

7.30. *Hall's theorem.* Returning to the matchmaking scenario of Section 7.3, suppose we have a bipartite graph with boys on the left and an equal number of girls on the right. Hall's theorem says that there is a perfect matching if and only if the following condition holds: any subset S of boys is connected to at least $|S|$ girls.

Prove this theorem. (*Hint:* The max-flow min-cut theorem should be helpful.)

7.31. Consider the following simple network with edge capacities as shown.



(a) Show that, if the Ford-Fulkerson algorithm is run on this graph, a careless choice of updates might cause it to take 1000 iterations. Imagine if the capacities were a million instead of 1000!

We will now find a strategy for choosing paths under which the algorithm is guaranteed to terminate in a reasonable number of iterations.

Consider an arbitrary directed network $(G = (V, E), s, t, \{c_e\})$ in which we want to find the maximum flow. Assume for simplicity that all edge capacities are at least 1, and define the capacity of an $s - t$ path to be the smallest capacity of its constituent edges. The *fattest path* from s to t is the path with the most capacity.

(b) Show that the fattest $s - t$ path in a graph can be computed by a variant of Dijkstra's algorithm.

(c) Show that the maximum flow in G is the sum of individual flows along at most $|E|$ paths from s to t.

(d) Now show that if we always increase flow along the fattest path in the residual graph, then the Ford-Fulkerson algorithm will terminate in at

most $O(|E| \log F)$ iterations, where F is the size of the maximum flow. (*Hint:* It might help to recall the proof for the greedy set cover algorithm in Section 5.4.)

In fact, an even simpler rule—finding a path in the residual graph using breadth-first search—guarantees that at most $O(|V| \cdot |E|)$ iterations will be needed.

Chapter 8

NP-complete problems

8.1 Search problems

Over the past seven chapters we have developed algorithms for finding shortest paths and minimum spanning trees in graphs, matchings in bipartite graphs, maximum increasing subsequences, maximum flows in networks, and so on. All these algorithms are *efficient*, because in each case their time requirement grows as a polynomial function (such as n, n^2, or n^3) of the size of the input.

To better appreciate such efficient algorithms, consider the alternative: In all these problems we are searching for a solution (path, tree, matching, etc.) from among an *exponential* population of possibilities. Indeed, n boys can be matched with n girls in $n!$ different ways, a graph with n vertices has n^{n-2} spanning trees, and a typical graph has an exponential number of paths from s to t. All these problems could in principle be solved in exponential time by checking through all candidate solutions, one by one. But an algorithm whose running time is 2^n, or worse, is all but useless in practice (see the next box). The quest for efficient algorithms is about finding clever ways to bypass this process of exhaustive search, using clues from the input in order to dramatically narrow down the search space.

So far in this book we have seen the most brilliant successes of this quest, algorithmic techniques that defeat the specter of exponentiality: greedy algorithms, dynamic programming, linear programming (while divide-and-conquer typically yields faster algorithms for problems we can already solve in polynomial time). Now the time has come to meet the quest's most embarrassing and persistent failures. We shall see some other "search problems," in which again we are seeking a solution with particular properties among an exponential chaos of alternatives. But for these new problems no shortcut seems possible. The fastest algorithms we know for them are all exponential—not substantially better than an exhaustive search. We now introduce some important examples.

Satisfiability

SATISFIABILITY, or SAT (recall Exercise 3.28 and Section 5.3), is a problem of great practical importance, with applications ranging from chip testing and computer design

The story of Sissa and Moore

According to the legend, the game of chess was invented by the Brahmin Sissa to amuse and teach his king. Asked by the grateful monarch what he wanted in return, the wise man requested that the king place one grain of rice in the first square of the chessboard, two in the second, four in the third, and so on, doubling the amount of rice up to the 64th square. The king agreed on the spot, and as a result he was the first person to learn the valuable—albeit humbling—lesson of *exponential growth*. Sissa's request amounted to $2^{64} - 1 = 18{,}446{,}744{,}073{,}709{,}551{,}615$ grains of rice, enough rice to pave all of India several times over!

All over nature, from colonies of bacteria to cells in a fetus, we see systems that grow exponentially—for a while. In 1798, the British philosopher T. Robert Malthus published an essay in which he predicted that the exponential growth (he called it "geometric growth") of the human population would soon deplete linearly growing resources, an argument that influenced Charles Darwin deeply. Malthus knew the fundamental fact that an exponential sooner or later takes over any polynomial.

In 1965, computer chip pioneer Gordon E. Moore noticed that transistor density in chips had doubled every year in the early 1960s, and he predicted that this trend would continue. This prediction, moderated to a doubling every 18 months and extended to computer speed, is known as *Moore's law.* It has held remarkably well for 40 years. And these are the two root causes of the explosion of information technology in the past decades: *Moore's law and efficient algorithms.*

It would appear that Moore's law provides a disincentive for developing polynomial algorithms. After all, if an algorithm is exponential, why not wait it out until Moore's law makes it feasible? But in reality the exact opposite happens: Moore's law is a huge incentive for developing efficient algorithms, because such algorithms are needed in order to take advantage of the exponential increase in computer speed.

Here is why. If, for example, an $O(2^n)$ algorithm for Boolean satisfiability (SAT) were given an hour to run, it would have solved instances with 25 variables back in 1975, 31 variables on the faster computers available in 1985, 38 variables in 1995, and about 45 variables with today's machines. Quite a bit of progress—except that each extra variable requires a year and a half's wait, while the appetite of applications (many of which are, ironically, related to computer design) grows much faster. In contrast, the size of the instances solved by an $O(n)$ or $O(n \log n)$ algorithm would be *multiplied by a factor of about 100* each decade. In the case of an $O(n^2)$ algorithm, the instance size solvable in a fixed time would be multiplied by about 10 each decade. Even an $O(n^6)$ algorithm, polynomial yet unappetizing, would more than double the size of the instances solved each decade. When it comes to the growth of the size of problems we can attack with an algorithm, we have a reversal: exponential algorithms make polynomially slow progress, while polynomial algorithms advance exponentially fast! For Moore's law to be reflected in the world we *need* efficient algorithms.

The story of Sissa and Moore (*Continued*)

As Sissa and Malthus knew very well, exponential expansion cannot be sustained indefinitely in our finite world. Bacterial colonies run out of food; chips hit the atomic scale. Moore's law will stop doubling the speed of our computers within a decade or two. And then progress will depend on algorithmic ingenuity—or otherwise perhaps on novel ideas such as *quantum computation*, explored in Chapter 10.

to image analysis and software engineering. It is also a canonical hard problem. Here's what an instance of SAT looks like:

$$(x \vee y \vee z)\ (x \vee \overline{y})\ (y \vee \overline{z})\ (z \vee \overline{x})\ (\overline{x} \vee \overline{y} \vee \overline{z}).$$

This is a *Boolean formula in conjunctive normal form (CNF)*. It is a collection of *clauses* (the parentheses), each consisting of the disjunction (logical *or*, denoted $\vee$) of several *literals*, where a literal is either a Boolean variable (such as x) or the negation of one (such as $\overline{x}$). A *satisfying truth assignment* is an assignment of `false` or `true` to each variable so that every clause contains a literal whose value is `true`. The SAT problem is the following: given a Boolean formula in conjunctive normal form, either find a satisfying truth assignment or else report that none exists.

In the instance shown previously, setting all variables to `true`, for example, satisfies every clause except the last. Is there a truth assignment that satisfies *all* clauses?

With a little thought, it is not hard to argue that in this particular case no such truth assignment exists. (*Hint*: The three middle clauses constrain all three variables to have the same value.) But how do we decide this in general? Of course, we can always search through all truth assignments, one by one, but for formulas with n variables, the number of possible assignments is exponential, 2^n.

SAT is a typical *search problem*. We are given an *instance* I (that is, some input data specifying the problem at hand, in this case a Boolean formula in conjunctive normal form), and we are asked to find a *solution* S (an object that meets a particular specification, in this case an assignment that satisfies each clause). If no such solution exists, we must say so.

More specifically, a search problem must have the property that any proposed solution S to an instance I can be *quickly checked* for correctness. What does this entail? For one thing, S must at least be concise (quick to read), with length polynomially bounded by that of I. This is clearly true in the case of SAT, for which S is an assignment to the variables. To formalize the notion of quick checking, we will say that there is a polynomial-time algorithm that takes as input I and S and decides whether or not S is a solution of I. For SAT, this is easy as it just involves checking whether the assignment specified by S indeed satisfies every clause in I.

Later in this chapter it will be useful to shift our vantage point and to think of this efficient algorithm for checking proposed solutions as *defining* the search problem. Thus:

> *A* search problem *is specified by an algorithm C that takes two inputs, an instance I and a proposed solution S, and runs in time polynomial in $|I|$. We say S is a solution to I if and only if $C(I, S) =$* `true`.

Given the importance of the SAT search problem, researchers over the past 50 years have tried hard to find efficient ways to solve it, but without success. The fastest algorithms we have are still exponential on their worst-case inputs.

Yet, interestingly, there are two natural variants of SAT for which we do have good algorithms. If all clauses contain at most one positive literal, then the Boolean formula is called a *Horn formula*, and a satisfying truth assignment, if one exists, can be found by the greedy algorithm of Section 5.3. Alternatively, if all clauses have only *two* literals, then graph theory comes into play, and SAT can be solved in linear time by finding the strongly connected components of a particular graph constructed from the instance (recall Exercise 3.28). In fact, in Chapter 9 we'll see a different polynomial algorithm for this same special case, which is called 2SAT.

On the other hand, if we are just a little more permissive and allow clauses to contain *three* literals, then the resulting problem, known as 3SAT (an example of which we saw earlier), once again becomes hard to solve!

The traveling salesman problem

In the traveling salesman problem (TSP) we are given n vertices $1, \ldots, n$ and all $n(n-1)/2$ distances between them, as well as a *budget* b. We are asked to find a *tour*, a cycle that passes through every vertex exactly once, of total cost b or less—or to report that no such tour exists. That is, we seek a permutation $\tau(1), \ldots, \tau(n)$ of the vertices such that when they are toured in this order, the total distance covered is at most b:

$$d_{\tau(1),\tau(2)} + d_{\tau(2),\tau(3)} + \cdots + d_{\tau(n),\tau(1)} \leq b.$$

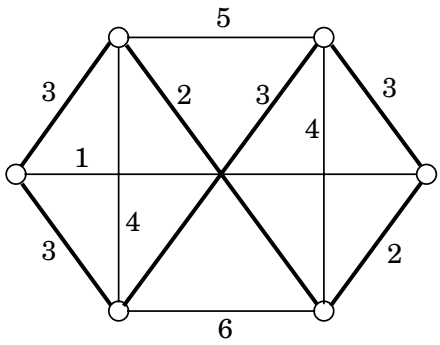
See Figure 8.1 for an example (only some of the distances are shown; assume the rest are very large).

Notice how we have defined the TSP as a *search problem*: given an instance, find a tour within the budget (or report that none exists). But why are we expressing the traveling salesman problem in this way, when in reality it is an *optimization problem*, in which the *shortest* possible tour is sought? Why dress it up as something else?

For a good reason. Our plan in this chapter is to compare and relate problems. The framework of search problems is helpful in this regard, because it encompasses optimization problems like the TSP in addition to true search problems like SAT.

Turning an optimization problem into a search problem does not change its difficulty at all, because the two versions *reduce to one another*. Any algorithm that solves the optimization TSP also readily solves the search problem: find the optimum tour and if it is within budget, return it; if not, there is no solution.

Figure 8.1 The optimal traveling salesman tour, shown in bold, has length 18.



Conversely, an algorithm for the search problem can also be used to solve the optimization problem. To see why, first suppose that we somehow knew the *cost* of the optimum tour; then we could find this tour by calling the algorithm for the search problem, using the optimum cost as the budget. Fine, but how do we find the optimum cost? Easy: By binary search! (See Exercise 8.1.)

Incidentally, there is a subtlety here: Why do we have to introduce a budget? Isn't any optimization problem also a search problem in the sense that we are searching for a solution that has the property of being optimal? The catch is that the solution to a search problem should be easy to recognize, or as we put it earlier, polynomial-time checkable. Given a potential solution to the TSP, it is easy to check the properties "is a tour" (just check that each vertex is visited exactly once) and "has total length $\leq b$." But how could one check the property "is optimal"?

As with SAT, there are no known polynomial-time algorithms for the TSP, despite much effort by researchers over nearly a century. Of course, there is an exponential algorithm for solving it, by trying all $(n-1)!$ tours, and in Section 6.6 we saw a faster, yet still exponential, dynamic programming algorithm.

The minimum spanning tree (MST) problem, for which we *do* have efficient algorithms, provides a stark contrast here. To phrase it as a search problem, we are again given a distance matrix and a bound b, and are asked to find a tree T with total weight $\sum_{(i,j)\in T} d_{ij} \leq b$. The TSP can be thought of as a tough cousin of the MST problem, in which the tree is not allowed to branch and is therefore a path.[1] This extra restriction on the structure of the tree results in a much harder problem.

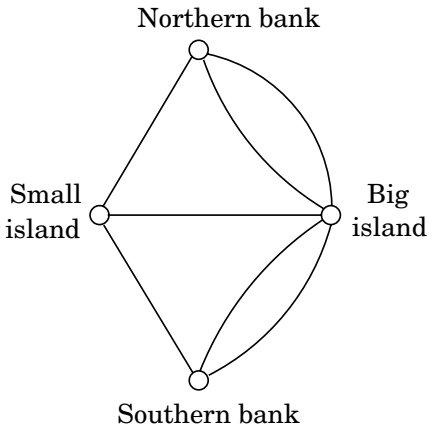
Euler and Rudrata

In the summer of 1735 Leonhard Euler (pronounced "Oiler"), the famous Swiss mathematician, was walking the bridges of the East Prussian town of Königsberg.

[1] Actually the TSP demands a cycle, but one can define an alternative version that seeks a path, and it is not hard to see that this is just as hard as the TSP itself.

After a while, he noticed in frustration that, no matter where he started his walk, no matter how cleverly he continued, it was impossible to cross each bridge exactly once. And from this silly ambition, the field of graph theory was born.

Euler identified at once the roots of the park's deficiency. First, you turn the map of the park into a graph whose vertices are the four land masses (two islands, two banks) and whose edges are the seven bridges:



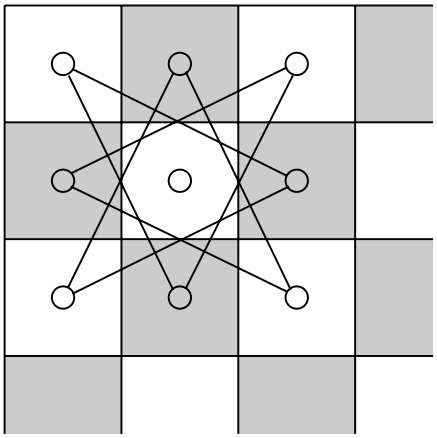
This graph has multiple edges between two vertices—a feature we have not been allowing so far in this book, but one that is meaningful for this particular problem, since each bridge must be accounted for separately. We are looking for a path that goes through each edge exactly once (the path is allowed to repeat vertices). In other words, we are asking this question: *When can a graph be drawn without lifting the pencil from the paper?*

The answer discovered by Euler is simple, elegant, and intuitive: *If and only if (a) the graph is connected and (b) every vertex, with the possible exception of two vertices (the start and final vertices of the walk), has even degree* (Exercise 3.26). This is why Königsberg's park was impossible to traverse: all four vertices have odd degree.

To put it in terms of our present concerns, let us define a search problem called EULER PATH: Given a graph, find a path that contains each edge exactly once. It follows from Euler's observation, and a little more thinking, that this search problem can be solved in polynomial time.

Almost a millennium before Euler's fateful summer in East Prussia, a Kashmiri poet named Rudrata had asked this question: Can one visit all the squares of the chessboard, without repeating any square, in one long walk that ends at the starting square and at each step makes a legal knight move? This is again a graph problem: the graph now has 64 vertices, and two squares are joined by an edge if a knight can go from one to the other in a single move (that is, if their coordinates differ by 2 in one dimension and by 1 in the other). See Figure 8.2 for the portion of the graph

Figure 8.2 Knight's moves on a corner of a chessboard.



corresponding to the upper left corner of the board. Can you find a knight's tour on your chessboard?

This is a different kind of search problem in graphs: we want a cycle that goes through all *vertices* (as opposed to all edges in Euler's problem), without repeating any vertex. And there is no reason to stick to chessboards; this question can be asked of any graph. Let us define the RUDRATA CYCLE search problem to be the following: given a graph, find a cycle that visits each vertex exactly once—or report that no such cycle exists.[2] This problem is ominously reminiscent of the TSP, and indeed no polynomial algorithm is known for it.

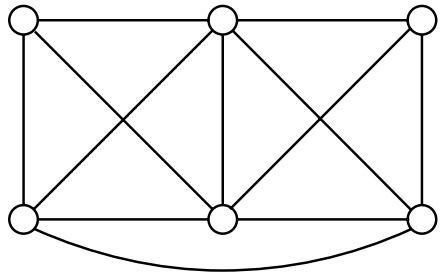
There are two differences between the definitions of the Euler and Rudrata problems. The first is that Euler's problem visits all *edges* while Rudrata's visits all *vertices*. But there is also the issue that one of them demands a path while the other requires a cycle. Which of these differences accounts for the huge disparity in computational complexity between the two problems? It must be the first, because the second difference can be shown to be purely cosmetic. Indeed, define the RUDRATA PATH problem to be just like RUDRATA CYCLE, except that the goal is now to find a *path* that goes through each vertex exactly once. As we will soon see, there is a precise equivalence between the two versions of the Rudrata problem.

Cuts and bisections

A *cut* is a set of edges whose removal leaves a graph disconnected. It is often of interest to find small cuts, and the MINIMUM CUT problem is, given a graph and a

[2] In the literature this problem is known as the *Hamilton cycle* problem, after the great Irish mathematician who rediscovered it in the 19th century.

Figure 8.3 What is the smallest cut in this graph?



budget b, to find a cut with at most b edges. For example, the smallest cut in Figure 8.3 is of size 3. This problem can be solved in polynomial time by $n-1$ max-flow computations: give each edge a capacity of 1, and find the maximum flow between some fixed node and every single other node. The smallest such flow will correspond (via the max-flow min-cut theorem) to the smallest cut. Can you see why? We've also seen a very different, randomized algorithm for this problem (page 140).

In many graphs, such as the one in Figure 8.3, the smallest cut leaves just a singleton vertex on one side—it consists of all edges adjacent to this vertex. Far more interesting are small cuts that partition the vertices of the graph into nearly equal-sized sets. More precisely, the BALANCED CUT problem is this: given a graph with n vertices and a budget b, partition the vertices into two sets S and T such that $|S|, |T| \geq n/3$ and such that there are at most b edges between S and T. Another hard problem.

Balanced cuts arise in a variety of important applications, such as *clustering*. Consider for example the problem of segmenting an image into its constituent components (say, an elephant standing in a grassy plain with a clear blue sky above). A good way of doing this is to create a graph with a node for each pixel of the image and to put an edge between nodes whose corresponding pixels are spatially close together and are also similar in color. A single object in the image (like the elephant, say) then corresponds to a set of highly connected vertices in the graph. A balanced cut is therefore likely to divide the pixels into two clusters without breaking apart any of the primary constituents of the image. The first cut might, for instance, separate the elephant on the one hand from the sky and from grass on the other. A further cut would then be needed to separate the sky from the grass.

Integer linear programming

Even though the simplex algorithm is not polynomial time, we mentioned in Chapter 7 that there *is* a different, polynomial algorithm for linear programming. Therefore, linear programming is efficiently solvable both in practice and in theory. But the situation changes completely if, in addition to specifying a linear objective function

and linear inequalities, we also constrain the solution (the values for the variables) to be *integer*. This latter problem is called INTEGER LINEAR PROGRAMMING (ILP). Let's see how we might formulate it as a search problem. We are given a set of linear inequalities $\mathbf{Ax} \leq \mathbf{b}$, where $\mathbf{A}$ is an $m \times n$ matrix and $\mathbf{b}$ is an m-vector; an objective function specified by an n-vector $\mathbf{c}$; and finally, a *goal* g (the counterpart of a budget in maximization problems). We want to find a nonnegative *integer* n-vector $\mathbf{x}$ such that $\mathbf{Ax} \leq \mathbf{b}$ and $\mathbf{c} \cdot \mathbf{x} \geq g$.

But there is a redundancy here: the last constraint $\mathbf{c} \cdot \mathbf{x} \geq g$ is itself a linear inequality and can be absorbed into $\mathbf{Ax} \leq \mathbf{b}$. So, we define ILP to be following search problem: given $\mathbf{A}$ and $\mathbf{b}$, find a nonnegative integer vector $\mathbf{x}$ satisfying the inequalities $\mathbf{Ax} \leq \mathbf{b}$, or report that none exists. Despite the many crucial applications of this problem, and intense interest by researchers, no efficient algorithm is known for it.

There is a particularly clean special case of ILP that is very hard in and of itself: the goal is to find a vector $\mathbf{x}$ of 0's and 1's satisfying $\mathbf{Ax} = \mathbf{1}$, where $\mathbf{A}$ is an $m \times n$ matrix with $0 - 1$ entries and $\mathbf{1}$ is the m-vector of all 1's. It should be apparent from the reductions in Section 7.1.4 that this is indeed a special case of ILP. We call it ZERO-ONE EQUATIONS (ZOE).

We have now introduced a number of important search problems, some of which are familiar from earlier chapters and for which there are efficient algorithms, and others which are different in small but crucial ways that make them very hard computational problems. To complete our story we will introduce a few more hard problems, which will play a role later in the chapter, when we relate the computational difficulty of all these problems. The reader is invited to skip ahead to Section 8.2 and then return to the definitions of these problems as required.

Three-dimensional matching

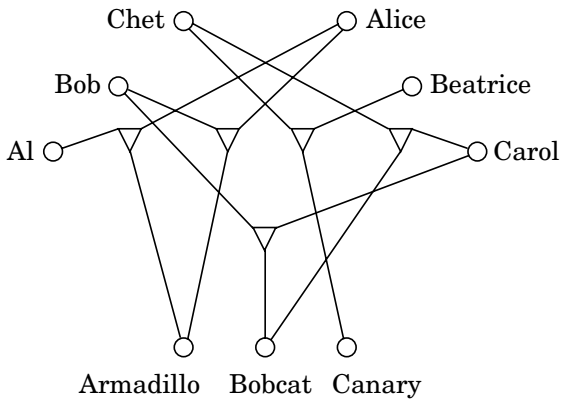
Recall the BIPARTITE MATCHING problem: given a bipartite graph with n nodes on each side (the *boys* and the *girls*), find a set of n disjoint edges, or decide that no such set exists. In Section 7.3, we saw how to efficiently solve this problem by a reduction to maximum flow. However, there is an interesting generalization, called 3D MATCHING, for which no polynomial algorithm is known. In this new setting, there are n boys and n girls, but also n *pets*, and the compatibilities among them are specified by a set of *triples*, each containing a boy, a girl, and a pet. Intuitively, a triple (b, g, p) means that boy b, girl g, and pet p get along well together. We want to find n disjoint triples and thereby create n harmonious households.

Can you spot a solution in Figure 8.4?

Independent set, vertex cover, and clique

In the INDEPENDENT SET problem (recall Section 6.7) we are given a graph and an integer g, and the aim is to find g vertices that are independent, that is, no two of which have an edge between them. Can you find an independent set of three vertices in Figure 8.5? How about four vertices? We saw in Section 6.7 that this

Figure 8.4 A more elaborate matchmaking scenario. Each triple is shown as a triangular-shaped node joining boy, girl, and pet.

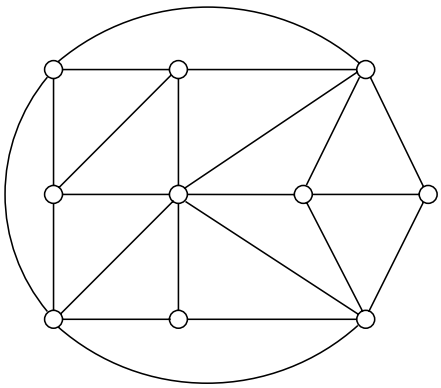


problem can be solved efficiently on trees, but for general graphs no polynomial algorithm is known.

There are many other search problems about graphs. In VERTEX COVER, for example, the input is a graph and a budget b, and the idea is to find b vertices that cover (touch) every edge. Can you cover all edges of Figure 8.5 with seven vertices? With six? (And do you see the intimate connection to the INDEPENDENT SET problem?)

VERTEX COVER is a special case of SET COVER, which we encountered in Chapter 5. In that problem, we are given a set E and several subsets of it, $S_1, \ldots, S_m$, along with

Figure 8.5 What is the size of the largest independent set in this graph?



a budget b. We are asked to select b of these subsets so that their union is E. VERTEX COVER is the special case in which E consists of the edges of a graph, and there is a subset S_i for each vertex, containing the edges adjacent to that vertex. Can you see why 3D MATCHING is also a special case of SET COVER?

And finally there is the CLIQUE problem: given a graph and a goal g, find a set of g vertices such that all possible edges between them are present. What is the largest clique in Figure 8.5?

Longest path

We know the shortest-path problem can be solved very efficiently, but how about the LONGEST PATH problem? Here we are given a graph G with nonnegative edge weights and two distinguished vertices s and t, along with a goal g. We are asked to find a path from s to t with total weight at least g. Naturally, to avoid trivial solutions we require that the path be *simple*, containing no repeated vertices.

No efficient algorithm is known for this problem (which sometimes also goes by the name of TAXICAB RIP-OFF).

Knapsack and subset sum

Recall the KNAPSACK problem (Section 6.4): we are given integer weights $w_1, \ldots, w_n$ and integer values $v_1, \ldots, v_n$ for n items. We are also given a weight capacity W and a goal g (the former is present in the original optimization problem, the latter is added to make it a search problem). We seek a set of items whose total weight is at most W and whose total value is at least g. As always, if no such set exists, we should say so.

In Section 6.4, we developed a dynamic programming scheme for KNAPSACK with running time $O(nW)$, which we noted is exponential in the input size, since it involves W rather than $\log W$. And we have the usual exhaustive algorithm as well, which looks at all subsets of items—all 2^n of them. Is there a polynomial algorithm for KNAPSACK? Nobody knows of one.

But suppose that we are interested in the variant of the knapsack problem in which the integers are coded in *unary*—for instance, by writing $IIIIIIIIIIII$ for 12. This is admittedly an exponentially wasteful way to represent integers, but it does define a legitimate problem, which we could call UNARY KNAPSACK. It follows from our discussion that this somewhat artificial problem does have a polynomial algorithm.

A different variation: suppose now that each item's value is equal to its weight (all given in binary), and to top it off, the goal g is the same as the capacity W. (To adapt the silly break-in story whereby we first introduced the knapsack problem, the items are all gold nuggets, and the burglar wants to fill his knapsack to the hilt.) This special case is tantamount to finding a subset of a given set of integers that adds up to exactly W. Since it is a special case of KNAPSACK, it cannot be any harder. But could it be polynomial? As it turns out, this problem, called SUBSET SUM, is also very hard.

At this point one could ask: If SUBSET SUM is a special case that happens to be as hard as the general KNAPSACK problem, why are we interested in it? The reason is *simplicity*. In the complicated calculus of reductions between search problems that we shall develop in this chapter, conceptually simple problems like SUBSET SUM and 3SAT are invaluable.

8.2 NP-complete problems

Hard problems, easy problems:
In short, the world is full of search problems, some of which can be solved efficiently, while others seem to be very hard. This is depicted in the following table.

Hard problems (**NP**-complete)	Easy problems (in **P**)
3SAT	2SAT, HORN SAT
TRAVELING SALESMAN PROBLEM	MINIMUM SPANNING TREE
LONGEST PATH	SHORTEST PATH
3D MATCHING	BIPARTITE MATCHING
KNAPSACK	UNARY KNAPSACK
INDEPENDENT SET	INDEPENDENT SET on trees
INTEGER LINEAR PROGRAMMING	LINEAR PROGRAMMING
RUDRATA PATH	EULER PATH
BALANCED CUT	MINIMUM CUT

This table is worth contemplating. On the right we have problems that can be solved efficiently. On the left, we have a bunch of hard nuts that have escaped efficient solution over many decades or centuries.

The various problems on the right can be solved by algorithms that are specialized and diverse: dynamic programming, network flow, graph search, greedy. These problems are easy for a variety of different reasons.

In stark contrast, the problems on the left *are all difficult for the same reason*! At their core, they are all the same problem, just in different disguises! They are all *equivalent*: as we shall see in Section 8.3, each of them can be reduced to any of the others—and back.

P and NP
It's time to introduce some important concepts. We know what a search problem is: its defining characteristic is that any proposed solution can be quickly checked for correctness, in the sense that there is an efficient checking algorithm C that takes as input the given instance I (the data specifying the problem to be solved), as well as the proposed solution S, and outputs `true` if and only if S really is a solution

Why P and NP?

Okay, **P** must stand for "polynomial." But why use the initials **NP** (the common chatroom abbreviation for "no problem") to describe the class of search problems, some of which are terribly hard?

NP stands for "nondeterministic polynomial time," a term going back to the roots of complexity theory. Intuitively, it means that a solution to any search problem can be found and verified in polynomial time by a special (and quite unrealistic) sort of algorithm, called a *nondeterministic algorithm*. Such an algorithm has the power of *guessing* correctly at every step.

Incidentally, the original definition of **NP** (and its most common usage to this day) was not as a class of search problems, but as a class of *decision problems*: algorithmic questions that can be answered by yes or no. Example: "Is there a truth assignment that satisfies this Boolean formula?" But this too reflects a historical reality: At the time the theory of **NP**-completeness was being developed, researchers in the theory of computation were interested in formal languages, a domain in which such decision problems are of central importance.

to instance I. Moreover the running time of $C(I, S)$ is bounded by a polynomial in $|I|$, the length of the instance. *We denote the class of all search problems by* **NP.**

We've seen many examples of **NP** search problems that are solvable in polynomial time. In such cases, there is an algorithm that takes as input an instance I and has a running time polynomial in $|I|$. If I has a solution, the algorithm returns such a solution; and if I has no solution, the algorithm correctly reports so. *The class of all search problems that can be solved in polynomial time is denoted* **P.** Hence, all the search problems on the right-hand side of the table are in **P.**

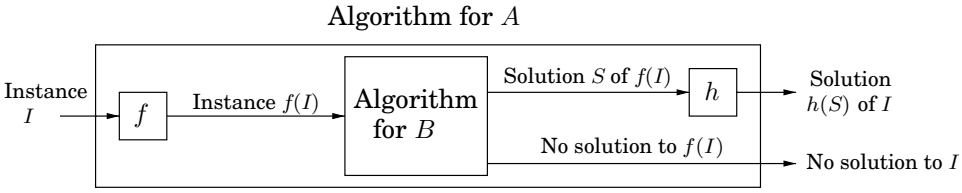
Are there search problems that cannot be solved in polynomial time? In other words, is $\mathbf{P} \neq \mathbf{NP}$**?** Most algorithms researchers think so. It is hard to believe that exponential search can always be avoided, that a simple trick will crack all these hard problems, famously unsolved for decades and centuries. And there is a good reason for mathematicians to believe that $\mathbf{P} \neq \mathbf{NP}$—the task of finding a proof for a given mathematical assertion is a search problem and is therefore in **NP** (after all, when a formal proof of a mathematical statement is written out in excruciating detail, it can be checked mechanically, line by line, by an efficient algorithm). So if $\mathbf{P} = \mathbf{NP}$**,** there would be an efficient way to prove theorems, thus eliminating the need for mathematicians! All in all, there are a variety of reasons why it is widely believed that $\mathbf{P} \neq \mathbf{NP}$**.** However, proving this has turned out to be extremely difficult, one of the deepest and most important unsolved puzzles of mathematics.

Reductions, again

Even if we accept that $\mathbf{P} \neq \mathbf{NP}$**,** what about the specific problems on the left side of the table? On the basis of what evidence do we believe that these particular problems

have no efficient algorithm (besides, of course, the historical fact that many clever mathematicians and computer scientists have tried hard and failed to find any)? Such evidence is provided by *reductions*, which translate one search problem into another. What they demonstrate is that the problems on the left side of the table are all, in some sense, *exactly the same problem*, except that they are stated in different languages. What's more, we will also use reductions to show that these problems are the *hardest* search problems in **NP**—if even one of them has a polynomial time algorithm, then *every* problem in **NP** has a polynomial time algorithm. Thus if we believe that $\mathbf{P} \neq \mathbf{NP}$, then all these search problems are hard.

We defined reductions in Chapter 7 and saw many examples of them. Let's now specialize this definition to search problems. A *reduction* from search problem A to search problem B is a polynomial-time algorithm f that transforms any instance I of A into an instance $f(I)$ of B, together with another polynomial-time algorithm h that maps any solution S of $f(I)$ back into a solution $h(S)$ of I; see the following diagram. If $f(I)$ has no solution, then neither does I. These two translation procedures f and h imply that any algorithm for B can be converted into an algorithm for A by bracketing it between f and h.



And now we can finally define the class of the hardest search problems.

A search problem is **NP***-complete if all other search problems reduce to it.*

This is a very strong requirement indeed. For a problem to be **NP**-complete, it must be useful in solving every search problem in the world! It is remarkable that such problems exist. But they do, and the first column of the table we saw earlier is filled with the most famous examples. In Section 8.3 we shall see how all these problems reduce to one another, and also why all other search problems reduce to them.

Factoring

One last point: we started off this book by introducing another famously hard search problem: FACTORING, the task of finding all prime factors of a given integer. But the difficulty of FACTORING is of a different nature than that of the other hard search problems we have just seen. For example, nobody believes that FACTORING is **NP**-complete. One major difference is that, in the case of FACTORING, the definition does not contain the now familiar clause "or report that none exists." A number can *always* be factored into primes.

The two ways to use reductions

So far in this book the purpose of a reduction from a problem A to a problem B has been straightforward and honorable: We know how to solve B efficiently, and we want to use this knowledge to solve A. In this chapter, however, reductions from A to B serve a somewhat perverse goal: we know A is hard, and we use the reduction to prove that B is hard as well!

If we denote a reduction from A to B by

$$A \longrightarrow B$$

then we can say that *difficulty* flows in the direction of the arrow, while *efficient algorithms* move in the opposite direction. It is through this propagation of difficulty that we know **NP**-complete problems are hard: all other search problems reduce to them, and thus each **NP**-complete problem contains the complexity of all search problems. If even one **NP**-complete problem is in **P**, then **P = NP.**

Reductions also have the convenient property that they *compose*.

$$\text{If } A \longrightarrow B \text{ and } B \longrightarrow C, \text{ then } A \longrightarrow C\,.$$

To see this, observe first of all that any reduction is completely specified by the pre- and postprocessing functions f and h (see the reduction diagram). If (f_{AB}, h_{AB}) and (f_{BC}, h_{BC}) define the reductions from A to B and from B to C, respectively, then a reduction from A to C is given by compositions of these functions: $f_{BC} \circ f_{AB}$ maps an instance of A to an instance of C and $h_{AB} \circ h_{BC}$ sends a solution of C back to a solution of A.

This means that once we know a problem A is **NP**-complete, we can use it to prove that a new search problem B is also **NP**-complete, simply by reducing A to B. Such a reduction establishes that all problems in **NP** reduce to B, via A.

Figure 8.6 The space **NP** of all search problems, assuming **P** $\neq$ **NP.**

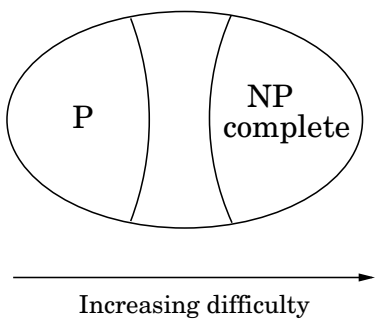
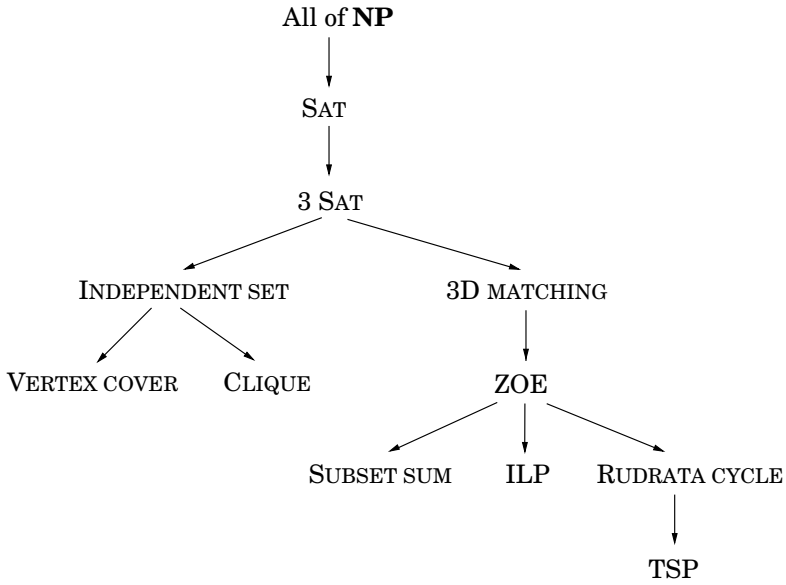


Figure 8.7 Reductions between search problems.



Another difference (possibly not completely unrelated) is this: as we shall see in Chapter 10, FACTORING succumbs to the power of *quantum computation*—while SAT, TSP, and the other **NP**-complete problems do not seem to.

8.3 The reductions

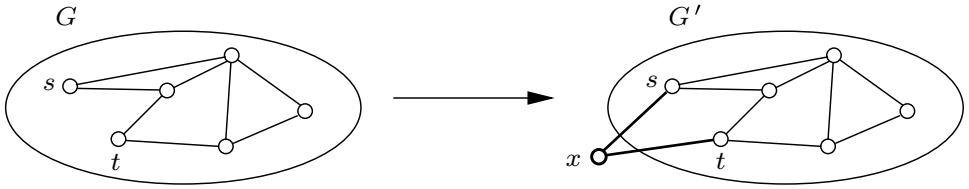
We shall now see that the search problems of Section 8.1 can be reduced to one another as depicted in Figure 8.7. As a consequence, they are all **NP**-complete.

Before we tackle the specific reductions in the tree, let's warm up by relating two versions of the Rudrata problem.

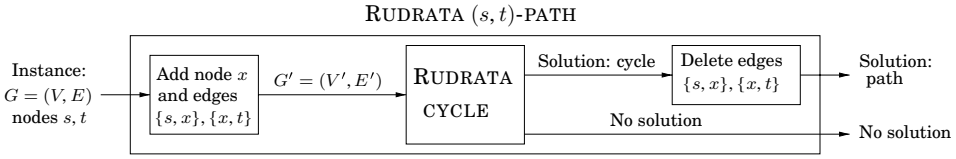
RUDRATA (s,t)-PATH $\longrightarrow$ RUDRATA CYCLE

Recall the RUDRATA CYCLE problem: given a graph, is there a cycle that passes through each vertex exactly once? We can also formulate the closely related RUDRATA (s, t)-PATH problem, in which two vertices s and t are specified, and we want a path starting at s and ending at t that goes through each vertex exactly once. Is it possible that RUDRATA CYCLE is easier than RUDRATA (s, t)-PATH? We will show by a reduction that the answer is no.

The reduction maps an instance $(G = (V, E), s, t)$ of RUDRATA (s, t)-PATH into an instance $G' = (V', E')$ of RUDRATA CYCLE as follows: G' is simply G with an additional vertex x and two new edges $\{s, x\}$ and $\{x, t\}$. For instance:



So $V' = V \cup \{x\}$, and $E' = E \cup \{\{s, x\}, \{x, t\}\}$. How do we recover a Rudrata (s, t)-*path* in G given any Rudrata *cycle* in G'? Easy, we just delete the edges $\{s, x\}$ and $\{x, t\}$ from the cycle.



To confirm the validity of this reduction, we have to show that it works in the case of either outcome depicted.

1. When the instance of RUDRATA CYCLE has a solution.

Since the new vertex x has only two neighbors, s and t, any Rudrata cycle in G' must consecutively traverse the edges $\{t, x\}$ and $\{x, s\}$. The rest of the cycle then traverses every other vertex en route from s to t. Thus deleting the two edges $\{t, x\}$ and $\{x, s\}$ from the Rudrata cycle gives a Rudrata path from s to t in the original graph G.

2. When the instance of RUDRATA CYCLE does not have a solution.

In this case we must show that the original instance of RUDRATA (s, t)-PATH cannot have a solution either. It is usually easier to prove the contrapositive, that is, to show that if there is a Rudrata (s, t)-path in G, then there is also a Rudrata cycle in G'. But this is easy: just add the two edges $\{t, x\}$ and $\{x, s\}$ to the Rudrata path to close the cycle.

One last detail, crucial but typically easy to check, is that the pre- and postprocessing functions take time polynomial in the size of the instance (G, s, t).

It is also possible to go in the other direction and reduce RUDRATA CYCLE to RUDRATA (s, t)-PATH. Together, these reductions demonstrate that the two Rudrata variants are in essence the same problem—which is not too surprising, given that their

descriptions are almost the same. But most of the other reductions we will see are between pairs of problems that, on the face of it, look quite different. To show that they are essentially the same, our reductions will have to cleverly translate between them.

3SAT $\longrightarrow$ INDEPENDENT SET

One can hardly think of two more different problems. In 3SAT the input is a set of clauses, each with three or fewer literals, for example

$$(\overline{x} \vee y \vee \overline{z})\ (x \vee \overline{y} \vee z)\ (x \vee y \vee z)\ (\overline{x} \vee \overline{y}),$$

and the aim is to find a satisfying truth assignment. In INDEPENDENT SET the input is a graph and a number g, and the problem is to find a set of g pairwise non-adjacent vertices. We must somehow relate Boolean logic with graphs!

Let us think. To form a satisfying truth assignment we must pick one literal from each clause and give it the value `true`. But our choices must be consistent: if we choose $\overline{x}$ in one clause, we cannot choose x in another. Any consistent choice of literals, one from each clause, specifies a truth assignment (variables for which neither literal has been chosen can take on either value).

So, let us represent a clause, say $(x \vee \overline{y} \vee z)$, by a triangle, with vertices labeled $x, \overline{y}, z$. Why triangle? Because a triangle has its three vertices maximally connected, and thus forces us to pick only one of them for the independent set. Repeat this construction for all clauses—a clause with two literals will be represented simply by an edge joining the literals. (A clause with one literal is silly and can be removed in a preprocessing step, since the value of the variable is determined.) In the resulting graph, an independent set has to pick at most one literal from each group (clause). To force exactly one choice from each clause, take the goal g to be the number of clauses; in our example, $g = 4$.

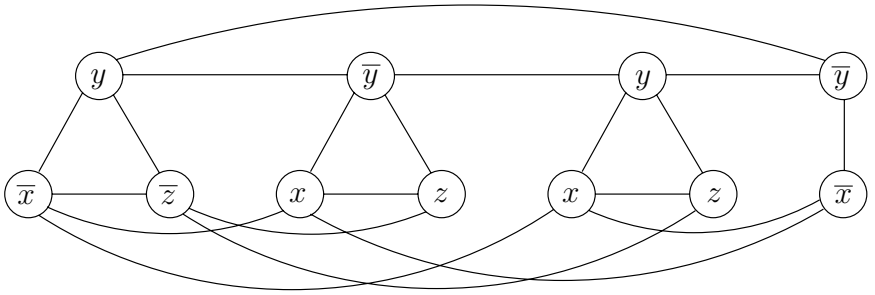
All that is missing now is a way to prevent us from choosing opposite literals (that is, both x and $\overline{x}$) in different clauses. But this is easy: put an edge between any two vertices that correspond to opposite literals. The resulting graph for our example is shown in Figure 8.8.

Let's recap the construction. Given an instance I of 3SAT, we create an instance (G, g) of INDEPENDENT SET as follows.

- Graph G has a triangle for each clause (or just an edge, if the clause has two literals), with vertices labeled by the clause's literals, and has additional edges between any two vertices that represent opposite literals.
- The goal g is set to the number of clauses.

Clearly, this construction takes polynomial time. However, recall that for a reduction we do not just need an efficient way to map instances of the first problem to instances of the second (the function f in the diagram on page 245), but also a way to

Figure 8.8 The graph corresponding to $(\overline{x} \vee y \vee \overline{z})\ (x \vee \overline{y} \vee z)\ (x \vee y \vee z)\ (\overline{x} \vee \overline{y})$.



reconstruct a solution to the first instance from any solution of the second (the function h). As always, there are two things to show.

1. Given an independent set S of g vertices in G, it is possible to efficiently recover a satisfying truth assignment to I.

For any variable x, the set S cannot contain vertices labeled both x and $\overline{x}$, because any such pair of vertices is connected by an edge. So assign x a value of `true` if S contains a vertex labeled x, and a value of `false` if S contains a vertex labeled $\overline{x}$ (if S contains neither, then assign either value to x). Since S has g vertices, it must have one vertex per clause; this truth assignment satisfies those particular literals, and thus satisfies all clauses.

2. If graph G has no independent set of size g, then the Boolean formula I is unsatisfiable.

It is usually cleaner to prove the contrapositive, that if I has a satisfying assignment then G has an independent set of size g. This is easy: for each clause, pick any literal whose value under the satisfying assignment is `true` (there must be at least one such literal), and add the corresponding vertex to S. Do you see why set S must be independent?

SAT $\longrightarrow$ 3SAT

This is an interesting and common kind of reduction, from a problem to a *special case* of itself. We want to show that the problem remains hard even if its inputs are restricted somehow—in the present case, even if all clauses are restricted to have ≤ 3 literals. Such reductions modify the given instance so as to get rid of the forbidden feature (clauses with ≥ 4 literals) while keeping the instance essentially the same, in that we can read off a solution to the original instance from any solution of the modified one.

Here's the trick for reducing SAT to 3SAT: given an instance I of SAT, use exactly the same instance for 3SAT, except that any clause with more than three literals, $(a_1 \vee a_2 \vee \cdots \vee a_k)$ (where the a_i's are literals and $k > 3$), is replaced by a set of clauses,

$$(a_1 \vee a_2 \vee y_1)\ (\overline{y}_1 \vee a_3 \vee y_2)\ (\overline{y}_2 \vee a_4 \vee y_3)\ \cdots\ (\overline{y}_{k-3} \vee a_{k-1} \vee a_k),$$

where the y_i's are new variables. Call the resulting 3SAT instance I'. The conversion from I to I' is clearly polynomial time.

Why does this reduction work? I' is equivalent to I in terms of satisfiability, because for any assignment to the a_i's,

$$\left\{ \begin{array}{c} (a_1 \vee a_2 \vee \cdots \vee a_k) \\ \text{is satisfied} \end{array} \right\} \Longleftrightarrow \left\{ \begin{array}{c} \text{there is a setting of the } y_i\text{'s for which} \\ (a_1 \vee a_2 \vee y_1)\ (\overline{y}_1 \vee a_3 \vee y_2)\ \cdots\ (\overline{y}_{k-3} \vee a_{k-1} \vee a_k) \\ \text{are all satisfied} \end{array} \right\}.$$

To see this, first suppose that the clauses on the right are all satisfied. Then at least one of the literals $a_1, \ldots, a_k$ must be true—otherwise y_1 would have to be true, which would in turn force y_2 to be true, and so on, eventually falsifying the last clause. But this means $(a_1 \vee a_2 \vee \cdots \vee a_k)$ is also satisfied.

Conversely, if $(a_1 \vee a_2 \vee \cdots \vee a_k)$ is satisfied, then some a_i must be true. Set $y_1, \ldots, y_{i-2}$ to `true` and the rest to `false`. This ensures that the clauses on the right are all satisfied.

Thus, any instance of SAT can be transformed into an equivalent instance of 3SAT. In fact, 3SAT remains hard even under the further restriction that no variable appears in more than three clauses. To show this, we must somehow get rid of any variable that appears too many times.

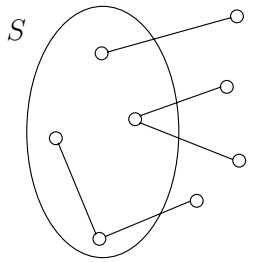
Here's the reduction from 3SAT to its constrained version. Suppose that in the 3SAT instance, variable x appears in $k > 3$ clauses. Then replace its first appearance by x_1, its second appearance by x_2, and so on, replacing each of its k appearances by a different new variable. Finally, add the clauses

$$(\overline{x}_1 \vee x_2)\ (\overline{x}_2 \vee x_3)\ \cdots\ (\overline{x}_k \vee x_1).$$

And repeat for every variable that appears more than three times.

It is easy to see that in the new formula no variable appears more than three times (and in fact, no literal appears more than twice). Furthermore, the extra clauses involving $x_1, x_2, \ldots, x_k$ constrain these variables to have the same value; do you see why? Hence the original instance of 3SAT is satisfiable if and only if the constrained instance is satisfiable.

Figure 8.9 S is a vertex cover if and only if $V - S$ is an independent set.



INDEPENDENT SET $\longrightarrow$ VERTEX COVER

Some reductions rely on ingenuity to relate two very different problems. Others simply record the fact that one problem is a thin disguise of another. To reduce INDEPENDENT SET to VERTEX COVER we just need to notice that a set of nodes S is a vertex cover of graph $G = (V, E)$ (that is, S touches every edge in E) if and only if the remaining nodes, $V - S$, are an independent set of G (Figure 8.9).

Therefore, to solve an instance (G, g) of INDEPENDENT SET, simply look for a vertex cover of G with $|V| - g$ nodes. If such a vertex cover exists, then take all nodes *not* in it. If no such vertex cover exists, then G cannot possibly have an independent set of size g.

INDEPENDENT SET $\longrightarrow$ CLIQUE

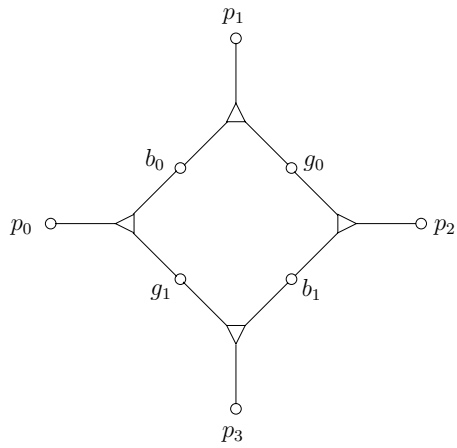
INDEPENDENT SET and CLIQUE are also easy to reduce to one another. Define the *complement* of a graph $G = (V, E)$ to be $\overline{G} = (V, \overline{E})$, where $\overline{E}$ contains precisely those unordered pairs of vertices that are not in E. Then a set of nodes S is an independent set of G if and only if S is a clique of $\overline{G}$. To paraphrase, these nodes have no edges between them in G if and only if they have all possible edges between them in $\overline{G}$.

Therefore, we can reduce INDEPENDENT SET to CLIQUE by mapping an instance (G, g) of INDEPENDENT SET to the corresponding instance $(\overline{G}, g)$ of CLIQUE; the solution to both is identical.

3SAT $\longrightarrow$ 3D MATCHING

Again, two very different problems. We must reduce 3SAT to the problem of finding, among a set of boy-girl-pet triples, a subset that contains each boy, each girl, and each pet exactly once. In short, we must design sets of boy-girl-pet triples that somehow behave like Boolean variables and gates!

Consider the following set of four triples, each represented by a triangular node joining a boy, girl, and pet:



Suppose that the two boys b_0 and b_1 and the two girls g_0 and g_1 are not involved in any other triples. (The four pets $p_0, \ldots, p_3$ will of course belong to other triples as well; for otherwise the instance would trivially have no solution.) Then any matching must contain either the two triples (b_0, g_1, p_0), (b_1, g_0, p_2) or the two triples (b_0, g_0, p_1), (b_1, g_1, p_3), because these are the only ways in which these two boys and girls can find any match. Therefore, this "gadget" has two possible states: it behaves like a Boolean variable!

To then transform an instance of 3SAT to one of 3D MATCHING, we start by creating a copy of the preceding gadget for *each* variable x. Call the resulting nodes p_{x1}, b_{x0}, g_{x1}, and so on. The intended interpretation is that boy b_{x0} is matched with girl g_{x1} if $x = \texttt{true}$, and with girl g_{x0} if $x = \texttt{false}$.

Next we must create triples that somehow mimic clauses. For each clause, say $c = (x \vee \overline{y} \vee z)$, introduce a new boy b_c and a new girl g_c. They will be involved in three triples, one for each literal in the clause. And the pets in these triples must reflect the three ways whereby the clause can be satisfied: (1) $x = \texttt{true}$, (2) $y = \texttt{false}$, (3) $z = \texttt{true}$. For (1), we have the triple (b_c, g_c, p_{x1}), where p_{x1} is the pet p_1 in the gadget for x. Here is why we chose p_1: if $x = \texttt{true}$, then b_{x0} is matched with g_{x1} and b_{x1} with g_{x0}, and so pets p_{x0} and p_{x2} are taken. In which case b_c and g_c can be matched with p_{x1}. But if $x = \texttt{false}$, then p_{x1} and p_{x3} are taken, and so g_c and b_c cannot be accommodated this way. We do the same thing for the other two literals of the clause, which yield triples involving b_c and g_c with either p_{y0} or p_{y2} (for the negated variable y) and with either p_{z1} or p_{z3} (for variable z).

We have to make sure that for every occurrence of a literal in a clause c there is a different pet to match with b_c and g_c. But this is easy: by an earlier reduction we can assume that no literal appears more than twice, and so each variable gadget has enough pets, two for negated occurrences and two for unnegated.

The reduction now seems complete: from any matching we can recover a satisfying truth assignment by simply looking at each variable gadget and seeing with which girl b_{x0} was matched. And from any satisfying truth assignment we can match the gadget corresponding to each variable x so that triples (b_{x0}, g_{x1}, p_{x0}) and (b_{x1}, g_{x0}, p_{x2}) are chosen if $x = \texttt{true}$ and triples (b_{x0}, g_{x0}, p_{x1}) and (b_{x1}, g_{x1}, p_{x3}) are chosen if $x = \texttt{false}$; and for each clause c match b_c and g_c with the pet that corresponds to one of its satisfying literals.

But one last problem remains: in the matching defined at the end of the last paragraph, *some pets may be left unmatched*. In fact, if there are n variables and m clauses, then exactly $2n - m$ pets *will* be left unmatched (you can check that this number is sure to be positive, because we have at most three occurrences of every variable, and at least two literals in every clause). But this is easy to fix: Add $2n - m$ new boy-girl couples that are "generic animal-lovers," and match them by triples with all the pets!

3D MATCHING $\longrightarrow$ ZOE

Recall that in ZOE we are given an $m \times n$ matrix $\mathbf{A}$ with $0-1$ entries, and we must find a $0-1$ vector $\mathbf{x} = (x_1, \ldots, x_n)$ such that the m equations

$$\mathbf{Ax} = \mathbf{1}$$

are satisfied, where by $\mathbf{1}$ we denote the column vector of all 1's. How can we express the 3D MATCHING problem in this framework?

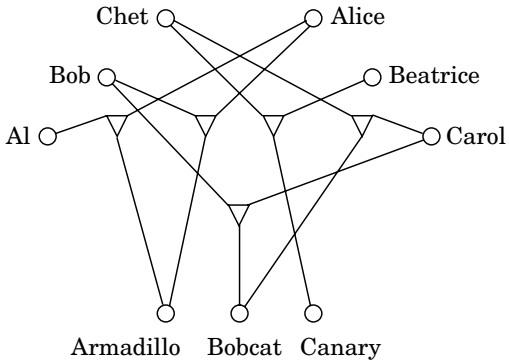
ZOE and ILP are very useful problems precisely because they provide a format in which many combinatorial problems can be expressed. In such a formulation we think of the $0-1$ variables as describing a solution, and we write equations expressing the constraints of the problem.

For example, here is how we express an instance of 3D MATCHING (m boys, m girls, m pets, and n boy-girl-pet triples) in the language of ZOE. We have $0-1$ variables $x_1, \ldots, x_n$, one per triple, where $x_i = 1$ means that the ith triple is chosen for the matching, and $x_i = 0$ means that it is not chosen.

Now all we have to do is write equations stating that the solution described by the x_i's is a legitimate matching. For each boy (or girl, or pet), suppose that the triples containing him (or her, or it) are those numbered $j_1, j_2, \ldots, j_k$; the appropriate equation is then

$$x_{j_1} + x_{j_2} + \cdots + x_{j_k} = 1,$$

which states that exactly one of these triples must be included in the matching. For example, here is the $\mathbf{A}$ matrix for an instance of 3D MATCHING we saw earlier.



$$\mathbf{A} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{pmatrix}$$

The five columns of **A** correspond to the five triples, while the nine rows are for Al, Bob, Chet, Alice, Beatrice, Carol, Armadillo, Bobcat, and Canary, respectively.

It is straightforward to argue that solutions to the two instances translate back and forth.

ZOE⟶SUBSET SUM

This is a reduction between two special cases of ILP: one with many equations but only $0-1$ coefficients, and the other with a single equation but arbitrary integer coefficients. The reduction is based on a simple and time-honored idea: $0-1$ vectors can encode numbers!

For example, given this instance of ZOE:

$$\mathbf{A} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix},$$

we are looking for a set of columns of **A** that, added together, make up the all-1's vector. But if we think of the columns as binary integers (read from top to bottom), we are looking for a subset of the integers 18, 5, 4, 8 that add up to the binary integer $11111_2 = 31$. And this is an instance of SUBSET SUM. The reduction is complete!

Except for one detail, the one that usually spoils the close connection between $0-1$ vectors and binary integers: *carry*. Because of carry, 5-bit binary integers can add up to 31 (for example, $5+6+20=31$ or, in binary, $00101_2 + 00110_2 + 10100_2 = 11111_2$) even when the sum of the corresponding vectors is not $(1,1,1,1,1)$. But this is easy to fix: Think of the column vectors not as integers in base 2, but as integers in base $n+1$—one more than the number of columns. This way, since at most n integers are added, and all their digits are 0 and 1, there can be no carry, and our reduction works.

ZOE⟶ILP

3SAT is a special case of SAT—or, SAT is a generalization of 3SAT. By *special case* we mean that the instances of 3SAT are a subset of the instances of SAT (in particular, the ones with no long clauses), and the definition of solution is the same in both problems (an assignment satisfying all clauses). Consequently, there is a reduction from 3SAT to SAT, in which the input undergoes no transformation, and the solution to the target instance is also kept unchanged. In other words, functions f and h from the reduction diagram (on page 245) are both the identity.

This sounds trivial enough, but it is a very useful and common way of establishing that a problem is **NP**-complete: Simply notice that it is a generalization of a known **NP**-complete problem. For example, the SET COVER problem is **NP**-complete because it is a generalization of VERTEX COVER (and also, incidentally, of 3D MATCHING). See Exercise 8.10 for more examples.

Often it takes a little work to establish that one problem is a special case of another. The reduction from ZOE to ILP is a case in point. In ILP we are looking for an integer vector **x** that satisfies $\mathbf{Ax} \leq \mathbf{b}$, for given matrix **A** and vector **b.** To write an instance of ZOE in this precise form, we need to rewrite each equation of the ZOE instance as two inequalities (recall the transformations of Section 7.1.4), and to add for each variable x_i the inequalities $x_i \leq 1$ and $-x_i \leq 0$.

ZOE⟶RUDRATA CYCLE

In the RUDRATA CYCLE problem we seek a cycle in a graph that visits every vertex exactly once. We shall prove it **NP**-complete in two stages: first we will reduce ZOE to a generalization of RUDRATA CYCLE, called RUDRATA CYCLE WITH PAIRED EDGES, and then

Figure 8.10 Rudrata cycle with paired edges: $C = \{(e_1, e_3), (e_5, e_6), (e_4, e_5), (e_3, e_7), (e_3, e_8)\}$.

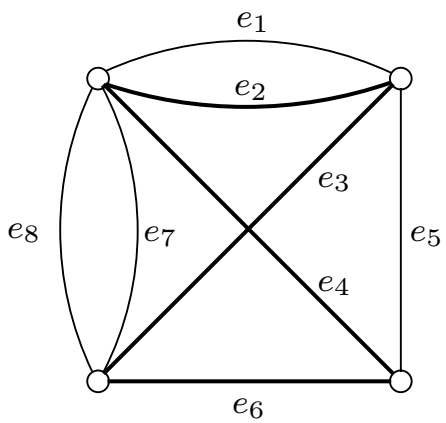
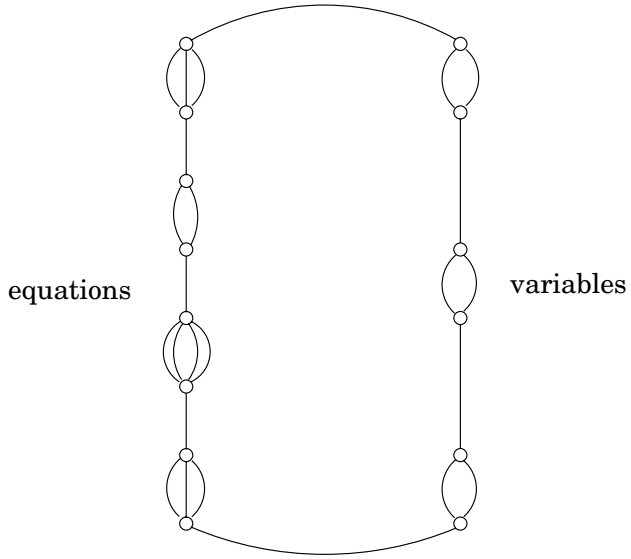


Figure 8.11 Reducing ZOE to RUDRATA CYCLE WITH PAIRED EDGES.



we shall see how to get rid of the extra features of that problem and reduce it to the plain RUDRATA CYCLE problem.

In an instance of RUDRATA CYCLE WITH PAIRED EDGES we are given a graph $G = (V, E)$ and a set $C \subseteq E \times E$ of pairs of edges. We seek a cycle that (1) visits all vertices once, like a Rudrata cycle should, and (2) for every pair of edges (e, e') in C, traverses either edge e or edge e'—*exactly one* of them. In the simple example of Figure 8.10 a solution is shown in bold. Notice that we allow two or more parallel edges between two nodes—a feature that doesn't make sense in most graph problems—since now the different copies of an edge can be paired with other copies of edges in ways that do make a difference.

Now for the reduction of ZOE to RUDRATA CYCLE WITH PAIRED EDGES. Given an instance of ZOE, $\mathbf{Ax} = \mathbf{1}$ (where $\mathbf{A}$ is an $m \times n$ matrix with $0 - 1$ entries, and thus describes m equations in n variables), the graph we construct has the very simple structure shown in Figure 8.11: a cycle that connects $m + n$ collections of parallel edges. For each variable x_i we have two parallel edges (corresponding to $x_i = 1$ and $x_i = 0$). And for each equation $x_{j_1} + \cdots + x_{j_k} = 1$ involving k variables we have k parallel edges, one for every variable appearing in the equation. This is the whole graph. Evidently, any Rudrata cycle in this graph must traverse the $m + n$ collections of parallel edges one by one, choosing one edge from each collection. This way, the cycle "chooses" for each variable a value—0 or 1—and, for each equation, a variable appearing in it.

The whole reduction can't be this simple, of course. The structure of the matrix **A** (and not just its dimensions) must be reflected somewhere, and there is one place left: the set C of pairs of edges such that exactly one edge in each pair is traversed. For every equation (recall there are m in total), and for every variable x_i appearing in it, we add to C the pair (e, e') where e is the edge corresponding to the appearance of x_i in that particular equation (on the left-hand side of Figure 8.11), and e' is the edge corresponding to the variable assignment $x_i = 0$ (on the right side of the figure). This completes the construction.

Take any solution of this instance of RUDRATA CYCLE WITH PAIRED EDGES. As discussed before, it picks a value for each variable and a variable for every equation. We claim that the values thus chosen are a solution to the original instance of ZOE. If a variable x_i has value 1, then the edge $x_i = 0$ is not traversed, and thus all edges associated with x_i on the equation side must be traversed (since they are paired in C with the $x_i = 0$ edge). So, in each equation exactly one of the variables appearing in it has value 1—which is the same as saying that all equations are satisfied. The other direction is straightforward as well: from a solution to the instance of ZOE one easily obtains an appropriate Rudrata cycle.

Getting Rid of the Edge Pairs:
So far we have a reduction from ZOE to RUDRATA CYCLE WITH PAIRED EDGES; but we are really interested in RUDRATA CYCLE, which is a special case of the problem with paired edges: the one in which the set of pairs C is empty. To accomplish our goal, we need, as usual, to find a way of getting rid of the unwanted feature—in this case the edge pairs.

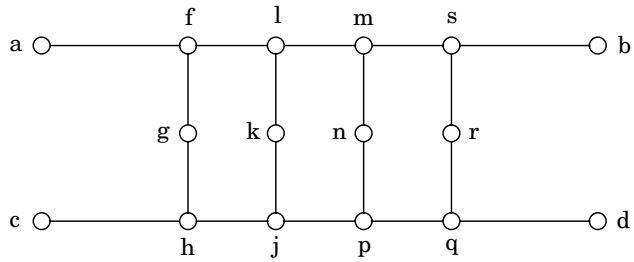
Consider the graph shown in Figure 8.12, and suppose that it is a part of a larger graph G in such a way that only the four endpoints a, b, c, d touch the rest of the graph. We claim that this graph has the following important property: *in any Rudrata cycle of G the subgraph shown must be traversed in one of the two ways shown in bold in Figure 8.12(b) and (c).* Here is why. Suppose that the cycle first enters the subgraph from vertex a continuing to f. Then it must continue to vertex g, because g has degree 2 and so it must be visited immediately after one of its adjacent nodes is visited—otherwise there is no way to include it in the cycle. Hence we must go on to node h, and here we seem to have a choice. We could continue on to j, or return to c. But if we take the second option, how are we going to visit the rest of the subgraph? (A Rudrata cycle must leave no vertex unvisited.) It is easy to see that this would be impossible, and so from h we have no choice but to continue to j and from there to visit the rest of the graph as shown in Figure 8.12(b). By symmetry, if the Rudrata cycle enters this subgraph at c, it must traverse it as in Figure 8.12(c). And these are the only two ways.

But this property tells us something important: this gadget behaves just like two edges $\{a, b\}$ and $\{c, d\}$ that are paired up in the RUDRATA CYCLE WITH PAIRED EDGES problem (see Figure 8.12(d)).

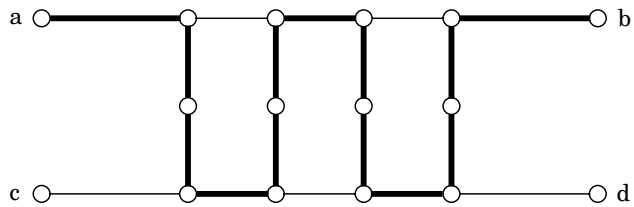
The rest of the reduction is now clear: to reduce RUDRATA CYCLE WITH PAIRED EDGES to RUDRATA CYCLE we go through the pairs in C one by one. To get rid of each

Figure 8.12 A gadget for enforcing paired behavior.

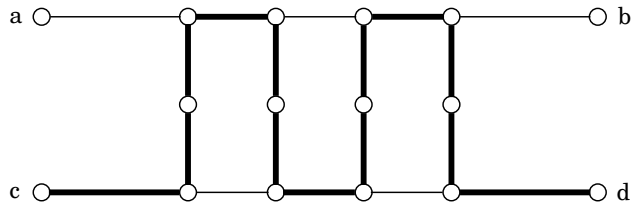
(a)



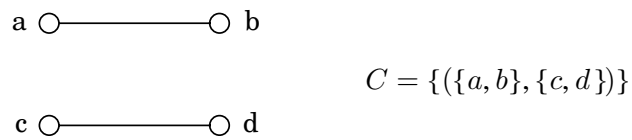
(b)



(c)



(d)



$C = \{(\{a, b\}, \{c, d\})\}$

pair $(\{a,b\},\{c,d\})$ we replace the two edges with the gadget in Figure 8.12(a). For any other pair in C that involves $\{a,b\}$, we replace the edge $\{a,b\}$ with the new edge $\{a,f\}$, where f is from the gadget: the traversal of $\{a,f\}$ is from now on an indication that edge $\{a,b\}$ in the old graph would be traversed. Similarly, $\{c,h\}$ replaces $\{c,d\}$. After $|C|$ such replacements (performed in polynomial time, since each replacement adds only 12 vertices to the graph) we are done, and the Rudrata cycles in the resulting graph will be in one-to-one correspondence with the Rudrata cycles in the original graph that conform to the constraints in C.

RUDRATA CYCLE$\longrightarrow$TSP

Given a graph $G=(V,E)$, construct the following instance of the TSP: the set of cities is the same as V, and the distance between cities u and v is 1 if $\{u,v\}$ is an edge of G and $1+\alpha$ otherwise, for some $\alpha>1$ to be determined. The budget of the TSP instance is equal to the number of nodes, $|V|$.

It is easy to see that if G has a Rudrata cycle, then the same cycle is also a tour within the budget of the TSP instance; and that conversely, if G has no Rudrata cycle, then there is no solution: the cheapest possible TSP tour has cost at least $n+\alpha$ (it must use at least one edge of length $1+\alpha$, and the total length of all $n-1$ others is at least $n-1$). Thus RUDRATA CYCLE reduces to TSP.

In this reduction, we introduced the parameter α because by varying it, we can obtain two interesting results. If $\alpha=1$, then all distances are either 1 or 2, and so this instance of the TSP satisfies the triangle inequality: if i,j,k are cities, then $d_{ij}+d_{jk}\geq d_{ik}$ (proof: $a+b\geq c$ holds for any numbers $1\leq a,b,c\leq 2$). This is a special case of the TSP which is of practical importance and which, as we shall see in Chapter 9, is in a certain sense easier, because it can be efficiently *approximated*.

If on the other hand α is large, then the resulting instance of the TSP may not satisfy the triangle inequality, but has another important property: either it has a solution of cost n or less, or all its solutions have cost at least $n+\alpha$ (which now can be arbitrarily larger than n). There can be nothing in between! As we shall see in Chapter 9, this important *gap* property implies that, unless **P = NP,** no approximation algorithm is possible.

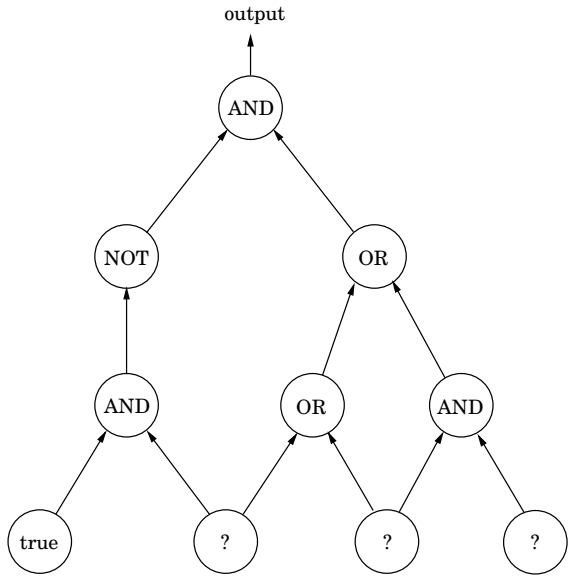
ANY PROBLEM IN **NP**$\longrightarrow$SAT

We have reduced SAT to the various search problems in Figure 8.7. Now we come full circle and argue that all these problems—and in fact all problems in **NP**—reduce to SAT.

In particular, we shall show that all problems in **NP** can be reduced to a generalization of SAT which we call CIRCUIT SAT. In CIRCUIT SAT we are given a (*Boolean*) *circuit* (see Figure 8.13, and recall Section 7.7), a dag whose vertices are *gates* of five different types:

- AND gates and OR gates have indegree 2.
- NOT gates have indegree 1.

Figure 8.13 An instance of CIRCUIT SAT.



- *Known input* gates have no incoming edges and are labeled `false` or `true`.
- *Unknown input* gates have no incoming edges and are labeled "?".

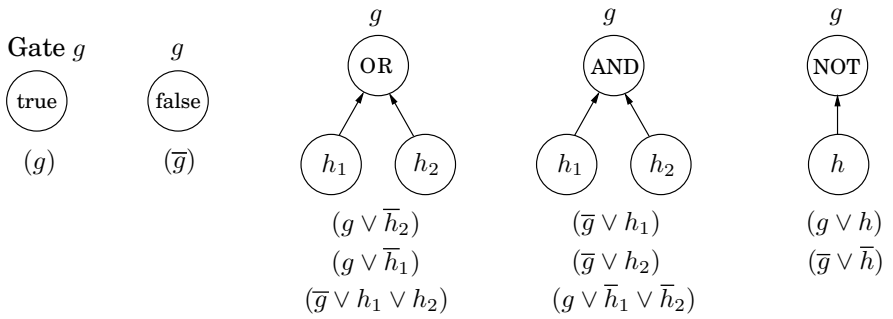
One of the sinks of the dag is designated as the *output* gate.

Given an assignment of values to the unknown inputs, we can evaluate the gates of the circuit in topological order, using the rules of Boolean logic (such as `false` $\vee$ `true` = `true`), until we obtain the value at the output gate. This is the value of the circuit for the particular assignment to the inputs. For instance, the circuit in Figure 8.13 evaluates to `false` under the assignment `true`, `false`, `true` (from left to right).

CIRCUIT SAT is then the following search problem: Given a circuit, find a truth assignment for the unknown inputs such that the output gate evaluates to `true`, or report that no such assignment exists. For example, if presented with the circuit in Figure 8.13 we could have returned the assignment (`false`, `true`, `true`) because, if we substitute these values to the unknown inputs (from left to right), the output becomes `true`.

CIRCUIT SAT is a generalization of SAT. To see why, notice that SAT asks for a satisfying truth assignment for a circuit that has this simple structure: a bunch of AND gates at the top join the clauses, and the result of this big AND is the output. Each clause is the OR of its literals. And each literal is either an unknown input gate or the NOT of one. There are no known input gates.

Going in the other direction, CIRCUIT SAT can also be reduced to SAT. Here is how we can rewrite any circuit in conjunctive normal form (the AND of clauses): for each gate g in the circuit we create a variable g, and we model the effect of the gate using a few clauses:



(Do you see that these clauses do, in fact, force exactly the desired effect?) And to finish up, if g is the output gate, we force it to be `true` by adding the clause (g). The resulting instance of SAT is equivalent to the given instance of CIRCUIT SAT: the satisfying truth assignments of this conjunctive normal form are in one-to-one correspondence with those of the circuit.

Now that we know CIRCUIT SAT reduces to SAT, we turn to our main job, showing that *all* search problems reduce to CIRCUIT SAT. So, suppose that A is a problem in **NP.** We must discover a reduction from A to CIRCUIT SAT. This sounds very difficult, *because we know almost nothing about* A!

All we know about A is that it is a search problem, so we must put this knowledge to work. The main feature of a search problem is that any solution to it can quickly be checked: there is an algorithm $\mathcal{C}$ that checks, given an instance I and a proposed solution S, whether or not S is a solution of I. Moreover, $\mathcal{C}$ makes this decision in time polynomial in the length of I (we can assume that S is itself encoded as a binary string, and we know that the length of this string is polynomial in the length of I).

Recall now our argument in Section 7.7 that any polynomial algorithm can be rendered as a circuit, whose input gates encode the input to the algorithm. Naturally, for any input length (number of input bits) the circuit will be scaled to the appropriate number of inputs, but the total number of gates of the circuit will be polynomial in the number of inputs. If the polynomial algorithm in question solves a problem that requires a yes or no answer (as is the situation with $\mathcal{C}$: "Does S encode a solution to the instance encoded by I?"), then this answer is given at the output gate.

We conclude that, given any instance I of problem A, we can construct in polynomial time a circuit whose known inputs are the bits of I, and whose unknown inputs are the bits of S, such that the output is `true` if and only if the unknown inputs spell a solution S of I. In other words, *the satisfying truth assignments to the unknown inputs of the circuit are in one-to-one correspondence with the solutions of instance I of A*. The reduction is complete.

Unsolvable problems

At least an **NP**-complete problem can be solved by *some* algorithm—the trouble is that this algorithm will be exponential. But it turns out there are perfectly decent computational problems for which *no algorithms exist at all*!

One famous problem of this sort is an arithmetical version of SAT. Given a polynomial equation in many variables, perhaps

$$x^3yz + 2y^4z^2 - 7xy^5z = 6,$$

are there integer values of x, y, z that satisfy it? There is no algorithm that solves this problem. No algorithm at all, polynomial, exponential, doubly exponential, or worse! Such problems are called *unsolvable*.

The first unsolvable problem was discovered in 1936 by Alan M. Turing, then a student of mathematics at Cambridge, England. When Turing came up with it, there were no computers or programming languages (in fact, it can be argued that these things came about later *exactly because* this brilliant thought occurred to Turing). But today we can state it in familiar terms.

Suppose that you are given a program in your favorite programming language, along with a particular input. Will the program ever terminate, once started on this input? This is a very reasonable question. Many of us would be ecstatic if we had an algorithm, call it `terminates(p,x)`, that took as input a file containing a program `p`, and a file of data `x`, and after grinding away, finally told us whether or not `p` would ever stop if started on `x`.

But how would you go about writing the program `terminates`? (If you haven't seen this before, it's worth thinking about it for a while, to appreciate the difficulty of writing such a "universal infinite-loop detector.")

Well, you can't. *Such an algorithm does not exist*!

And here is the proof: Suppose we actually had such a program `terminates(p,x)`. Then we could use it as a subroutine of the following evil program:

```
function paradox(z:file)
1: if terminates(z,z) goto 1
```

Unsolvable problems (*Continued*)

Notice what `paradox` does: it terminates if and only if program `z` does *not* terminate when given its own code as input.

You should smell trouble. What if we put this program in a file named `paradox` and we executed `paradox(paradox)`? Would this execution ever stop? Or not? Neither answer is possible. Since we arrived at this contradiction by assuming that there is an algorithm for telling whether programs terminate, we must conclude that this problem cannot be solved by any algorithm.

By the way, all this tells us something important about programming: It will never be automated, it will forever depend on discipline, ingenuity, and hackery. We now know that you can't tell whether a program has an infinite loop. But can you tell if it has a buffer overrun? Do you see how to use the unsolvability of the "halting problem" to show that this, too, is unsolvable?

Exercises

8.1. *Optimization versus search.* Recall the traveling salesman problem:

> TSP
> *Input:* A matrix of distances; a budget b
> *Output:* A tour which passes through all the cities and has length $\leq b$, if such a tour exists.

The optimization version of this problem asks directly for the shortest tour.

> TSP-OPT
> *Input:* A matrix of distances
> *Output:* The shortest tour which passes through all the cities.

Show that if TSP can be solved in polynomial time, then so can TSP-OPT.

8.2. *Search versus decision.* Suppose you have a procedure which runs in polynomial time and tells you whether or not a graph has a Rudrata path. Show that you can use it to develop a polynomial-time algorithm for RUDRATA PATH (which returns the actual path, if it exists).

8.3. STINGY SAT is the following problem: given a set of clauses (each a disjunction of literals) and an integer k, find a satisfying assignment in which at most k variables are `true`, if such an assignment exists. Prove that STINGY SAT is **NP**-complete.

8.4. Consider the CLIQUE problem restricted to graphs in which every vertex has degree at most 3. Call this problem CLIQUE-3.

(a) Prove that CLIQUE-3 is in **NP.**

(b) What is wrong with the following proof of **NP**-completeness for CLIQUE-3? We know that the CLIQUE problem in general graphs is **NP**-complete, so it is enough to present a reduction from CLIQUE-3 to CLIQUE. Given a graph G with vertices of degree ≤ 3, and a parameter g, the reduction leaves the graph and the parameter unchanged: clearly the output of the reduction is a possible input for the CLIQUE problem. Furthermore, the answer to both problems is identical. This proves the correctness of the reduction and, therefore, the **NP**-completeness of CLIQUE-3.

(c) It is true that the VERTEX COVER problem remains **NP**-complete even when restricted to graphs in which every vertex has degree at most 3. Call this problem VC-3. What is wrong with the following proof of **NP**-completeness for CLIQUE-3?

We present a reduction from VC-3 to CLIQUE-3. Given a graph $G = (V, E)$ with node degrees bounded by 3, and a parameter b, we create an instance of CLIQUE-3 by leaving the graph unchanged and switching the parameter to $|V| - b$. Now, a subset $C \subseteq V$ is a vertex cover in G if and only if the complementary set $V - C$ is a clique in G. Therefore G has a vertex cover of size $\leq b$ if and only if it has a clique of size $\geq |V| - b$. This proves the correctness of the reduction and, consequently, the **NP**-completeness of CLIQUE-3.

(d) Describe an $O(|V|)$ algorithm for CLIQUE-3.

8.5. Give a simple reduction from 3D MATCHING to SAT, and another from RUDRATA CYCLE to SAT. (*Hint:* In the latter case you may use variables x_{ij} whose intuitive meaning is "vertex i is the jth vertex of the Rudrata cycle"; you then need to write clauses that express the constraints of the problem.)

8.6. On page 251 we saw that 3SAT remains **NP**-complete even when restricted to formulas in which each literal appears at most twice.

(a) Show that if each literal appears at most *once*, then the problem is solvable in polynomial time.

(b) Show that INDEPENDENT SET remains **NP**-complete even in the special case when all the nodes in the graph have degree at most 4.

8.7. Consider a special case of 3SAT in which all clauses have exactly three literals, and each variable appears exactly three times. Show that this problem can be solved in polynomial time. (*Hint:* Create a bipartite graph with clauses on the left, variables on the right, and edges whenever a variable appears in a clause. Use Exercise 7.30 to show that this graph has a matching.)

8.8. In the EXACT 4SAT problem, the input is a set of clauses, each of which is a disjunction of exactly four literals, and such that each variable occurs at most once in each clause. The goal is to find a satisfying assignment, if one exists. Prove that EXACT 4SAT is NP-complete.

8.9. In the HITTING SET problem, we are given a family of sets $\{S_1, S_2, \ldots, S_n\}$ and a budget b, and we wish to find a set H of size $\leq b$ which intersects every S_i, if such an H exists. In other words, we want $H \cap S_i \neq \emptyset$ for all i.

Show that HITTING SET is **NP**-complete.

8.10. *Proving* **NP***-completeness by generalization.* For each of the problems below, prove that it is **NP**-complete by showing that it is a *generalization* of some **NP**-complete problem we have seen in this chapter.

(a) SUBGRAPH ISOMORPHISM: Given as input two undirected graphs G and H, determine whether G is a subgraph of H (that is, whether by deleting certain vertices and edges of H we obtain a graph that is, up to renaming of vertices, identical to G), and if so, return the corresponding mapping of $V(G)$ into $V(H)$.

(b) LONGEST PATH: Given a graph G and an integer g, find in G a simple path of length g.

(c) MAX SAT: Given a CNF formula and an integer g, find a truth assignment that satisfies at least g clauses.

(d) DENSE SUBGRAPH: Given a graph and two integers a and b, find a set of a vertices of G such that there are at least b edges between them.

(e) SPARSE SUBGRAPH: Given a graph and two integers a and b, find a set of a vertices of G such that there are at most b edges between them.

(f) SET COVER. (This problem generalizes *two* known **NP**-complete problems.)

(g) RELIABLE NETWORK: We are given two $n \times n$ matrices, a distance matrix d_{ij} and a *connectivity requirement* matrix r_{ij}, as well as a budget b; we must find a graph $G = (\{1, 2, \ldots, n\}, E)$ such that (1) the total cost of all edges is b or less and (2) between any two distinct vertices i and j there are r_{ij} vertex-disjoint paths. (*Hint:* Suppose that all d_{ij}'s are 1 or 2, $b = n$, and all r_{ij}'s are 2. Which well known **NP**-complete problem is this?)

8.11. There are many variants of Rudrata's problem, depending on whether the graph is undirected or directed, and whether a cycle or path is sought. Reduce the DIRECTED RUDRATA PATH problem to each of the following.

(a) The (undirected) RUDRATA PATH problem.

(b) The undirected RUDRATA (s, t)-PATH problem, which is just like RUDRATA PATH except that the endpoints of the path are specified in the input.

8.12. The k-SPANNING TREE problem is the following.

Input: An undirected graph $G = (V, E)$
Output: A spanning tree of G in which each node has degree $\leq k$, if such a tree exists.

Show that for any $k \geq 2$:

(a) k-SPANNING TREE is a search problem.

(b) k-SPANNING TREE is **NP**-complete. (*Hint:* Start with $k = 2$ and consider the relation between this problem and RUDRATA PATH.)

8.13. Determine which of the following problems are **NP**-complete and which are solvable in polynomial time. In each problem you are given an undirected graph $G = (V, E)$, along with:

(a) A set of nodes $L \subseteq V$, and you must find a spanning tree such that its set of leaves includes the set L.

(b) A set of nodes $L \subseteq V$, and you must find a spanning tree such that its set of leaves is precisely the set L.

(c) A set of nodes $L \subseteq V$, and you must find a spanning tree such that its set of leaves is included in the set L.

(d) An integer k, and you must find a spanning tree with k or fewer leaves.

(e) An integer k, and you must find a spanning tree with k or more leaves.

(f) An integer k, and you must find a spanning tree with exactly k leaves.

(*Hint:* All the **NP**-completeness proofs are by generalization, except for one.)

8.14. Prove that the following problem is **NP**-complete: given an undirected graph $G = (V, E)$ and an integer k, return a clique of size k *as well as* an independent set of size k, provided both exist.

8.15. Show that the following problem is **NP**-complete.

MAXIMUM COMMON SUBGRAPH
Input: Two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$; a budget b.
Output: Two set of nodes $V_1' \subseteq V_1$ and $V_2' \subseteq V_2$ whose deletion leaves at least b nodes in each graph, and makes the two graphs identical.

8.16. We are feeling experimental and want to create a new dish. There are various ingredients we can choose from and we'd like to use as many of them as possible, but some ingredients don't go well with others. If there are n possible ingredients (numbered 1 to n), we write down an $n \times n$ matrix giving the *discord* between any pair of ingredients. This *discord* is a real number between 0.0 and 1.0, where 0.0 means "they go together perfectly" and 1.0 means "they really don't go together." Here's an example matrix when there are five possible ingredients.

	1	2	3	4	5
1	0.0	0.4	0.2	0.9	1.0
2	0.4	0.0	0.1	1.0	0.2
3	0.2	0.1	0.0	0.8	0.5
4	0.9	1.0	0.8	0.0	0.2
5	1.0	0.2	0.5	0.2	0.0

In this case, ingredients 2 and 3 go together pretty well whereas 1 and 5 clash badly. Notice that this matrix is necessarily symmetric; and that the diagonal entries are always 0.0. Any set of ingredients incurs a *penalty* which is *the sum of all discord values between pairs of ingredients*. For instance, the set of ingredients $\{1, 3, 5\}$ incurs a penalty of $0.2 + 1.0 + 0.5 = 1.7$. We want this penalty to be small.

EXPERIMENTAL CUISINE
Input: n, the number of ingredients to choose from; D, the $n \times n$ "discord" matrix; some number $p \geq 0$
Output: The maximum number of ingredients we can choose with penalty $\leq p$.

Show that if EXPERIMENTAL CUISINE is solvable in polynomial time, then so is 3SAT.

8.17. Show that for any problem Π in **NP**, there is an algorithm which solves Π in time $O(2^{p(n)})$, where n is the size of the input instance and $p(n)$ is a polynomial (which may depend on Π).

8.18. Show that if $\mathbf{P} = \mathbf{NP}$ then the RSA cryptosystem (Section 1.4.2) can be broken in polynomial time.

8.19. A *kite* is a graph on an even number of vertices, say $2n$, in which n of the vertices form a clique and the remaining n vertices are connected in a "tail" that consists of a path joined to one of the vertices of the clique. Given a graph and a goal g, the KITE problem asks for a subgraph which is a kite and which contains $2g$ nodes. Prove that KITE is **NP**-complete.

8.20. In an undirected graph $G = (V, E)$, we say $D \subseteq V$ is a *dominating set* if every $v \in V$ is either in D or adjacent to at least one member of D. In the DOMINATING SET problem, the input is a graph and a budget b, and the aim is to find a dominating set in the graph of size at most b, if one exists. Prove that this problem is **NP**-complete.

8.21. *Sequencing by hybridization.* One experimental procedure for identifying a new DNA sequence repeatedly probes it to determine which k-mers (substrings of length k) it contains. Based on these, the full sequence must then be reconstructed.

Let's now formulate this as a combinatorial problem. For any string x (the DNA sequence), let $\Gamma(x)$ denote the multiset of all of its k-mers. In particular, $\Gamma(x)$ contains exactly $|x| - k + 1$ elements.

The reconstruction problem is now easy to state: given a multiset of k-length strings, find a string x such that $\Gamma(x)$ is exactly this multiset.

(a) Show that the reconstruction problem reduces to RUDRATA PATH. (*Hint:* Construct a directed graph with one node for each k-mer, and with an edge from a to b if the last $k-1$ characters of a match the first $k-1$ characters of b.)

(b) But in fact, there is much better news. Show that the same problem also reduces to EULER PATH. (*Hint:* This time, use one directed edge for each k-mer.)

8.22. In task scheduling, it is common to use a graph representation with a node for each task and a directed edge from task i to task j if i is a precondition for j. This directed graph depicts the precedence constraints in the scheduling problem. Clearly, a schedule is possible if and only if the graph is acyclic; if it isn't, we'd like to identify the smallest number of constraints that must be dropped so as to make it acyclic.

Given a directed graph $G = (V, E)$, a subset $E' \subseteq E$ is called a *feedback arc set* if the removal of edges E' renders G acyclic.

> FEEDBACK ARC SET (FAS): Given a directed graph $G = (V, E)$ and a budget b, find a feedback arc set of $\leq b$ edges, if one exists.

(a) Show that FAS is in **NP**.

FAS can be shown to be **NP**-complete by a reduction from VERTEX COVER. Given an instance (G, b) of VERTEX COVER, where G is an undirected graph and we want a vertex cover of size $\leq b$, we construct a instance (G', b) of FAS as follows. If $G = (V, E)$ has n vertices $v_1, \ldots, v_n$, then make $G' = (V', E')$ a directed graph with $2n$ vertices $w_1, w_1', \ldots, w_n, w_n'$, and $n + 2|E|$ (directed) edges:

- (w_i, w_i') for all $i = 1, 2, \ldots, n$.
- (w_i', w_j) and (w_j', w_i) for every $(v_i, v_j) \in E$.

(b) Show that if G contains a vertex cover of size b, then G' contains a feedback arc set of size b.

(c) Show that if G' contains a feedback arc set of size b, then G contains a vertex cover of size (at most) b. (*Hint:* Given a feedback arc set of size b in G', you may need to first modify it slightly to obtain another one which is of a more convenient form, but is of the same size or smaller. Then, argue that G must contain a vertex cover of the same size as the modified feedback arc set.)

8.23. In the NODE-DISJOINT PATHS problem, the input is an undirected graph in which some vertices have been specially marked: a certain number of "sources" $s_1, s_2, \ldots s_k$ and an equal number of "destinations" $t_1, t_2, \ldots t_k$. The goal is to find k node-disjoint paths (that is, paths which have no nodes in common) where the ith path goes from s_i to t_i. Show that this problem is **NP**-complete.

Here is a sequence of progressively stronger hints.

(i) Reduce from 3SAT.

(ii) For a 3SAT formula with m clauses and n variables, use $k = m + n$ sources and destinations. Introduce one source/destination pair (s_x, t_x) for each variable x, and one source/destination pair (s_c, t_c) for each clause c.

(iii) For each 3SAT clause, introduce 6 new intermediate vertices, one for each literal occurring in that clause and one for its complement.

(iv) Notice that if the path from s_c to t_c goes through some intermediate vertex representing, say, an occurrence of variable x, then no other path can go through that vertex. What vertex would you like the other path to be forced to go through instead?

Chapter 9

Coping with NP-completeness

You are the junior member of a seasoned project team. Your current task is to write code for solving a simple-looking problem involving graphs and numbers. What are you supposed to do?

If you are very lucky, your problem will be among the half-dozen problems concerning graphs with weights (shortest path, minimum spanning tree, maximum flow, etc.), that we have solved in this book. Even if this is the case, recognizing such a problem in its natural habitat—grungy and obscured by reality and context—requires practice and skill. It is more likely that you will need to reduce your problem to one of these lucky ones—or to solve it using dynamic programming or linear programming.

But chances are that nothing like this will happen. The world of search problems is a bleak landscape. There are a few spots of light—brilliant algorithmic ideas—each illuminating a small area around it (the problems that reduce to it; two of these areas, linear and dynamic programming, are in fact decently large). But the remaining vast expanse is pitch dark: **NP**-complete. What are you to do?

You can start by proving that your problem is actually **NP**-complete. Often a proof by generalization (recall the discussion on page 256 and Exercise 8.10) is all that you need; and sometimes a simple reduction from 3SAT or ZOE is not too difficult to find. This sounds like a theoretical exercise, but, if carried out successfully, it does bring some tangible rewards: now your status in the team has been elevated, you are no longer the kid who can't do, and you have become the noble knight with the impossible quest.

But, unfortunately, a problem does not go away when proved **NP**-complete. The real question is, *What do you do next?*

This is the subject of the present chapter and also the inspiration for some of the most important modern research on algorithms and complexity. **NP**-completeness is not a death certificate—it is only the beginning of a fascinating adventure.

Your problem's **NP**-completeness proof probably constructs graphs that are complicated and weird, very much unlike those that come up in your application. For example, even though SAT is **NP**-complete, satisfying assignments for HORN SAT (the

instances of SAT that come up in logic programming) can be found efficiently (recall Section 5.3). Or, suppose the graphs that arise in your application are trees. In this case, many **NP**-complete problems, such as INDEPENDENT SET, can be solved in linear time by dynamic programming (recall Section 6.7).

Unfortunately, this approach does not always work. For example, we know that 3SAT is **NP**-complete. And the INDEPENDENT SET problem, along with many other **NP**-complete problems, remains so even for planar graphs (graphs that can be drawn in the plane without crossing edges). Moreover, often you cannot neatly characterize the instances that come up in your application. Instead, you will have to rely on some form of *intelligent exponential search*—procedures such as *backtracking* and *branch and bound* which are exponential time in the worst-case, but, with the right design, could be very efficient on typical instances that come up in your application. We discuss these methods in Section 9.1.

Or you can develop an algorithm for your **NP**-complete optimization problem that falls short of the optimum *but never by too much*. For example, in Section 5.4 we saw that the greedy algorithm always produces a set cover that is no more than $\log n$ times the optimal set cover. An algorithm that achieves such a guarantee is called an *approximation algorithm*. As we will see in Section 9.2, such algorithms are known for many **NP**-complete optimization problems, and they are some of the most clever and sophisticated algorithms around. And the theory of **NP**-completeness can again be used as a guide in this endeavor, by showing that, for some problems, there are even severe limits to how well they can be approximated—unless of course $\mathbf{P} = \mathbf{NP}$.

Finally, there are *heuristics*, algorithms with no guarantees on either the running time or the degree of approximation. Heuristics rely on ingenuity, intuition, a good understanding of the application, meticulous experimentation, and often insights from physics or biology, to attack a problem. We see some common kinds in Section 9.3.

9.1 Intelligent exhaustive search

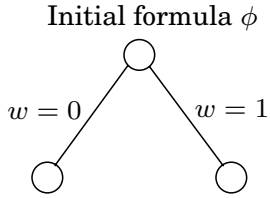
9.1.1 Backtracking

Backtracking is based on the observation that it is often possible to reject a solution by looking at just a small portion of it. For example, if an instance of SAT contains the clause $(x_1 \vee x_2)$, then all assignments with $x_1 = x_2 = 0$ (i.e., `false`) can be instantly eliminated. To put it differently, by quickly checking and discrediting this *partial assignment*, we are able to prune a quarter of the entire search space. A promising direction, but can it be systematically exploited?

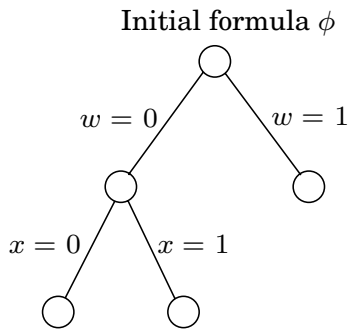
Here's how it is done. Consider the Boolean formula $\phi(w, x, y, z)$ specified by the set of clauses

$$(w \vee x \vee y \vee z),\ (w \vee \overline{x}),\ (x \vee \overline{y}),\ (y \vee \overline{z}),\ (z \vee \overline{w}),\ (\overline{w} \vee \overline{z}).$$

We will incrementally grow a tree of partial solutions. We start by branching on any one variable, say w:



Plugging $w = 0$ and $w = 1$ into ϕ, we find that no clause is immediately violated and thus neither of these two partial assignments can be eliminated outright. So we need to keep branching. We can expand either of the two available nodes, and on any variable of our choice. Let's try this one:



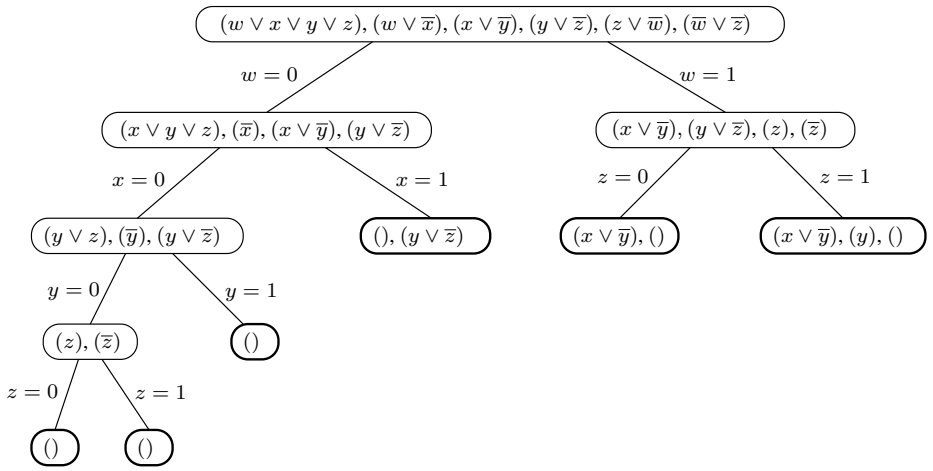
This time, we are in luck. The partial assignment $w = 0, x = 1$ violates the clause $(w \vee \overline{x})$ and can be terminated, thereby pruning a good chunk of the search space. We backtrack out of this cul-de-sac and continue our explorations at one of the two remaining active nodes.

In this manner, backtracking explores the space of assignments, growing the tree only at nodes where there is uncertainty about the outcome, and stopping if at any stage a satisfying assignment is encountered.

In the case of Boolean satisfiability, each node of the search tree can be described either by a partial assignment or by the clauses that remain when those values are plugged into the original formula. For instance, if $w = 0$ and $x = 0$ then any clause with $\overline{w}$ or $\overline{x}$ is instantly satisfied and any literal w or x is not satisfied and can be removed. What's left is

$$(y \vee z), (\overline{y}), (y \vee \overline{z}).$$

Figure 9.1 Backtracking reveals that ϕ is not satisfiable.



Likewise, $w = 0$ and $x = 1$ leaves

$$(), (y \vee \overline{z}),$$

with the "empty clause" () ruling out satisfiability. Thus the nodes of the search tree, representing partial assignments, are themselves SAT *subproblems*.

This alternative representation is helpful for making the two decisions that repeatedly arise: which subproblem to expand next, and which branching variable to use. Since the benefit of backtracking lies in its ability to eliminate portions of the search space, and since this happens only when an empty clause is encountered, it makes sense to choose the subproblem that contains the *smallest* clause and to then branch on a variable in that clause. If this clause happens to be a singleton, then at least one of the resulting branches will be terminated. (If there is a tie in choosing subproblems, one reasonable policy is to pick the one lowest in the tree, in the hope that it is close to a satisfying assignment.) See Figure 9.1 for the conclusion of our earlier example.

More abstractly, a backtracking algorithm requires a *test* that looks at a subproblem and quickly declares one of three outcomes:

1. Failure: the subproblem has no solution.
2. Success: a solution to the subproblem is found.
3. Uncertainty.

In the case of SAT, this test declares failure if there is an empty clause, success if there are no clauses, and uncertainty otherwise. The backtracking procedure then has the following format.

```
Start with some problem P_0
Let S = {P_0}, the set of active subproblems
Repeat while S is nonempty:
   choose a subproblem P ∈ S and remove it from S
   expand it into smaller subproblems P_1, P_2, ..., P_k
   For each P_i:
      If test(P_i) succeeds: halt and announce this solution
      If test(P_i) fails: discard P_i
      Otherwise: add P_i to S
Announce that there is no solution
```

For SAT, the `choose` procedure picks a clause, and `expand` picks a variable within that clause. We have already discussed some reasonable ways of making such choices.

With the right `test`, `expand`, and `choose` routines, backtracking can be remarkably effective in practice. The backtracking algorithm we showed for SAT is the basis of many successful satisfiability programs. Another sign of quality is this: if presented with a 2SAT instance, it will always find a satisfying assignment, if one exists, in polynomial time (Exercise 9.1)!

9.1.2 Branch-and-bound

The same principle can be generalized from search problems such as SAT to optimization problems. For concreteness, let's say we have a minimization problem; maximization will follow the same pattern.

As before, we will deal with partial solutions, each of which represents a *subproblem*, namely, what is the (cost of the) best way to complete this solution? And as before, we need a basis for eliminating partial solutions, since there is no other source of efficiency in our method. To reject a subproblem, we must be certain that its cost exceeds that of some other solution we have already encountered. But its exact cost is unknown to us and is generally not efficiently computable. So instead we use a quick *lower bound* on this cost.

```
Start with some problem P_0
Let S = {P_0}, the set of active subproblems
bestsofar = ∞
Repeat while S is nonempty:
  choose a subproblem P ∈ S and remove
  it from S
  expand it into smaller subproblems P_1, P_2, ..., P_k
  For each P_i:
    If P_i is a complete solution: update bestsofar
    else if lowerbound(P_i) < bestsofar: add P_i to S
return bestsofar
```

Let's see how this works for the traveling salesman problem on a graph $G = (V, E)$ with edge lengths $d_e > 0$. A partial solution is a simple path $a \rightsquigarrow b$ passing through

some vertices $S \subseteq V$, where S includes the endpoints a and b. We can denote such a partial solution by the tuple $[a, S, b]$—in fact, a will be fixed throughout the algorithm. The corresponding subproblem is to find the best completion of the tour, that is, the cheapest complementary path $b \rightsquigarrow a$ with intermediate nodes $V - S$. Notice that the initial problem is of the form $[a, \{a\}, a]$ for any $a \in V$ of our choosing.

At each step of the branch-and-bound algorithm, we extend a particular partial solution $[a, S, b]$ by a single edge (b, x), where $x \in V - S$. There can be up to $|V - S|$ ways to do this, and each of these branches leads to a subproblem of the form $[a, S \cup \{x\}, x]$.

How can we lower-bound the cost of completing a partial tour $[a, S, b]$? Many sophisticated methods have been developed for this, but let's look at a rather simple one. The remainder of the tour consists of a path through $V - S$, plus edges from a and b to $V - S$. Therefore, its cost is at least the sum of the following:

1. The lightest edge from a to $V - S$.
2. The lightest edge from b to $V - S$.
3. The minimum spanning tree of $V - S$.

(Do you see why?) And this lower bound can be computed quickly by a minimum spanning tree algorithm. Figure 9.2 runs through an example: each node of the tree represents a partial tour (specifically, the path from the root to that node) that at some stage is considered by the branch-and-bound procedure. Notice how just 28 partial solutions are considered, instead of the $7! = 5{,}040$ that would arise in a brute-force search.

9.2 Approximation algorithms

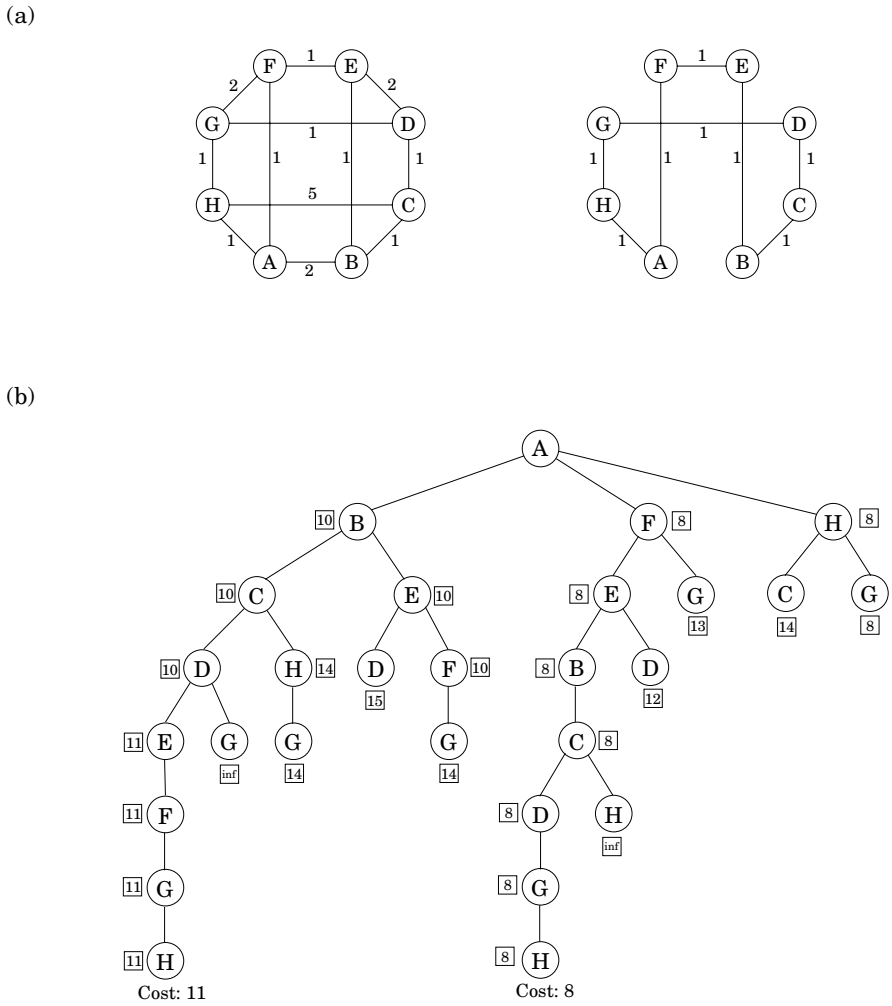
In an optimization problem we are given an instance I and are asked to find the optimum solution—the one with the maximum gain if we have a maximization problem like INDEPENDENT SET, or the minimum cost if we are dealing with a minimization problem such as the TSP. For every instance I, let us denote by $\text{OPT}(I)$ the value (benefit or cost) of the optimum solution. It makes the math a little simpler (and is not too far from the truth) to *assume that* $\text{OPT}(I)$ *is always a positive integer.*

We have already seen an example of a (famous) approximation algorithm in Section 5.4: the greedy scheme for SET COVER. For any instance I of size n, we showed that this greedy algorithm is guaranteed to quickly find a set cover of cardinality at most $\text{OPT}(I) \log n$. This $\log n$ factor is known as the approximation guarantee of the algorithm.

More generally, consider any minimization problem. Suppose now that we have an algorithm $\mathcal{A}$ for our problem which, given an instance I, returns a solution with value $\mathcal{A}(I)$. The *approximation ratio* of algorithm $\mathcal{A}$ is defined to be

$$\alpha_{\mathcal{A}} = \max_I \frac{\mathcal{A}(I)}{\text{OPT}(I)}.$$

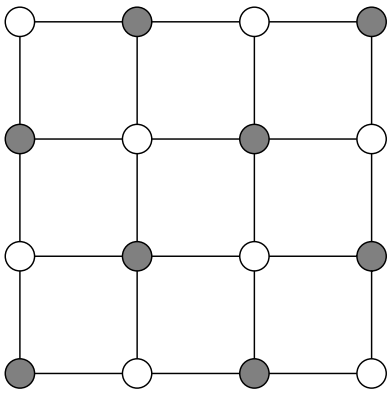
Figure 9.2 (a) A graph and its optimal traveling salesman tour. (b) The branch-and-bound search tree, explored left to right. Boxed numbers indicate lower bounds on cost.



In other words, $\alpha_{\mathcal{A}}$ measures by the factor by which the output of algorithm $\mathcal{A}$ exceeds the optimal solution, on the worst-case input. The approximation ratio can also be defined for maximization problems, such as INDEPENDENT SET, in the same way—except that to get a number larger than 1 we take the reciprocal.

So, when faced with an **NP**-complete optimization problem, a reasonable goal is to look for an approximation algorithm $\mathcal{A}$ whose $\alpha_{\mathcal{A}}$ is as small as possible. But this kind of guarantee might seem a little puzzling: How can we come close to the optimum if we cannot determine the optimum? Let's look at a simple example.

Figure 9.3 A graph whose optimal vertex cover, shown shaded, is of size 8.



9.2.1 Vertex cover

We already know the VERTEX COVER problem is **NP**-hard.

Vertex Cover

Input: An undirected graph $G = (V, E)$.

Output: A subset of the vertices $S \subseteq V$ that touches every edge.

Goal: Minimize $|S|$.

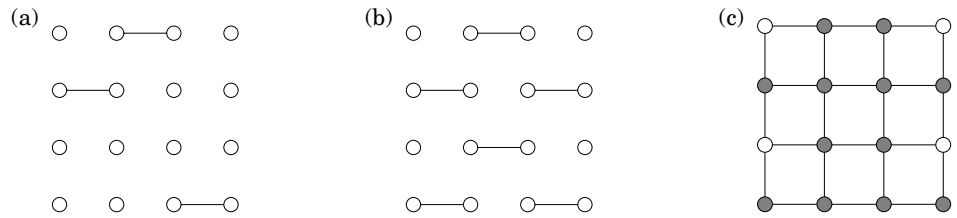
See Figure 9.3 for an example.

Since VERTEX COVER is a special case of SET COVER, we know from Chapter 5 that it can be approximated within a factor of $O(\log n)$ by the greedy algorithm: repeatedly delete the vertex of highest degree and include it in the vertex cover. And there are graphs on which the greedy algorithm returns a vertex cover that is indeed $\log n$ times the optimum.

A better approximation algorithm for VERTEX COVER is based on the notion of a *matching*, a subset of edges that have no vertices in common (Figure 9.4). A matching is *maximal* if no more edges can be added to it. Maximal matchings will help us find good vertex covers, and moreover, they are easy to generate: repeatedly pick edges that are disjoint from the ones chosen already, until this is no longer possible.

What is the relationship between matchings and vertex covers? Here is the crucial fact: any vertex cover of a graph G must be at least as large as the number of edges in any matching in G; that is, *any matching provides a lower bound on* OPT. This is simply because each edge of the matching must be covered by one of its endpoints in any vertex cover! Finding such a lower bound is a key step in designing an approximation algorithm, because we must compare the quality of the solution found by our algorithm to OPT, which is **NP**-complete to compute.

Figure 9.4 (a) A matching, (b) its completion to a maximal matching, and (c) the resulting vertex cover.



One more observation completes the design of our approximation algorithm: let S be a set that contains both endpoints of each edge in a maximal matching M. Then S must be a vertex cover—if it isn't, that is, if it doesn't touch some edge $e \in E$, then M could not possibly be maximal since we could still add e to it. But our cover S has $2|M|$ vertices. And from the previous paragraph we know that *any* vertex cover must have size at least $|M|$. So we're done.

Here's the algorithm for VERTEX COVER.

```
Find a maximal matching M ⊆ E
Return S = {all endpoints of edges in M}
```

This simple procedure always returns a vertex cover whose size is at most twice optimal!

In summary, even though we have no way of finding the best vertex cover, we can easily find another structure, a maximal matching, with two key properties:

1. Its size gives us a lower bound on the optimal vertex cover.
2. It can be used to build a vertex cover, whose size can be related to that of the optimal cover using property 1.

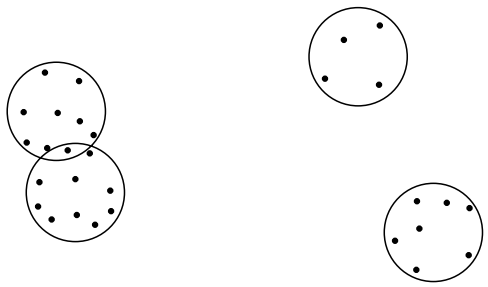
Thus, this simple algorithm has an approximation ratio of $\alpha_{\mathcal{A}} \leq 2$. In fact, it is not hard to find examples on which it does make a 100% error; hence $\alpha_{\mathcal{A}} = 2$.

9.2.2 Clustering

We turn next to a *clustering* problem, in which we have some data (text documents, say, or images, or speech samples) that we want to divide into groups. It is often useful to define "distances" between these data points, numbers that capture how close or far they are from one another. Often the data are true points in some high-dimensional space and the distances are the usual Euclidean distance; in other cases, the distances are the result of some "similarity tests" to which we have subjected the data points. Assume that we have such distances and that they satisfy the usual *metric* properties:

1. $d(x, y) \geq 0$ for all x, y.

Figure 9.5 Some data points and the optimal $k = 4$ clusters.



2. $d(x, y) = 0$ if and only if $x = y$.
3. $d(x, y) = d(y, x)$.
4. (Triangle inequality) $d(x, y) \leq d(x, z) + d(z, y)$.

We would like to partition the data points into groups that are compact in the sense of having small diameter.

k-Cluster

Input: Points $X = \{x_1, \ldots, x_n\}$ with underlying distance metric $d(\cdot, \cdot)$; integer k.

Output: A partition of the points into k clusters $C_1, \ldots, C_k$.

Goal: Minimize the diameter of the clusters,

$$\max_j \max_{x_a, x_b \in C_j} d(x_a, x_b).$$

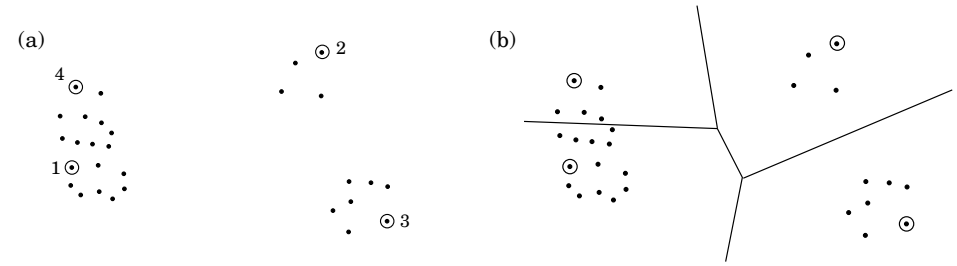
One way to visualize this task is to imagine n points in space, which are to be covered by k spheres of equal size. What is the smallest possible diameter of the spheres? Figure 9.5 shows an example.

This problem is **NP**-hard, but has a very simple approximation algorithm. The idea is to pick k of the data points as *cluster centers* and to then assign each of the remaining points to the center closest to it, thus creating k clusters. The centers are picked one at a time, using an intuitive rule: always pick the next center to be as far as possible from the centers chosen so far (see Figure 9.6).

```
Pick any point μ_1 ∈ X as the first cluster center
for i = 2 to k:
  Let μ_i be the point in X farthest from μ_1, ..., μ_{i-1}
  (i.e., that maximizes min_{j<i} d(·, μ_j))
Create k clusters: C_i = {all x ∈ X whose closest center is μ_i}
```

It's clear that this algorithm returns a valid partition. What's more, the resulting diameter is guaranteed to be at most twice optimal.

Figure 9.6 (a) Four centers chosen by farthest-first traversal. (b) The resulting clusters.



Here's the argument. Let $x \in X$ be the point farthest from $\mu_1, \ldots, \mu_k$ (in other words the next center we would have chosen, if we wanted $k+1$ of them), and let r be its distance to its closest center. Then every point in X must be within distance r of its cluster center. By the triangle inequality, this means that every cluster has diameter at most $2r$.

But how does r relate to the diameter of the optimal clustering? Well, we have identified $k+1$ points $\{\mu_1, \mu_2, \ldots, \mu_k, x\}$ that are all at a distance at least r from each other (why?). *Any* partition into k clusters must put two of these points in the same cluster and must therefore have diameter at least r.

This algorithm has a certain high-level similarity to our scheme for VERTEX COVER. Instead of a maximal matching, we use a different easily computable structure—a set of k points that cover all of X within some radius r, while at the same time being mutually separated by a distance of at least r. This structure is used both to generate a clustering and to give a lower bound on the optimal clustering.

We know of no better approximation algorithm for this problem.

9.2.3 TSP

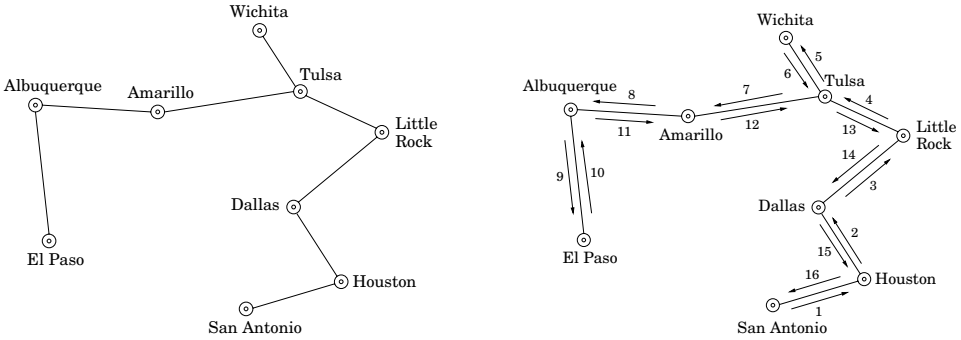
The triangle inequality played a crucial role in making the k-CLUSTER problem approximable. It also helps with the TRAVELING SALESMAN PROBLEM: if the distances between cities satisfy the metric properties, then there is an algorithm that outputs a tour of length at most 1.5 times optimal. We'll now look at a slightly weaker result that achieves a factor of 2.

Continuing with the thought processes of our previous two approximation algorithms, we can ask whether there is some structure that is easy to compute and that is plausibly related to the best traveling salesman tour (as well as providing a good lower bound on OPT). A little thought and experimentation reveals the answer to be the *minimum spanning tree*.

Let's understand this relation. Removing any edge from a traveling salesman tour leaves a path through all the vertices, which is a spanning tree. Therefore,

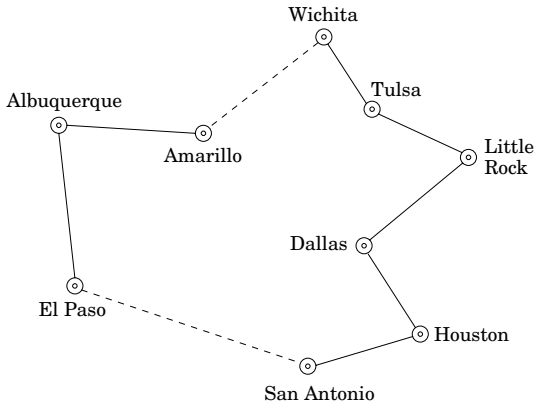
$$\text{TSP cost} \;\geq\; \text{cost of this path} \;\geq\; \text{MST cost}.$$

Now, we somehow need to use the MST to build a traveling salesman tour. If we can use each edge *twice*, then by following the shape of the MST we end up with a tour that visits all the cities, some of them more than once. Here's an example, with the MST on the left and the resulting tour on the right (the numbers show the order in which the edges are taken).



Therefore, this tour has a length at most twice the MST cost, which as we've already seen is at most twice the TSP cost.

This is the result we wanted, but we aren't quite done because our tour visits some cities multiple times and is therefore not legal. To fix the problem, the tour should simply skip any city it is about to revisit, and instead move directly to the next *new* city in its list:



By the triangle inequality, these bypasses can only make the overall tour shorter.

General TSP

But what if we are interested in instances of TSP that do not satisfy the triangle inequality? It turns out that this is a *much* harder problem to approximate.

Here is why: Recall that on page 260 we gave a polynomial-time reduction which given any graph G and any integer $C > 0$ produces an instance $I(G, C)$ of the TSP such that:

(i) If G has a Rudrata path, then $\text{OPT}(I(G, C)) = n$, the number of vertices in G.

(ii) If G has no Rudrata path, then $\text{OPT}(I(G, C)) \geq n + C$.

This means that even an approximate solution to TSP would enable us to solve RUDRATA PATH! Let's work out the details.

Consider an approximation algorithm $\mathcal{A}$ for TSP and let $\alpha_{\mathcal{A}}$ denote its approximation ratio. From any instance G of RUDRATA PATH, we will create an instance $I(G, C)$ of TSP using the specific constant $C = n\alpha_{\mathcal{A}}$. What happens when algorithm $\mathcal{A}$ is run on this TSP instance? In case (i), it must output a tour of length at most $\alpha_{\mathcal{A}}\text{OPT}(I(G, C)) = n\alpha_{\mathcal{A}}$, whereas in case (ii) it must output a tour of length at least $\text{OPT}(I(G, C)) > n\alpha_{\mathcal{A}}$. Thus we can figure out whether G has a Rudrata path! Here is the resulting procedure:

```
Given any graph G:
   compute I(G,C) (with C = n·α_A) and run algorithm A on it
   if the resulting tour has length ≤ nα_A:
      conclude that G has a Rudrata path
   else: conclude that G has no Rudrata path
```

This tells us whether or not G has a Rudrata path; by calling the procedure a polynomial number of times, we can find the actual path (Exercise 8.2).

We've shown that if TSP has a polynomial-time approximation algorithm, then there is a polynomial algorithm for the **NP**-complete RUDRATA PATH problem. So, unless **P** = **NP**, there cannot exist an efficient approximation algorithm for the TSP.

9.2.4 Knapsack

Our last approximation algorithm is for a maximization problem and has a very impressive guarantee: given any $\epsilon > 0$, it will return a solution of value at least $(1 - \epsilon)$ times the optimal value, in time that scales only polynomially in the input size and in $1/\epsilon$.

The problem is KNAPSACK, which we first encountered in Chapter 6. There are n items, with weights $w_1, \ldots, w_n$ and values $v_1, \ldots, v_n$ (all positive integers), and the goal is to pick the most valuable combination of items subject to the constraint that their total weight is at most W.

Earlier we saw a dynamic programming solution to this problem with running time $O(nW)$. Using a similar technique, a running time of $O(nV)$ can also be achieved, where V is the sum of the values. Neither of these running times is polynomial, because W and V can be very large, exponential in the size of the input.

Let's consider the $O(nV)$ algorithm. In the bad case when V is large, what if we simply scale down all the values in some way? For instance, if

$$v_1 = 117{,}586{,}003,\ v_2 = 738{,}493{,}291,\ v_3 = 238{,}827{,}453,$$

we could simply knock off some precision and instead use 117, 738, and 238. This doesn't change the problem all that much and will make the algorithm much, much faster!

Now for the details. Along with the input, the user is assumed to have specified some approximation factor $\epsilon > 0$.

```
Discard any item with weight > W
Let v_max = max_i v_i
Rescale values v̂_i = ⌊v_i · n/(ε v_max)⌋
Run the dynamic programming algorithm with values {v̂_i}
Output the resulting choice of items
```

Let's see why this works. First of all, since the rescaled values $\widehat{v}_i$ are all at most n/ϵ, the dynamic program is efficient, running in time $O(n^3/\epsilon)$.

Now suppose the optimal solution to the original problem is to pick some subset of items S, with total value K^*. The rescaled value of this same assignment is

$$\sum_{i\in S}\widehat{v}_i \;=\; \sum_{i\in S}\left\lfloor v_i\cdot\frac{n}{\epsilon v_{\max}}\right\rfloor \;\geq\; \sum_{i\in S}\left(v_i\cdot\frac{n}{\epsilon v_{\max}}-1\right) \;\geq\; K^*\cdot\frac{n}{\epsilon v_{\max}}-n.$$

Therefore, the optimal assignment for the shrunken problem, call it $\widehat{S}$, has a rescaled value of at least this much. In terms of the original values, assignment $\widehat{S}$ has a value of at least

$$\sum_{i\in\widehat{S}}v_i \;\geq\; \sum_{i\in\widehat{S}}\widehat{v}_i\cdot\frac{\epsilon v_{\max}}{n} \;\geq\; \left(K^*\cdot\frac{n}{\epsilon v_{\max}}-n\right)\cdot\frac{\epsilon v_{\max}}{n} \;=\; K^*-\epsilon v_{\max} \;\geq\; K^*(1-\epsilon).$$

9.2.5 The approximability hierarchy

Given any **NP**-complete optimization problem, we seek the best approximation algorithm possible. Failing this, we try to prove *lower bounds* on the approximation ratios that are achievable in polynomial time (we just carried out such a proof for the general TSP). All told, **NP**-complete optimization problems are classified as follows:

- Those for which, like the TSP, no finite approximation ratio is possible.
- Those for which an approximation ratio is possible, but there are limits to how small this can be. VERTEX COVER, k-CLUSTER, and the TSP with triangle inequality belong here. (For these problems we have not established limits to their approximability, but these limits do exist, and their proofs constitute some of the most sophisticated results in this field.)
- Down below we have a more fortunate class of **NP**-complete problems for which approximability has no limits, and polynomial approximation algorithms with error ratios arbitrarily close to zero exist. KNAPSACK resides here.

- Finally, there is another class of problems, between the first two given here, for which the approximation ratio is about $\log n$. SET COVER is an example.

(A humbling reminder: All this is contingent upon the assumption **P** $\neq$ **NP.** Failing this, this hierarchy collapses down to **P,** and all **NP**-complete optimization problems can be solved exactly in polynomial time.)

A final point on approximation algorithms: often these algorithms, or their variants, perform much better on typical instances than their worst-case approximation ratio would have you believe.

9.3 Local search heuristics

Our next strategy for coping with **NP**-completeness is inspired by evolution (which is, after all, the world's best-tested optimization procedure)—by its incremental process of introducing small mutations, trying them out, and keeping them if they work well. This paradigm is called *local search* and can be applied to any optimization task. Here's how it looks for a minimization problem.

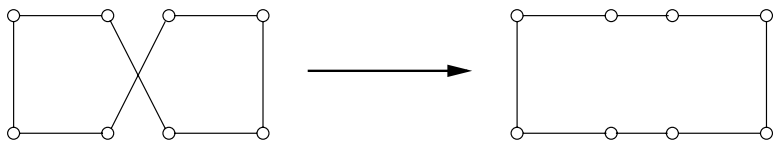
```
let s be any initial solution
while there is some solution s' in the neighborhood of s
   for which cost(s') < cost(s): replace s by s'
return s
```

On each iteration, the current solution is replaced by a better one close to it, in its *neighborhood*. This neighborhood structure is something we impose upon the problem and is the central design decision in local search. As an illustration, let's revisit the traveling salesman problem.

9.3.1 Traveling salesman, once more

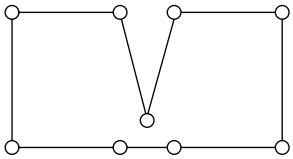
Assume we have all interpoint distances between n cities, giving a search space of $(n-1)!$ different tours. What is a good notion of neighborhood?

The most obvious notion is to consider two tours as being close if they differ in just a few edges. They can't differ in just one edge (do you see why?), so we will consider differences of two edges. We define the *2-change* neighborhood of tour s as being the set of tours that can be obtained by removing two edges of s and then putting in two other edges. Here's an example of a local move:

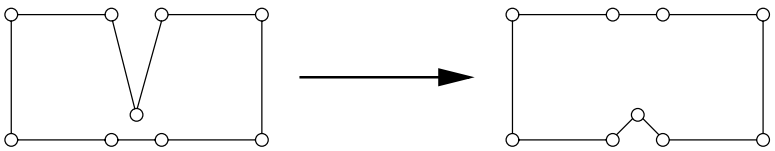


We now have a well-defined local search procedure. How does it measure up under our two standard criteria for algorithms—what is its overall running time, and does it always return the best solution?

Embarrassingly, neither of these questions has a satisfactory answer. Each iteration is certainly fast, because a tour has only $O(n^2)$ neighbors. However, it is not clear how many iterations will be needed: whether for instance, there might be an exponential number of them. Likewise, all we can easily say about the final tour is that it is *locally optimal*—that is, it is superior to the tours in its immediate neighborhood. There might be better solutions further away. For instance, the following picture shows a possible final answer that is clearly suboptimal; the range of local moves is simply too limited to improve upon it.



To overcome this, we may try a more generous neighborhood, for instance *3-change*, consisting of tours that differ on up to three edges. And indeed, the preceding bad case gets fixed:



But there is a downside, in that the size of a neighborhood becomes $O(n^3)$, making each iteration more expensive. Moreover, there may still be suboptimal local minima, although fewer than before. To avoid these, we would have to go up to *4-change*, or higher. In this manner, efficiency and quality often turn out to be competing considerations in a local search. Efficiency demands neighborhoods that can be searched quickly, but smaller neighborhoods can increase the abundance of low-quality local optima. The appropriate compromise is typically determined by experimentation.

Figure 9.7 shows a specific example of local search at work. Figure 9.8 is a more abstract, stylized depiction of local search. The solutions crowd the unshaded area, and cost decreases when we move downward. Starting from an initial solution, the algorithm moves downhill until a local optimum is reached.

In general, the search space might be riddled with local optima, and some of them may be of very poor quality. The hope is that with a judicious choice of neighborhood structure, most local optima will be reasonable. Whether this is the reality or merely

Figure 9.7 (a) Nine American cities. (b) Local search, starting at a random tour, and using 3-change. The traveling salesman tour is found after three moves.

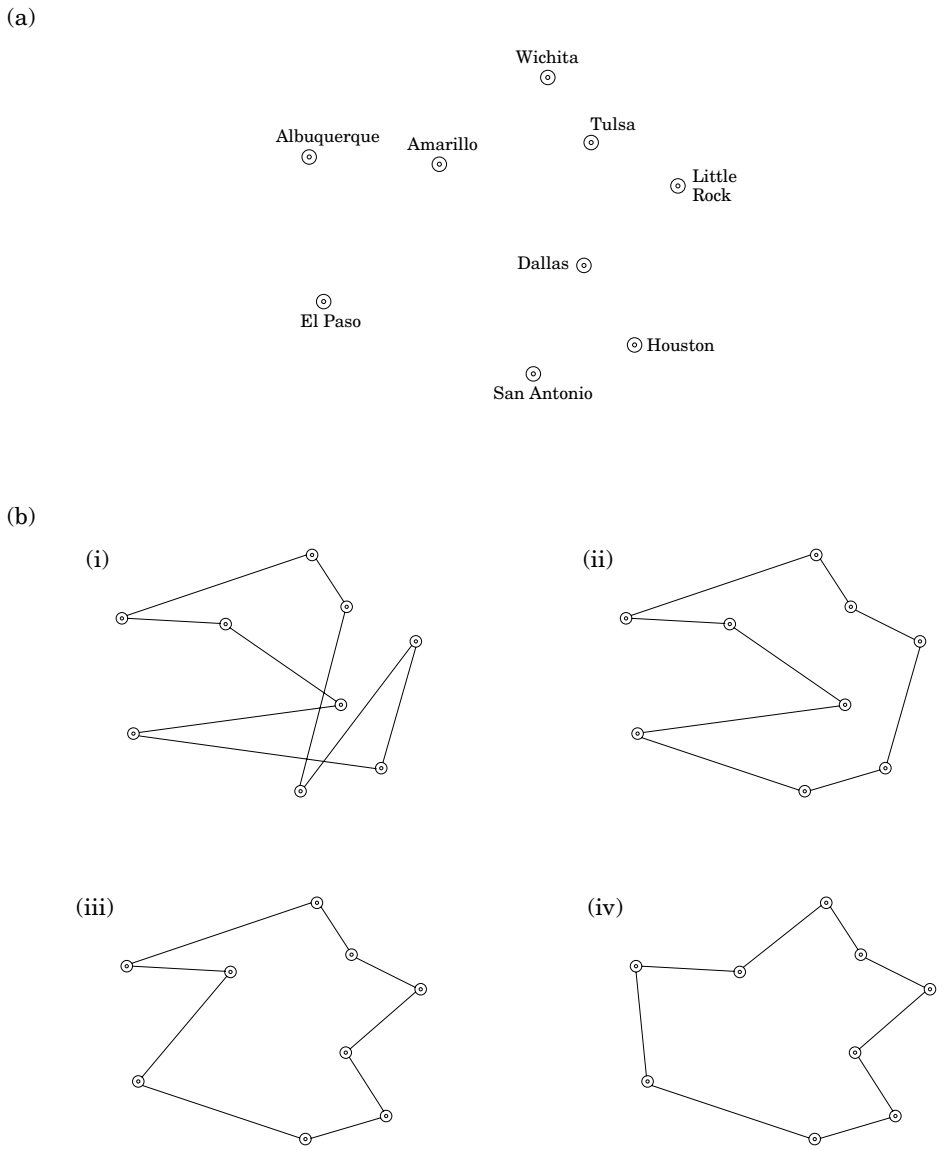
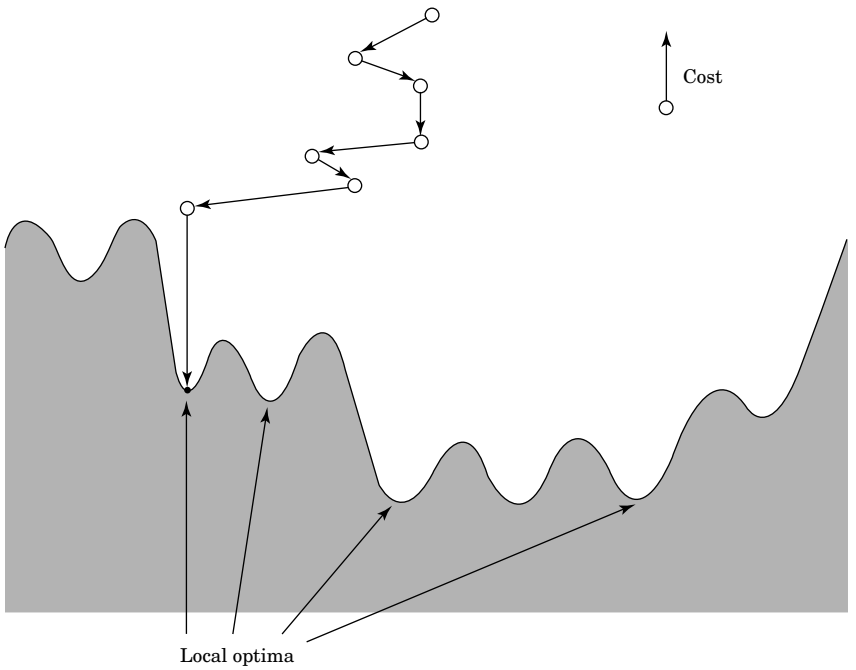


Figure 9.8 Local search.



misplaced faith, it is an empirical fact that local search algorithms are the top performers on a broad range of optimization problems. Let's look at another such example.

9.3.2 Graph partitioning

The problem of graph partitioning arises in a diversity of applications, from circuit layout to program analysis to image segmentation. We saw a special case of it, BALANCED CUT, in Chapter 8.

Graph Partitioning

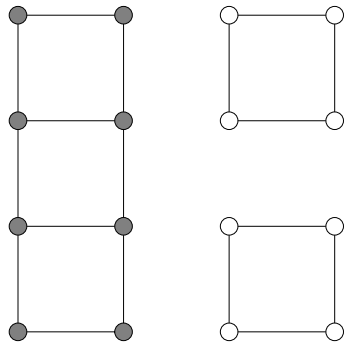
Input: An undirected graph $G = (V, E)$ with nonnegative edge weights; a real number $\alpha \in (0, 1/2]$.

Output: A partition of the vertices into two groups A and B, each of size at least $\alpha|V|$.

Goal: Minimize the capacity of the cut (A, B).

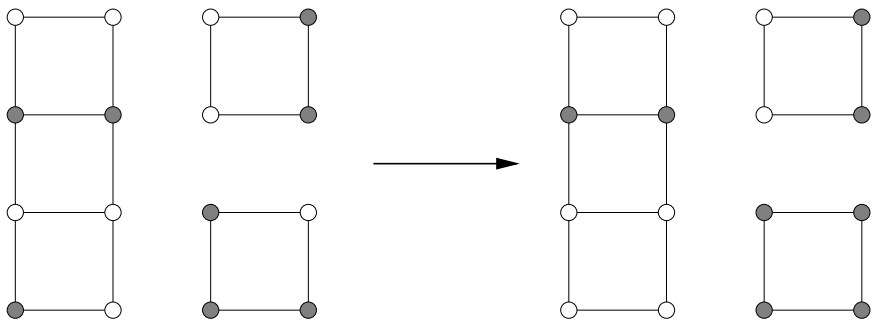
Figure 9.9 shows an example in which the graph has 16 nodes, all edge weights are 0 or 1, and the optimal solution has cost 0. Removing the restriction on the sizes of A and B would give the MINIMUM CUT problem, which we know to be efficiently solvable

Figure 9.9 An instance of GRAPH PARTITIONING, with the optimal partition for $\alpha = 1/2$. Vertices on one side of the cut are shaded.

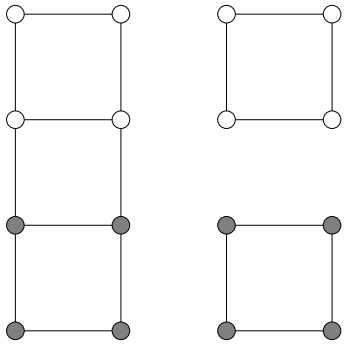


using flow techniques. The present variant, however, is **NP**-hard. In designing a local search algorithm, it will be a big convenience to focus on the special case $\alpha = 1/2$, in which A and B are forced to contain exactly half the vertices. The apparent loss of generality is purely cosmetic, as GRAPH PARTITIONING reduces to this particular case.

We need to decide upon a neighborhood structure for our problem, and there is one obvious way to do this. Let (A, B), with $|A| = |B|$, be a candidate solution; we will define its neighbors to be all solutions obtainable by swapping one pair of vertices across the cut, that is, all solutions of the form $(A - \{a\} + \{b\}, B - \{b\} + \{a\})$ where $a \in A$ and $b \in B$. Here's an example of a local move:



We now have a reasonable local search procedure, and we could just stop here. But there is still a lot of room for improvement in terms of the *quality* of the solutions produced. The search space includes some local optima that are quite far from the global solution. Here's one which has cost 2.



What can be done about such suboptimal solutions? We could expand the neighborhood size to allow two swaps at a time, but this particular bad instance would still stubbornly resist. Instead, let's look at some other generic schemes for improving local search procedures.

9.3.3 Dealing with local optima

Randomization and restarts

Randomization can be an invaluable ally in local search. It is typically used in two ways: to pick a random initial solution, for instance a random graph partition; and to choose a local move when several are available.

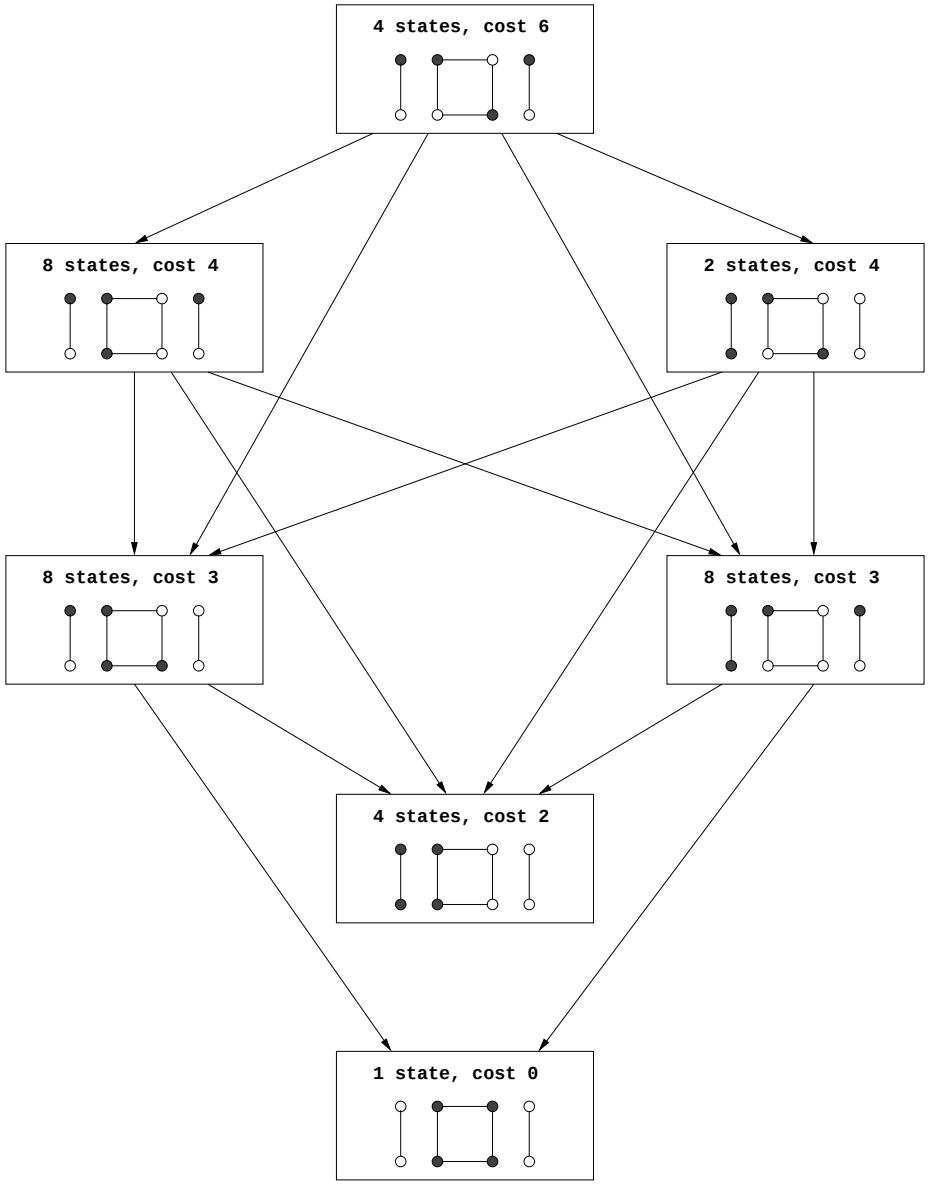
When there are many local optima, randomization is a way of making sure that there is at least some probability of getting to the right one. The local search can then be repeated several times, with a different random seed on each invocation, and the best solution returned. If the probability of reaching a good local optimum on any given run is p, then within $O(1/p)$ runs such a solution is likely to be found (recall Exercise 1.34).

Figure 9.10 shows a small instance of graph partitioning, along with the search space of solutions. There are a total of $\binom{8}{4} = 70$ possible states, but since each of them has an identical twin in which the left and right sides of the cut are flipped, in effect there are just 35 solutions. In the figure, these are organized into seven groups for readability. There are five local optima, of which four are bad, with cost 2, and one is good, with cost 0. If local search is started at a random solution, and at each step a random neighbor of lower cost is selected, then the search is at most four times as likely to wind up in a bad solution than a good one. Thus only a small handful of repetitions is needed.

Simulated annealing

In the example of Figure 9.10, each run of local search has a reasonable chance of finding the global optimum. This isn't always true. As the problem size grows, the ratio of bad to good local optima often increases, sometimes to the point of being exponentially large. In such cases, simply repeating the local search a few times is ineffective.

Figure 9.10 The search space for a graph with eight nodes. The space contains 35 solutions, which have been partitioned into seven groups for clarity. An example of each is shown. There are five local optima.



A different avenue of attack is to occasionally allow moves that actually increase the cost, in the hope that they will pull the search out of dead ends. This would be very useful at the bad local optima of Figure 9.10, for instance. The method of *simulated annealing* redefines the local search by introducing the notion of a *temperature* T.

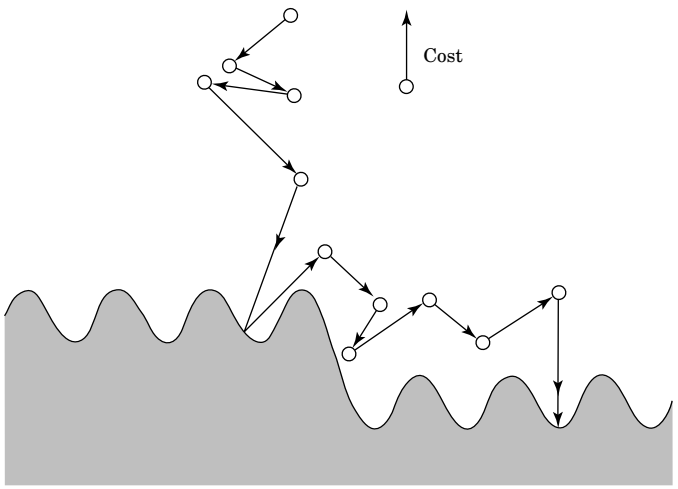
```
let s be any starting solution
repeat
   randomly choose a solution s' in the neighborhood of s
   if Δ = cost(s') − cost(s) is negative:
      replace s by s'
   else:
      replace s by s' with probability e^(−Δ/T).
```

If T is zero, this is identical to our previous local search. But if T is large, then moves that increase the cost are occasionally accepted. What value of T should be used?

The trick is to start with T large and then gradually reduce it to zero. Thus initially, the local search can wander around quite freely, with only a mild preference for low-cost solutions. As time goes on, this preference becomes stronger, and the system mostly sticks to the lower-cost region of the search space, with occasional excursions out of it to escape local optima. Eventually, when the temperature drops further, the system converges on a solution. Figure 9.11 shows this process schematically.

Simulated annealing is inspired by the physics of crystallization. When a substance is to be crystallized, it starts in liquid state, with its particles in relatively unconstrained motion. Then it is slowly cooled, and as this happens, the particles gradually

Figure 9.11 Simulated annealing.



move into more regular configurations. This regularity becomes more and more pronounced until finally a crystal lattice is formed.

The benefits of simulated annealing come at a significant cost: because of the changing temperature and the initial freedom of movement, many more local moves are needed until convergence. Moreover, it is quite an art to choose a good timetable by which to decrease the temperature, called an *annealing schedule*. But in many cases where the quality of solutions improves significantly, the tradeoff is worthwhile.

Exercises

9.1. In the backtracking algorithm for SAT, suppose that we always `choose` a subproblem (CNF formula) that has a clause that is as small as possible; and we `expand` it along a variable that appears in this small clause. Show that if the input formula only contains clauses with two literals (that is, it is an instance of 2SAT), then a satisfying assignment, if one exists, will be found in polynomial time.

9.2. Devise a backtracking algorithm for the RUDRATA PATH problem from a fixed vertex s. To fully specify such an algorithm you must define:

(a) What is a subproblem?

(b) How to `choose` a subproblem.

(c) How to `expand` a subproblem.

Argue briefly why your choices are reasonable.

9.3. Devise a branch-and-bound algorithm for the SET COVER problem. This entails deciding:

(a) What is a subproblem?

(b) How do you `choose` a subproblem to expand?

(c) How do you `expand` a subproblem?

(d) What is an appropriate `lowerbound`?

Do you think that your choices above will work well on typical instances of the problem? Why?

9.4. Given an undirected graph $G = (V, E)$ in which each node has degree $\leq d$, show how to efficiently find an independent set whose size is at least $1/(d+1)$ times that of the largest independent set.

9.5. *Local search for minimum spanning trees.* Consider the set of all spanning trees (not just minimum ones) of a weighted, connected, undirected graph $G = (V, E)$.

Recall from Section 5.1 that adding an edge e to a spanning tree T creates an unique cycle, and subsequently removing any other edge $e' \neq e$ from this cycle gives back a different spanning tree T'. We will say that T and T' differ by a single *edge swap* (e, e') and that they are *neighbors*.

(a) Show that it is possible to move from any spanning tree T to any other spanning tree T' by performing a series of edge-swaps, that is, by moving from neighbor to neighbor. At most how many edge-swaps are needed?

(b) Show that if T' is an MST, then it is possible to choose these swaps so that the costs of the spanning trees encountered along the way are nonincreasing. In other words, if the sequence of spanning trees encountered is

$$T = T_0 \to T_1 \to T_2 \to \cdots \to T_k = T',$$

then $\text{cost}(T_{i+1}) \leq \text{cost}(T_i)$ for all $i < k$.

(c) Consider the following local search algorithm which is given as input an undirected graph with distinct edge weights.

```
Let T be any spanning tree of G
while there is an edge-swap (e, e') which reduces
cost(T):
    T ← T + e − e'
return T
```

Show that this procedure always returns a minimum spanning tree. At most how many iterations does it take?

9.6. In the MINIMUM STEINER TREE problem, the input consists of: a complete graph $G = (V, E)$ with distances d_{uv} between all pairs of nodes; and a distinguished set of *terminal nodes* $V' \subseteq V$. The goal is to find a minimum-cost tree that includes the vertices V'. This tree may or may not include nodes in $V - V'$.

Suppose the distances in the input are a metric (recall the definition on page 279). Show that an efficient ratio-2 approximation algorithm for MINIMUM STEINER TREE can be obtained by ignoring the nonterminal nodes and simply returning the minimum spanning tree on V'. (*Hint:* Recall our approximation algorithm for the TSP.)

9.7. In the MULTIWAY CUT problem, the input is an undirected graph $G = (V, E)$ and a set of terminal nodes $s_1, s_2, \ldots, s_k \in V$. The goal is to find the minimum set of edges in E whose removal leaves all terminals in different components.

(a) Show that this problem can be solved exactly in polynomial time when $k = 2$.

(b) Give an approximation algorithm with ratio at most 2 for the case $k = 3$.

(c) Design a local search algorithm for multiway cut.

9.8. In the MAX SAT problem, we are given a set of clauses, and we want to find an assignment that satisfies as many of them as possible.

(a) Show that if this problem can be solved in polynomial time, then so can SAT.

(b) Here's a very naive algorithm.

```
for each variable:
  set its value to either 0 or 1 by flipping a coin
```

Suppose the input has m clauses, of which the jth has k_j literals. Show that the *expected* number of clauses satisfied by this simple algorithm is

$$\sum_{j=1}^{m}\left(1-\frac{1}{2^{k_j}}\right)\geq\frac{m}{2}.$$

In other words, this is a 2-approximation in expectation! And if the clauses all contain k literals, then this approximation factor improves to $1+1/(2^k-1)$.

(c) Can you make this algorithm deterministic? (*Hint:* Instead of flipping a coin for each variable, select the value that satisfies the most as-yet-unsatisfied clauses. What fraction of the clauses is satisfied in the end?)

9.9. In the MAXIMUM CUT problem we are given an undirected graph $G=(V,E)$ with a weight $w(e)$ on each edge, and we wish to separate the vertices into two sets S and $V-S$ so that the total weight of the edges between the two sets is as *large* as possible.

For each $S\subseteq V$, define $w(S)$ to be the sum of all w_{uv} over all edges $\{u,v\}$ such that $|S\cap\{u,v\}|=1$. Obviously, MAX CUT is about maximizing $w(S)$ over all subsets of V.

Consider the following local search algorithm for MAX CUT:

```
start with any S ⊆ V
while there is a subset S' ⊆ V such that
  |(S' − S) ∪ (S − S')| = 1 and w(S') > w(S) do:
    set S = S'
```

(a) Show that this is an approximation algorithm for MAX CUT with ratio 2.

(b) But is it a polynomial-time algorithm?

9.10. Let us call a local search algorithm *exact* when it always produces the optimum solution. For example, the local search algorithm for the minimum spanning tree problem introduced in Problem 9.5 is exact. For another example, simplex can be considered an exact local search algorithm for linear programming.

(a) Show that the 2-change local search algorithm for the TSP is not exact.

(b) Repeat for the $\lceil\frac{n}{2}\rceil$-change local search algorithm, where n is the number of cities.

(c) Show that the $(n-1)$-change local search algorithm is exact.

(d) If A is an optimization problem, define A-IMPROVEMENT to be the following search problem: Given an instance x of A and a solution s of A, find

another solution of x with better cost (or report that none exists, and thus s is optimum). For example, in TSP IMPROVEMENT we are given a distance matrix and a tour, and we are asked to find a better tour. It turns out that TSP IMPROVEMENT is **NP**-complete, and so is SET COVER IMPROVEMENT. Prove the latter.

(e) We say that a local search algorithm has *polynomial iteration* if each execution of the loop requires polynomial time. For example, the obvious implementations of the $(n-1)$-change local search algorithm for the TSP defined above do not have polynomial iteration. Show that, unless **P = NP,** there is no exact local search algorithm with polynomial iteration for the TSP and SET COVER problems.

Chapter 10

Quantum algorithms

This book started with the world's oldest and most widely used algorithms (the ones for adding and multiplying numbers) and an ancient hard problem (FACTORING). In this last chapter the tables are turned: we present one of the latest algorithms—and it is an efficient algorithm for FACTORING!

There is a catch, of course: this algorithm needs a *quantum computer* to execute.

Quantum physics is a beautiful and mysterious theory that describes Nature in the small, at the level of elementary particles. One of the major discoveries of the nineties was that quantum computers—computers based on quantum physics principles—are radically different from those that operate according to the more familiar principles of classical physics. Surprisingly, they can be exponentially more powerful: as we shall see, quantum computers can solve FACTORING in polynomial time! As a result, in a world with quantum computers, the systems that currently safeguard business transactions on the Internet (and are based on the RSA cryptosystem) will no longer be secure.

10.1 Qubits, superposition, and measurement

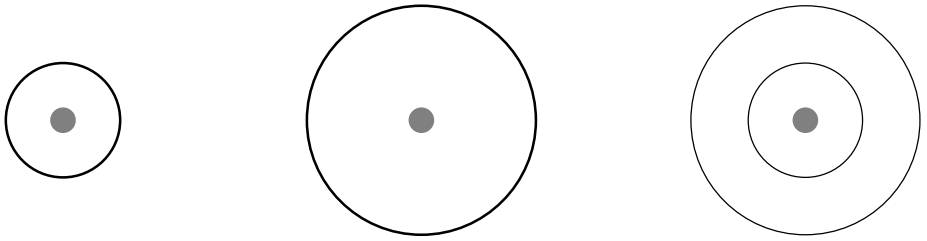
In this section we introduce the basic features of quantum physics that are necessary for understanding how quantum computers work.[1]

In ordinary computer chips, bits are physically represented by low and high voltages on wires. But there are many other ways a bit could be stored—for instance, in the state of a hydrogen atom. The single electron in this atom can either be in the ground state (the lowest energy configuration) or it can be in an excited state (a high energy configuration). We can use these two states to encode for bit values 0 and 1, respectively.

Let us now introduce some quantum physics notation. We denote the ground state of our electron by $|0\rangle$, since it encodes for bit value 0, and likewise the excited state

[1]This field is so strange that the famous physicist Richard Feynman is quoted as having said, "I think I can safely say that no one understands quantum physics." So there is little chance you will understand the theory in depth after reading this section! But if you are interested in learning more, see the recommended reading at the book's end.

Figure 10.1 An electron can be in a ground state or in an excited state. In the Dirac notation used in quantum physics, these are denoted $|0\rangle$ and $|1\rangle$. But the superposition principle says that, in fact, the electron is in a state that is *a linear combination* of these two: $\alpha_0|0\rangle + \alpha_1|1\rangle$. This would make immediate sense if the α's were probabilities, nonnegative real numbers adding to 1. But the superposition principle insists that they can be *arbitrary complex numbers*, as long as the squares of their norms add up to 1!



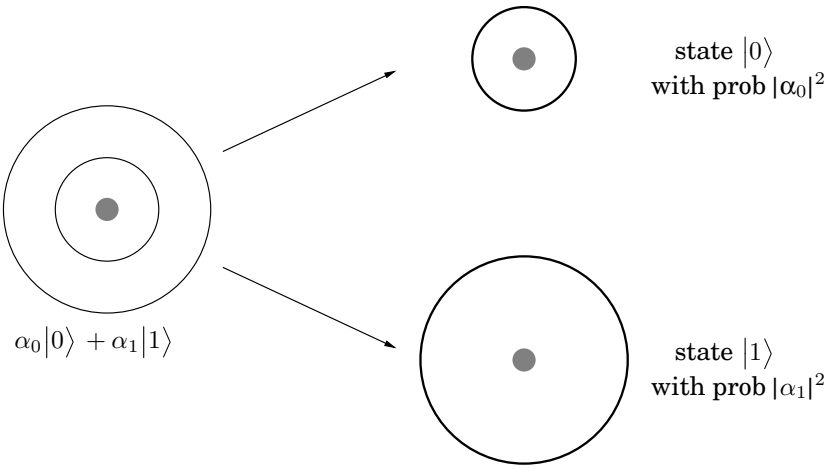
ground state $|0\rangle$ excited state $|1\rangle$ superposition $\alpha_0|0\rangle + \alpha_1|1\rangle$

by $|1\rangle$. These are the two possible states of the electron in classical physics. Many of the most counterintuitive aspects of quantum physics arise from the *superposition principle* which states that if a quantum system can be in one of two states, then it can also be in *any linear superposition* of those two states. For instance, the state of the electron could well be $\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$ or $\frac{1}{\sqrt{2}}|0\rangle - \frac{1}{\sqrt{2}}|1\rangle$; or an infinite number of other combinations of the form $\alpha_0|0\rangle + \alpha_1|1\rangle$. The coefficient α_0 is called the *amplitude* of state $|0\rangle$, and similarly with α_1. And—if things aren't already strange enough—the α's can be complex numbers, as long as they are normalized so that $|\alpha_0|^2 + |\alpha_1|^2 = 1$. For example, $\frac{1}{\sqrt{5}}|0\rangle + \frac{2i}{\sqrt{5}}|1\rangle$ (where i is the imaginary unit, $\sqrt{-1}$) is a perfectly valid quantum state! Such a superposition, $\alpha_0|0\rangle + \alpha_1|1\rangle$, is the basic unit of encoded information in quantum computers (Figure 10.1). It is called a *qubit* (pronounced "cubit").

The whole concept of a superposition suggests that the electron does not make up its mind about whether it is in the ground or excited state, and the amplitude α_0 is a measure of its inclination toward the ground state. Continuing along this line of thought, it is tempting to think of α_0 as the *probability* that the electron is in the ground state. But then how are we to make sense of the fact that α_0 can be negative, or even worse, imaginary? This is one of the most mysterious aspects of quantum physics, one that seems to extend beyond our intuitions about the physical world.

This linear superposition, however, is the private world of the electron. For us to get a glimpse of the electron's state we must make a *measurement*, and when we do so, we get a single bit of information—0 or 1. If the state of the electron is $\alpha_0|0\rangle + \alpha_1|1\rangle$, then the outcome of the measurement is 0 with probability $|\alpha_0|^2$ and

Figure 10.2 Measurement of a superposition has the effect of forcing the system to decide on a particular state, with probabilities determined by the amplitudes.



1 with probability $|\alpha_1|^2$ (luckily we normalized so $|\alpha_0|^2 + |\alpha_1|^2 = 1$). Moreover, the act of measurement causes the system to change its state: if the outcome of the measurement is 0, then the new state of the system is $|0\rangle$ (the ground state), and if the outcome is 1, the new state is $|1\rangle$ (the excited state). This feature of quantum physics, that a measurement disturbs the system and forces it to choose (in this case ground or excited state), is another strange phenomenon with no classical analog.

The superposition principle holds not just for 2-level systems like the one we just described, but in general for k-level systems. For example, in reality the electron in the hydrogen atom can be in one of many energy levels, starting with the ground state, the first excited state, the second excited state, and so on. So we could consider a k-level system consisting of the ground state and the first $k-1$ excited states, and we could denote these by $|0\rangle, |1\rangle, |2\rangle, \ldots, |k-1\rangle$. The superposition principle would then say that the general quantum state of the system is $\alpha_0|0\rangle + \alpha_1|1\rangle + \cdots + \alpha_{k-1}|k-1\rangle$, where $\sum_{j=0}^{k-1} |\alpha_j|^2 = 1$. Measuring the state of the system would now reveal a number between 0 and $k-1$, and outcome j would occur with probability $|\alpha_j|^2$. As before, the measurement would disturb the system, and the new state would *actually become* $|j\rangle$ or the jth excited state.

How do we encode n bits of information? We could choose $k = 2^n$ levels of the hydrogen atom. But a more promising option is to use n qubits.

Let us start by considering the case of two qubits, that is, the state of the electrons of *two* hydrogen atoms. Since each electron can be in either the ground or excited state, in classical physics the two electrons have a total of four possible states—00, 01, 10, or 11—and are therefore suitable for storing 2 bits of information. But in

Entanglement

Suppose we have two qubits, the first in the state $\alpha_0|0\rangle + \alpha_1|1\rangle$ and the second in the state $\beta_0|0\rangle + \beta_1|1\rangle$. What is the joint state of the two qubits? The answer is, the (tensor) product of the two: $\alpha_0\beta_0|00\rangle + \alpha_0\beta_1|01\rangle + \alpha_1\beta_0|10\rangle + \alpha_1\beta_1|11\rangle$.

Given an arbitrary state of two qubits, can we specify the state of each individual qubit in this way? No, in general the two qubits are *entangled* and cannot be decomposed into the states of the individual qubits. For example, consider the state $|\psi\rangle = \frac{1}{\sqrt{2}}|00\rangle + \frac{1}{\sqrt{2}}|11\rangle$, which is one of the famous Bell states. It cannot be decomposed into states of the two individual qubits (see Exercise 10.1). Entanglement is one of the most mysterious aspects of quantum mechanics and is ultimately the source of the power of quantum computation.

quantum physics, the superposition principle tells us that the quantum state of the two electrons is a linear combination of the four classical states,

$$|\boldsymbol{\alpha}\rangle = \alpha_{00}|00\rangle + \alpha_{01}|01\rangle + \alpha_{10}|10\rangle + \alpha_{11}|11\rangle,$$

normalized so that $\sum_{x\in\{0,1\}^2}|\alpha_x|^2 = 1$.[2] Measuring the state of the system now reveals 2 bits of information, and the probability of outcome $x \in \{0,1\}^2$ is $|\alpha_x|^2$. Moreover, as before, if the outcome of measurement is jk, then the new state of the system is $|jk\rangle$: if $jk = 10$, for example, then the first electron is in the excited state and the second electron is in the ground state.

An interesting question comes up here: what if we make a *partial measurement*? For instance, if we measure just the first qubit, what is the probability that the outcome is 0? This is simple. It is exactly the same as it would have been had we measured both qubits, namely, $\Pr\{1\text{st bit } = 0\} = \Pr\{00\} + \Pr\{01\} = |\alpha_{00}|^2 + |\alpha_{01}|^2$. Fine, but how much does this partial measurement disturb the state of the system?

The answer is elegant. If the outcome of measuring the first qubit is 0, then the new superposition is obtained by crossing out all terms of $|\boldsymbol{\alpha}\rangle$ that are inconsistent with this outcome (that is, whose first bit is 1). Of course the sum of the squares of the amplitudes is no longer 1, so we must renormalize. In our example, this new state would be

$$|\boldsymbol{\alpha}_{\text{new}}\rangle = \frac{\alpha_{00}}{\sqrt{|\alpha_{00}|^2 + |\alpha_{01}|^2}}|00\rangle + \frac{\alpha_{01}}{\sqrt{|\alpha_{00}|^2 + |\alpha_{01}|^2}}|01\rangle.$$

Finally, let us consider the general case of n hydrogen atoms. Think of n as a fairly small number of atoms, say $n = 500$. Classically the states of the 500 electrons could be used to store 500 bits of information in the obvious way. But the quantum state

[2] Recall that $\{0,1\}^2$ denotes the set consisting of the four 2-bit binary strings and in general $\{0,1\}^n$ denotes the set of all n-bit binary strings.

of the 500 qubits is a linear superposition of all 2^{500} possible classical states:

$$\sum_{x\in\{0,1\}^n} \alpha_x |x\rangle .$$

It is as if Nature has 2^{500} scraps of paper on the side, each with a complex number written on it, just to keep track of the state of this system of 500 hydrogen atoms! Moreover, at each moment, as the state of the system evolves in time, it is as though Nature crosses out the complex number on each scrap of paper and replaces it with its new value.

Let us consider the effort involved in doing all this. The number 2^{500} is much larger than estimates of the number of elementary particles in the universe. Where, then, does Nature store this information? How could microscopic quantum systems of a few hundred atoms contain more information than we can possibly store in the entire classical universe? Surely this is a most extravagant theory about the amount of effort put in by Nature just to keep a tiny system evolving in time.

In this phenomenon lies the basic motivation for quantum computation. After all, if Nature is so extravagant at the quantum level, why should we base our computers on classical physics? Why not tap into this massive amount of effort being expended at the quantum level?

But there is a fundamental problem: this exponentially large linear superposition is the private world of the electrons. Measuring the system only reveals n bits of information. As before, the probability that the outcome is a particular 500-bit string x is $|\alpha_x|^2$. And the new state after measurement is just $|x\rangle$.

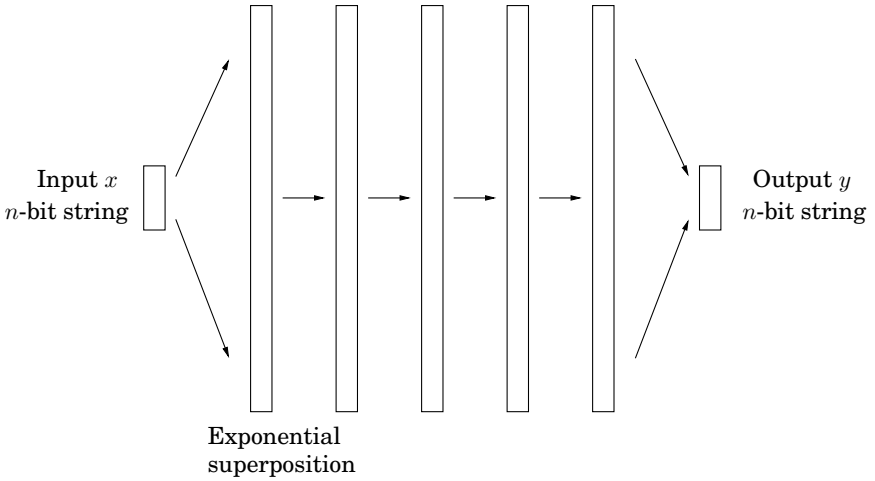
10.2 The plan

A quantum algorithm is unlike any you have seen so far. Its structure reflects the tension between the exponential "private workspace" of an n-qubit system and the mere n bits that can be obtained through measurement.

The input to a quantum algorithm consists of n classical bits, and the output also consists of n classical bits. It is while the quantum system is not being watched that the quantum effects take over and we have the benefit of Nature working exponentially hard on our behalf.

If the input is an n-bit string x, then the quantum computer takes as input n qubits in state $|x\rangle$. Then a series of quantum operations are performed, by the end of which the state of the n qubits has been transformed to some superposition $\sum_y \alpha_y |y\rangle$. Finally, a measurement is made, and the output is the n-bit string y with probability $|\alpha_y|^2$. Observe that this output is *random*. But this is not a problem, as we have seen before with randomized algorithms such as the one for primality testing. As long as y corresponds to the right answer with high enough probability, we can repeat the whole process a few times to make the chance of failure miniscule.

Figure 10.3 A quantum algorithm takes n "classical" bits as its input, manipulates them so as to create a superposition of their 2^n possible states, manipulates this exponentially large superposition to obtain the final quantum result, and then measures the result to get (with the appropriate probability distribution) the n output bits. For the middle phase, there are elementary operations which count as one step and yet manipulate all the exponentially many amplitudes of the superposition.



Now let us look more closely at the quantum part of the algorithm. Some of the key quantum operations (which we will soon discuss) can be thought of as looking for certain kinds of *patterns* in a superposition of states. Because of this, it is helpful to think of the algorithm as having two stages. In the first stage, the n classical bits of the input are "unpacked" into an exponentially large superposition, which is expressly set up so as to have an underlying pattern or regularity that, if detected, would solve the task at hand. The second stage then consists of a suitable set of quantum operations, followed by a measurement, which reveals the hidden pattern.

All this probably sounds quite mysterious at the moment, but more details are on the way. In Section 10.3 we will give a high-level description of the most important operation that can be efficiently performed by a quantum computer: a quantum version of the fast Fourier transform (FFT). We will then describe certain patterns that this quantum FFT is ideally suited to detect, and will show how to recast the problem of factoring an integer N in terms of detecting precisely such a pattern. Finally we will see how to set up the initial stage of the quantum algorithm, which converts the input N into an exponentially large superposition with the right kind of pattern.

The algorithm to factor a large integer N can be viewed as a sequence of reductions (and everything shown here in italics will be defined in good time):

- FACTORING is reduced to finding a *nontrivial square root* of 1 modulo N.
- Finding such a root is reduced to computing the *order* of a random integer modulo N.
- The order of an integer is precisely the *period* of a particular *periodic superposition*.
- Finally, periods of superpositions can be found by the *quantum FFT*.

We begin with the last step.

10.3 The quantum Fourier transform

Recall the fast Fourier transform (FFT) from Chapter 2. It takes as input an M-dimensional, complex-valued vector $\boldsymbol{\alpha}$ (where M is a power of 2, say $M = 2^m$), and outputs an M-dimensional complex-valued vector $\boldsymbol{\beta}$:

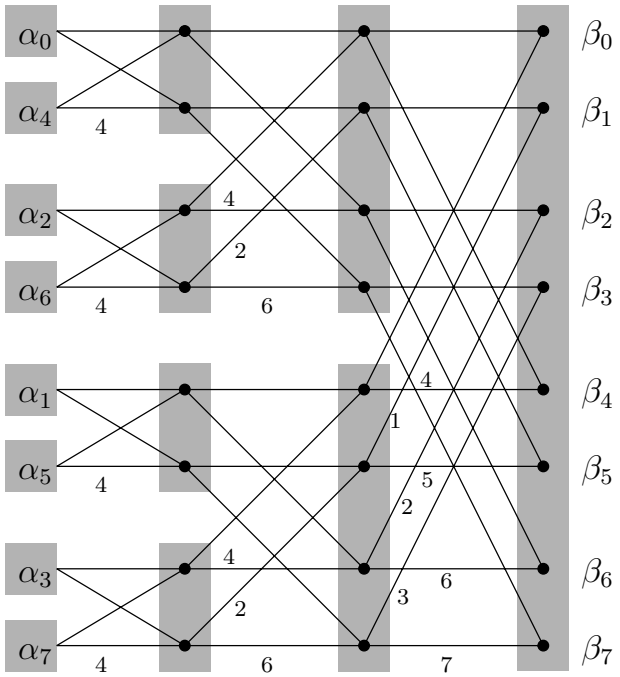
$$\begin{bmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \\ \vdots \\ \beta_{M-1} \end{bmatrix} = \frac{1}{\sqrt{M}} \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \cdots & \omega^{M-1} \\ 1 & \omega^2 & \omega^4 & \cdots & \omega^{2(M-1)} \\ & \vdots & & & \\ 1 & \omega^j & \omega^{2j} & \cdots & \omega^{(M-1)j} \\ & \vdots & & & \\ 1 & \omega^{(M-1)} & \omega^{2(M-1)} & \cdots & \omega^{(M-1)(M-1)} \end{bmatrix} \begin{bmatrix} \alpha_0 \\ \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_{M-1} \end{bmatrix},$$

where ω is a complex Mth root of unity (the extra factor of $\sqrt{M}$ is new and has the effect of ensuring that if the $|\alpha_i|^2$ add up to 1, then so do the $|\beta_i|^2$). Although the preceding equation suggests an $O(M^2)$ algorithm, the classical FFT is able to perform this calculation in just $O(M \log M)$ steps, and it is this speedup that has had the profound effect of making digital signal processing practically feasible. We will now see that quantum computers can implement the FFT *exponentially* faster, in $O(\log^2 M)$ time!

But wait, how can any algorithm take time less than M, the length of the input? The point is that we can encode the input in a superposition of just $m = \log M$ qubits: after all, this superposition consists of 2^m amplitude values. In the notation we introduced earlier, we would write the superposition as $|\boldsymbol{\alpha}\rangle = \sum_{j=0}^{M-1} \alpha_j |j\rangle$ where α_i is the amplitude of the m-bit binary string corresponding to the number i in the natural way. This brings up an important point: the $|j\rangle$ notation is really just another way of writing a vector, where the index of each entry of the vector is written out explicitly in the special bracket symbol.

Starting from this input superposition $|\boldsymbol{\alpha}\rangle$, the *quantum Fourier transform (QFT)* manipulates it appropriately in $m = \log M$ stages. At each stage the superposition evolves so that it encodes the intermediate results at the same stage of the

Figure 10.4 The classical FFT circuit from Chapter 2. Input vectors of M bits are processed in a sequence of $m = \log M$ levels.



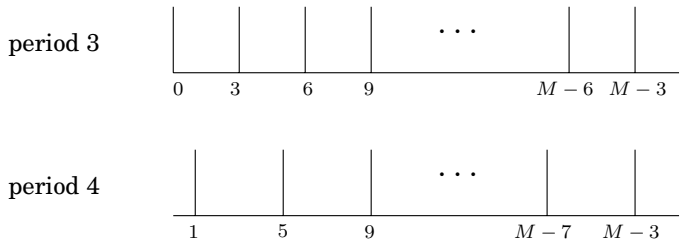
classical FFT (whose circuit, with $m = \log M$ stages, is reproduced from Chapter 2 in Figure 10.4). As we will see in Section 10.5, this can be achieved with m quantum operations per stage. Ultimately, after m such stages and $m^2 = \log^2 M$ elementary operations, we obtain the superposition $|\beta\rangle$ that corresponds to the desired output of the QFT.

So far we have only considered the good news about the QFT: its amazing speed. Now it is time to read the fine print. The classical FFT algorithm actually *outputs* the M complex numbers $\beta_0, \ldots, \beta_{M-1}$. In contrast, the QFT only prepares a superposition $|\beta\rangle = \sum_{j=0}^{M-1} \beta_j |j\rangle$. And, as we saw earlier, these amplitudes are part of the "private world" of this quantum system.

Thus the only way to get our hands on this result is by measuring it! And measuring the state of the system only yields $m = \log M$ classical bits: specifically, the output is index j with probability $|\beta_j|^2$.

So, instead of QFT, it would be more accurate to call this algorithm *quantum Fourier sampling*. Moreover, even though we have confined our attention to the case $M = 2^m$ in this section, the algorithm can be implemented for arbitrary values of M, and can be summarized as follows:

Figure 10.5 Examples of periodic superpositions.



Input: A superposition of $m = \log M$ qubits, $|\alpha\rangle = \sum_{j=0}^{M-1} \alpha_j |j\rangle$.

Method: Using $O(m^2) = O(\log^2 M)$ quantum operations perform the quantum FFT to obtain the superposition $|\beta\rangle = \sum_{j=0}^{M-1} \beta_j |j\rangle$.

Output: A random m-bit number j (that is, $0 \leq j \leq M-1$), from the probability distribution $Pr[j] = |\beta_j|^2$.

Quantum Fourier sampling is basically a quick way of getting a very rough idea about the output of the classical FFT, just detecting one of the larger components of the answer vector. In fact, we don't even see the value of that component—we only see its index. How can we use such meager information? In which applications of the FFT is just the index of the large components enough? This is what we explore next.

10.4 Periodicity

Suppose that the input to the QFT, $|\alpha\rangle = (\alpha_0, \alpha_1, \ldots, \alpha_{M-1})$, is such that $\alpha_i = \alpha_j$ whenever $i \equiv j \bmod k$, where k is a particular integer that divides M. That is, the array α consists of M/k repetitions of some sequence $(\alpha_0, \alpha_1, \ldots, \alpha_{k-1})$ of length k. Moreover, suppose that exactly one of the k numbers $\alpha_0, \ldots, \alpha_{k-1}$ is nonzero, say α_j. Then we say that $|\alpha\rangle$ *is periodic with period k and offset j* (Figure 10.5).

It turns out that if the input vector is periodic, we can use quantum Fourier sampling to compute its period! This is based on the following fact, proved in the next box:

Suppose the input to quantum Fourier sampling is periodic with period k, for some k that divides M. Then the output will be a multiple of M/k, and it is equally likely to be any of the k multiples of M/k.

Now a little thought tells us that by repeating the sampling a few times (repeatedly preparing the periodic superposition and doing Fourier sampling), and then taking

The Fourier transform of a periodic vector

Suppose the vector $|\alpha\rangle = (\alpha_0, \alpha_1, \ldots, \alpha_{M-1})$ is periodic with period k and with no offset (that is, the nonzero terms are $\alpha_0, \alpha_k, \alpha_{2k}, \ldots$). Thus,

$$|\alpha\rangle = \sum_{j=0}^{M/k-1} \sqrt{\tfrac{k}{M}}\, |jk\rangle.$$

We will show that its Fourier transform $|\beta\rangle = (\beta_0, \beta_1, \ldots, \beta_{M-1})$ is also periodic, with period M/k and no offset.

Claim $|\beta\rangle = \frac{1}{\sqrt{k}} \sum_{j=0}^{k-1} |\frac{jM}{k}\rangle$.

Proof. In the input vector, the coefficient α_ℓ is $\sqrt{k/M}$ if k divides ℓ, and is zero otherwise. We can plug this into the formula for the jth coefficient of $|\beta\rangle$:

$$\beta_j = \frac{1}{\sqrt{M}} \sum_{\ell=0}^{M-1} \omega^{j\ell}\alpha_\ell = \frac{\sqrt{k}}{M} \sum_{i=0}^{M/k-1} \omega^{jik}.$$

The summation is a geometric series, $1 + \omega^{jk} + \omega^{2jk} + \omega^{3jk} + \cdots$, containing M/k terms and with ratio ω^{jk} (recall that ω is a complex Mth root of unity). There are two cases. If the ratio is exactly 1, which happens if $jk \equiv 0 \bmod M$, then the sum of the series is simply the number of terms. If the ratio isn't 1, we can apply the usual formula for geometric series to find that the sum is $\frac{1-\omega^{jk(M/k)}}{1-\omega^{jk}} = \frac{1-\omega^{Mj}}{1-\omega^{jk}} = 0$.

Therefore β_j is $1/\sqrt{k}$ if M divides jk, and is zero otherwise. ∎

More generally, we can consider the original superposition to be periodic with period k, but with some offset $l < k$:

$$|\alpha\rangle = \sum_{j=0}^{M/k-1} \sqrt{\tfrac{k}{M}}\, |jk + l\rangle.$$

Then, as before, the Fourier transform $|\beta\rangle$ will have nonzero amplitudes precisely at multiples of M/k:

Claim $|\beta\rangle = \frac{1}{\sqrt{k}} \sum_{j=0}^{k-1} \omega^{lj M/k} |\frac{jM}{k}\rangle$.

The proof of this claim is very similar to the preceding one (Exercise 10.5).

We conclude that *the QFT of any periodic superposition with period k is an array that is everywhere zero, except at indices that are multiples of M/k, and all these k nonzero coefficients have equal absolute values.* So if we sample the output, we will get an index that is a multiple of M/k, and each of the k such indices will occur with probability $1/k$.

the greatest common divisor of all the indices returned, we will with very high probability get the number M/k—and from it the period k of the input!

Let's make this more precise.

Lemma *Suppose s independent samples are drawn uniformly from*

$$0, \frac{M}{k}, \frac{2M}{k}, \ldots, \frac{(k-1)M}{k}.$$

Then with probability at least $1 - k/2^s$, the greatest common divisor of these samples is M/k.

Proof. The only way this can fail is if all the samples are multiples of $j \cdot M/k$, where j is some integer greater than 1. So, fix any integer $j \geq 2$. The chance that a particular sample is a multiple of jM/k is at most $1/j \leq 1/2$; and thus the chance that *all* the samples are multiples of jM/k is at most $1/2^s$.

So far we have been thinking about a particular number j; the probability that this bad event will happen for *some* $j \leq k$ is at most equal to the *sum* of these probabilities over the different values of j, which is no more than $k/2^s$. ∎

We can make the failure probability as small as we like by taking s to be an appropriate multiple of $\log M$.

10.5 Quantum circuits

So quantum computers can carry out a Fourier transform exponentially faster than classical computers. But what do these computers actually look like? What is a *quantum circuit* made up of, and exactly how does it compute Fourier transforms so quickly?

10.5.1 Elementary quantum gates

An elementary quantum operation is analogous to an elementary gate like the AND or NOT gate in a classical circuit. It operates upon either a single qubit or two qubits. One of the most important examples is the Hadamard gate, denoted by H, which operates on a single qubit. On input $|0\rangle$, it outputs $\mathrm{H}(|0\rangle) = \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$. And for input $|1\rangle$, $\mathrm{H}(|1\rangle) = \frac{1}{\sqrt{2}}|0\rangle - \frac{1}{\sqrt{2}}|1\rangle$. In pictures:

Notice that in either case, measuring the resulting qubit yields 0 with probability 1/2 and 1 with probability 1/2. But what happens if the input to the Hadamard gate is an arbitrary superposition $\alpha_0|0\rangle + \alpha_1|1\rangle$? The answer, dictated by the linearity of

quantum physics, is the superposition $\alpha_0 H(|0\rangle) + \alpha_1 H(|1\rangle) = \frac{\alpha_0+\alpha_1}{\sqrt{2}}|0\rangle + \frac{\alpha_0-\alpha_1}{\sqrt{2}}|1\rangle$. And so, if we apply the Hadamard gate to the output of a Hadamard gate, it restores the qubit to its original state!

Another basic gate is the controlled-NOT, or CNOT. It operates upon two qubits, with the first acting as a control qubit and the second as the target qubit. The CNOT gate flips the second bit if and only if the first qubit is a 1. Thus $\text{CNOT}(|00\rangle) = |00\rangle$ and $\text{CNOT}(|10\rangle) = |11\rangle$:

Yet another basic gate, the controlled phase gate, is described below in the subsection describing the quantum circuit for the QFT.

Now let us consider the following question: Suppose we have a quantum state on n qubits, $|\alpha\rangle = \sum_{x\in\{0,1\}^n} \alpha_x |x\rangle$. How many of these 2^n amplitudes change if we apply the Hadamard gate to only the first qubit? The surprising answer is—all of them! The new superposition becomes $|\beta\rangle = \sum_{x\in\{0,1\}^n} \beta_x |x\rangle$, where $\beta_{0y} = \frac{\alpha_{0y}+\alpha_{1y}}{\sqrt{2}}$ and $\beta_{1y} = \frac{\alpha_{0y}-\alpha_{1y}}{\sqrt{2}}$. Looking at the results more closely, the quantum operation on the first qubit deals with each $n-1$ bit suffix y separately. Thus the pair of amplitudes α_{0y} and α_{1y} are transformed into $(\alpha_{0y}+\alpha_{1y})/\sqrt{2}$ and $(\alpha_{0y}-\alpha_{1y})/\sqrt{2}$. This is exactly the feature that will give us an exponential speedup in the quantum Fourier transform.

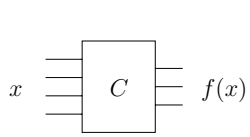
10.5.2 Two basic types of quantum circuits

A quantum circuit takes some number n of qubits as input, and outputs the same number of qubits. In the diagram these n qubits are carried by the n wires going from left to right. The quantum circuit consists of the application of a sequence of elementary quantum gates (of the kind described above) to single qubits and pairs of qubits.

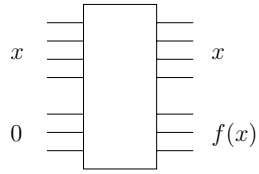
At a high level, there are two basic functionalities of quantum circuits that we use in the design of quantum algorithms:

Quantum Fourier Transform These quantum circuits take as input n qubits in some state $|\alpha\rangle$ and output the state $|\beta\rangle$ resulting from applying the QFT to $|\alpha\rangle$.

Classical Functions Consider a function f with n input bits and m output bits, and suppose we have a classical circuit that outputs $f(x)$. Then there is a quantum circuit that, on input consisting of an n-bit string x padded with m 0's, outputs x and $f(x)$:



Classical circuit



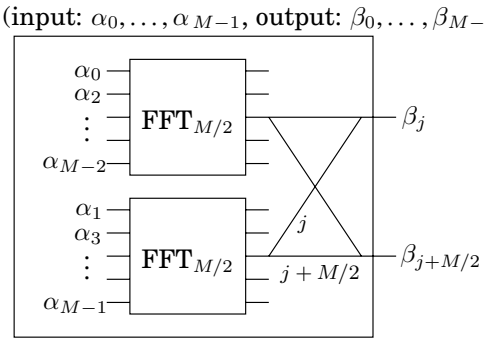
Quantum circuit

Now the input to this quantum circuit could be a superposition over the n bit strings x, $\sum_x |x, 0^k\rangle$, in which case the output has to be $\sum_x |x,\ f(x)\rangle$. Exercise 10.7 explores the construction of such circuits out of elementary quantum gates.

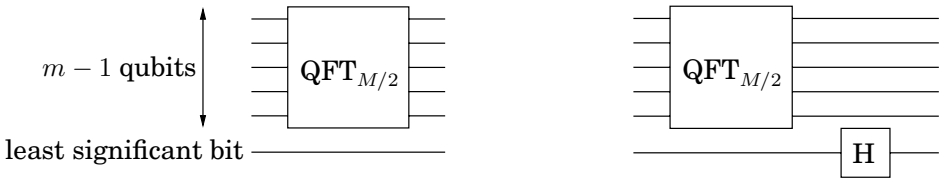
Understanding quantum circuits at this high level is sufficient to follow the rest of this chapter. The next subsection on quantum circuits for the QFT can therefore be safely skipped by anyone not wanting to delve into these details.

10.5.3 The quantum Fourier transform circuit

Here we have reproduced the diagram (from Section 2.6.4) showing how the classical FFT circuit for M-vectors is composed of two FFT circuits for $(M/2)$-vectors followed by some simple gates.



Let's see how to simulate this on a quantum system. The input is now encoded in the 2^m amplitudes of $m = \log M$ qubits. Thus the decomposition of the inputs into evens and odds, as shown in the preceding figure, is clearly determined by one of the qubits—the least significant qubit. How do we separate the even and odd inputs and apply the recursive circuits to compute $FFT_{M/2}$ on each half? The answer is remarkable: just apply the quantum circuit $QFT_{M/2}$ to the remaining $m-1$ qubits. The effect of this is to apply $QFT_{M/2}$ to the superposition of all the m-bit strings of the form $x0$ (of which there are $M/2$), and separately to the superposition of all the m-bit strings of the form $x1$. Thus the two recursive classical circuits can be emulated by a single quantum circuit—an exponential speedup when we unwind the recursion!



Let us now consider the gates in the classical FFT circuit *after* the recursive calls to $FFT_{M/2}$: the wires pair up j with $M/2+j$, and ignoring for now the phase that is applied to the contents of the $(M/2+j)$th wire, we must add and subtract these two quantities to obtain the jth and the $(M/2+j)$th outputs, respectively. How would a quantum circuit achieve the result of these M classical gates? Simple: just perform the Hadamard gate on the first qubit! Recall from the preceding discussion (Section 10.5.1) that for every possible configuration of the remaining $m-1$ qubits x, this pairs up the strings $0x$ and $1x$. Translating from binary, this means we are pairing up x and $M/2+x$. Moreover the result of the Hadamard gate is that for each such pair, the amplitudes are replaced by the sum and difference (normalized by $1/\sqrt{2}$) , respectively. So far the QFT requires almost no gates at all!

The phase that must be applied to the $(M/2+j)$th wire for each j requires a little more work. Notice that the phase of ω^j must be applied only if the first qubit is 1. Now if j is represented by the $m-1$ bits $j_1 \ldots j_{m-1}$, then $\omega^j = \Pi_{l=1}^{m-1} \omega^{2^l j_l}$. Thus the phase ω^j can be applied by applying for the lth wire (for each l) a phase of ω^{2^l} if the lth qubit is a 1 and the first qubit is a 1. This task can be accomplished by another two-qubit quantum gate—the controlled phase gate. It leaves the two qubits unchanged unless they are both 1, in which case it applies a specified phase factor.

The QFT circuit is now specified. The number of quantum gates is given by the formula $S(m) = S(m-1) + O(m)$, which works out to $S(m) = O(m^2)$. The QFT on inputs of size $M = 2^m$ thus requires $O(m^2) = O(\log^2 M)$ quantum operations.

10.6 Factoring as periodicity

We have seen how the quantum Fourier transform can be used to find the period of a periodic superposition. Now we show, by a sequence of simple reductions, how factoring can be recast as a period-finding problem.

Fix an integer N. A *nontrivial square root of* 1 *modulo* N (recall Exercises 1.36 and 1.40) is any integer $x \not\equiv \pm 1 \bmod N$ such that $x^2 \equiv 1 \bmod N$. If we can find a nontrivial square root of 1 mod N, then it is easy to decompose N into a product of two nontrivial factors (and repeating the process would factor N):

Lemma *If x is a nontrivial square root of* 1 *modulo N, then* $\gcd(x+1, N)$ *is a nontrivial factor of N.*

Proof. $x^2 \equiv 1 \bmod N$ implies that N divides $(x^2 - 1) = (x + 1)(x - 1)$. But N does not divide either of these individual terms, since $x \not\equiv \pm 1 \bmod N$. Therefore N must have a nontrivial factor in common with each of $(x + 1)$ and $(x - 1)$. In particular, $\gcd(N, x + 1)$ is a nontrivial factor of N. ∎

Example. Let $N = 15$. Then $4^2 \equiv 1 \bmod 15$, but $4 \not\equiv \pm 1 \bmod 15$. Both $\gcd(4 - 1, 15) = 3$ and $\gcd(4 + 1, 15) = 5$ are nontrivial factors of 15.

To complete the connection with periodicity, we need one further concept. Define the *order* of x modulo N to be the smallest positive integer r such that $x^r \equiv 1 \bmod N$. For instance, the order of 2 mod 15 is 4.

Computing the order of a *random* number $x \bmod N$ is closely related to the problem of finding nontrivial square roots, and thereby to factoring. Here's the link.

Lemma *Let N be an odd composite, with at least two distinct prime factors, and let x be chosen uniformly at random between 0 and $N - 1$. If $\gcd(x, N) = 1$, then with probability at least $1/2$, the order r of $x \bmod N$ is even, and moreover $x^{r/2}$ is a nontrivial square root of $1 \bmod N$.*

The proof of this lemma is left as an exercise. What it implies is that if we could compute the order r of a randomly chosen element $x \bmod N$, then there's a good chance that this order is even and that $x^{r/2}$ is a nontrivial square root of 1 modulo N. In which case $\gcd(x^{r/2} + 1, N)$ is a factor of N.

Example. If $x = 2$ and $N = 15$, then the order of 2 is 4 since $2^4 \equiv 1 \bmod 15$. Raising 2 to half this power, we get a nontrivial root of 1: $2^2 \equiv 4 \not\equiv \pm 1 \bmod 15$. So we get a divisor of 15 by computing $\gcd(4 + 1, 15) = 5$.

Hence we have reduced FACTORING to the problem of ORDER FINDING. The advantage of this latter problem is that it has a natural periodic function associated with it: fix N and x, and consider the function $f(a) = x^a \bmod N$. If r is the order of x, then $f(0) = f(r) = f(2r) = \cdots = 1$, and similarly, $f(1) = f(r + 1) = f(2r + 1) = \cdots = x$. Thus f is periodic, with period r. And we can compute it efficiently by the repeated squaring algorithm from Section 1.2.2. So, in order to factor N, all we need to do is to figure out how to use the function f to set up a periodic superposition with period r; whereupon we can use quantum Fourier sampling as in Section 10.3 to find r. This is described in the next box.

10.7 The quantum algorithm for factoring

We can now put together all the pieces of the quantum algorithm for FACTORING (see Figure 10.6). Since we can test in polynomial time whether the input is a prime or a prime power, we'll assume that we have already done that and that the input is an odd composite number with at least two distinct prime factors.

Input: an odd composite integer N.

Output: a factor of N.

Setting up a periodic superposition

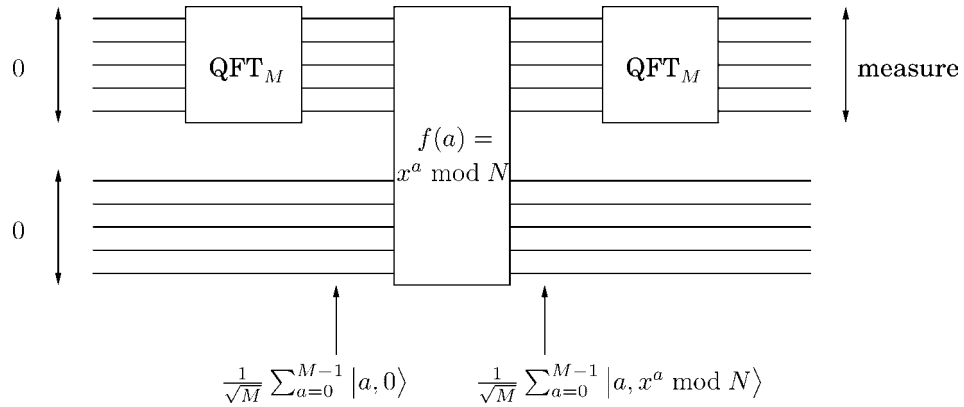
Let us now see how to use our periodic function $f(a) = x^a \bmod N$ to set up a periodic superposition. Here is the procedure:

- We start with two quantum registers, both initially 0.
- Compute the quantum Fourier transform of the first register modulo M, to get a superposition over all numbers between 0 and $M-1$: $\frac{1}{\sqrt{M}} \sum_{a=0}^{M-1} |a, 0\rangle$. This is because the initial superposition can be thought of as periodic with period M, so the transform is periodic with period 1.
- We now compute the function $f(a) = x^a \bmod N$. The quantum circuit for doing this regards the contents of the first register a as the input to f, and the second register (which is initially 0) as the answer register. After applying this quantum circuit, the state of the two registers is: $\sum_{a=0}^{M-1} \frac{1}{\sqrt{M}} |a, f(a)\rangle$.
- We now measure the second register. This gives a periodic superposition on the first register, with period r, the period of f. Here's why:

  Since f is a periodic function with period r, for every rth value in the first register, the contents of the second register are the same. The measurement of the second register therefore yields $f(k)$ for some random k between 0 and $r-1$. What is the state of the first register after this measurement? To answer this question, recall the rules of partial measurement outlined earlier in this chapter. The first register is now in a superposition of only those values a that are compatible with the outcome of the measurement on the second register. But these values of a are exactly $k, k+r, k+2r, \ldots, k+M-r$. So the resulting state of the first register is a periodic superposition $|\alpha\rangle$ with period r, which is exactly the order of x that we wish to find!

1. Choose x uniformly at random in the range $1 \leq x \leq N-1$.
2. Let M be a power of 2 near N (for reasons we cannot get into here, it is best to choose $M \approx N^2$).
3. Repeat $s = 2 \log N$ times:
   (a) Start with two quantum registers, both initially 0, the first large enough to store a number modulo M and the second modulo N.
   (b) Use the periodic function $f(a) \equiv x^a \bmod N$ to create a periodic superposition $|\alpha\rangle$ of length M as follows (see box for details):
      i. Apply the QFT to the first register to obtain the superposition $\sum_{a=0}^{M-1} \frac{1}{\sqrt{M}} |a, 0\rangle$.
      ii. Compute $f(a) = x^a \bmod N$ using a quantum circuit, to get the superposition $\sum_{a=0}^{M-1} \frac{1}{\sqrt{M}} |a, x^a \bmod N\rangle$.

Figure 10.6 Quantum factoring.



iii. Measure the second register. Now the first register contains the periodic superposition $|\alpha\rangle = \sum_{j=0}^{M/r-1} \sqrt{\frac{r}{M}} |jr + k\rangle$ where k is a random offset between 0 and $r - 1$ (recall that r is the order of x modulo N).

(c) Fourier sample the superposition $|\alpha\rangle$ to obtain an index between 0 and $M - 1$.

Let g be the gcd of the resulting indices $j_1, \ldots, j_s$.

4. If M/g is even, then compute $\gcd(N, x^{M/2g} + 1)$ and output it if it is a nontrivial factor of N; otherwise return to step 1.

From previous lemmas, we know that this method works for at least half the choices of x, and hence the entire procedure has to be repeated only a couple of times on average before a factor is found.

But there is one aspect of this algorithm, having to do with the number M, that is still quite unclear: M, the size of our FFT, must be a power of 2. And for our period-detecting idea to work, the period must divide M—hence it should also be a power of 2. But the period in our case is the order of x, definitely not a power of 2!

The reason it all works anyway is the following: *the quantum Fourier transform can detect the period of a periodic vector even if it does not divide M*. But the derivation is not as clean as in the case when the period does divide M, so we shall not go any further into this.

Let $n = \log N$ be the number of bits of the input N. The running time of the algorithm is dominated by the $2 \log N = O(n)$ repetitions of step 3. Since modular exponentiation takes $O(n^3)$ steps (as we saw in Section 1.2.2) and the quantum Fourier transform takes $O(n^2)$ steps, the total running time for the quantum factoring algorithm is $O(n^3 \log n)$.

Implications for computer science and quantum physics

In the early days of computer science, people wondered whether there were much more powerful computers than those made up of circuits composed of elementary gates. But since the seventies this question has been considered well settled. Computers implementing the von Neumann architecture on silicon were the obvious winners, and it was widely accepted that any other way of implementing computers is polynomially equivalent to them. That is, a T-step computation on any computer takes at most some polynomial in T steps on another. This fundamental principle is called *the extended Church-Turing thesis*. Quantum computers violate this fundamental thesis and therefore call into question some of our most basic assumptions about computers.

Can quantum computers be built? This is the challenge that is keeping busy many research teams of physicists and computer scientists around the world. The main problem is that quantum superpositions are very fragile and need to be protected from any inadvertent measurement by the environment. There is progress, but it is very slow: so far, the most ambitious reported quantum computation was the factorization of the number 15 into its factors 3 and 5 using nuclear magnetic resonance (NMR). And even in this experiment, there are questions about how faithfully the quantum factoring algorithm was really implemented. The next decade promises to be really exciting in terms of our ability to physically manipulate quantum bits and implement quantum computers.

But there is another possibility: What if all these efforts at implementing quantum computers fail? This would be even more interesting, because it would point to some fundamental flaw in quantum physics, a theory that has stood unchallenged for a century.

Quantum computation is motivated as much by trying to clarify the mysterious nature of quantum physics as by trying to create novel and superpowerful computers.

Exercises

10.1. $|\psi\rangle = \frac{1}{\sqrt{2}}|00\rangle + \frac{1}{\sqrt{2}}|11\rangle$ is one of the famous "Bell states," a highly entangled state of its two qubits. In this question we examine some of its strange properties.

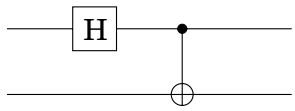
(a) Suppose this Bell state could be decomposed as the (tensor) product of two qubits (recall the box on page 300), the first in state $\alpha_0|0\rangle + \alpha_1|1\rangle$ and the second in state $\beta_0|0\rangle + \beta_1|1\rangle$. Write four equations that the amplitudes α_0, α_1, β_0, and β_1 must satisfy. Conclude that the Bell state cannot be so decomposed.

(b) What is the result of measuring the first qubit of $|\psi\rangle$?

(c) What is the result of measuring the second qubit after measuring the first qubit?

(d) If the two qubits in state $|\psi\rangle$ are very far from each other, can you see why the answer to (c) is surprising?

10.2. Show that the following quantum circuit prepares the Bell state $|\psi\rangle = \frac{1}{\sqrt{2}}|00\rangle + \frac{1}{\sqrt{2}}|11\rangle$ on input $|00\rangle$: apply a Hadamard gate to the first qubit followed by a CNOT with the first qubit as the control and the second qubit as the target.



What does the circuit output on input 10, 01, and 11? These are the rest of the Bell basis states.

10.3. What is the quantum Fourier transform modulo M of the uniform superposition $\frac{1}{\sqrt{M}} \sum_{j=0}^{M-1} |j\rangle$?

10.4. What is the QFT modulo M of $|j\rangle$?

10.5. *Convolution-Multiplication.* Suppose we shift a superposition $|\alpha\rangle = \sum_j \alpha_j |j\rangle$ by l to get the superposition $|\alpha'\rangle = \sum_j \alpha_j |j+l\rangle$. If the QFT of $|\alpha\rangle$ is $|\beta\rangle$, show that the QFT of α' is β', where $\beta'_j = \beta_j \omega^{lj}$. Conclude that if $|\alpha'\rangle = \sum_{j=0}^{M/k-1} \sqrt{\frac{k}{M}}\, |jk+l\rangle$, then $|\beta'\rangle = \frac{1}{\sqrt{k}} \sum_{j=0}^{k-1} \omega^{ljM/k} |jM/k\rangle$.

10.6. Show that if you apply the Hadamard gate to the inputs and outputs of a CNOT gate, the result is a CNOT gate with control and target qubits switched:

10.7. The CONTROLLED SWAP (C-SWAP) gate takes as input 3 qubits and swaps the second and third if and only if the first qubit is a 1.

(a) Show that each of the NOT, CNOT, and C-SWAP gates are their own inverses.

(b) Show how to implement an AND gate using a C-SWAP gate, i.e., what inputs a, b, c would you give to a C-SWAP gate so that one of the outputs is $a \wedge b$?

(c) How would you achieve fanout using just these three gates? That is, on input a and 0, output a and a.

(d) Conclude therefore that for any classical circuit C there is an equivalent quantum circuit Q using just NOT and C-SWAP gates in the following sense: if C outputs y on input x, then Q outputs $|x, y, z\rangle$ on input $|x, 0, 0\rangle$. (Here z is some set of junk bits that are generated during this computation.)

(e) Now show that that there is a quantum circuit Q^{-1} that outputs $|x, 0, 0\rangle$ on input $|x, y, z\rangle$.

(f) Show that there is a quantum circuit Q' made up of NOT, CNOT, and C-SWAP gates that outputs $|x, y, 0\rangle$ on input $|x, 0, 0\rangle$.

10.8. In this problem we will show that if $N = pq$ is the product of two odd primes, and if x is chosen uniformly at random between 0 and $N - 1$, such that $\gcd(x, N) = 1$, then with probability at least 3/8, the order r of x mod N is even, and moreover $x^{r/2}$ is a nontrivial square root of 1 mod N.

(a) Let p be an odd prime and let x be a uniformly random number modulo p. Show that the order of x mod p is even with probability at least 1/2. (*Hint:* Use Fermat's little theorem (Section 1.3).)

(b) Use the Chinese remainder theorem (Exercise 1.37) to show that with probability at least 3/4, the order r of x mod N is even.

(c) If r is even, prove that the probability that $x^{r/2} \equiv \pm 1$ is at most 1/2.

Historical notes and further reading

Chapters 1 and 2

The classical book on the theory of numbers is

> *G. H. Hardy and E. M. Wright, Introduction to the Theory of Numbers. Oxford University Press, 1980.*

The primality algorithm was discovered by Robert Solovay and Vblker Strassen in the mid-1970's, while the RSA cryptosystem came about a couple of years later. See

> *D. R. Stinson, Cryptography: Theory and Practice. Chapman and Hall, 2005*

for much more on cryptography. For randomized algorithms, see

> *R. Motwani and P. Raghavan, Randomized Algorithms. Cambridge University Press, 1995.*

Universal hash functions were proposed in 1979 by Larry Carter and Mark Wegman. The fast matrix multiplication algorithm is due to Volker Strassen (1969). Also due to Strassen, with Arnold Schönhage, is the fastest known algorithm for integer multiplication. It uses a variant of the FFT to multiply n-bit integers in $O(n \log n \log\log n)$ bit operations.

Chapter 3

Depth-first search and its many applications were articulated by John Hopcroft and Bob Tarjan in 1973—they were honored for this contribution by the Turing award, the highest distinction in Computer Science. The two-phase algorithm for finding strongly connected components is due to Rao Kosaraju.

Chapters 4 and 5

Dijkstra's algorithm was discovered in 1959 by Edsger Dijkstra (1930–2002), while the first algorithm for computing minimum spanning trees can be traced back to a 1926 paper by the Czech mathematician Otakar Boruvka. The analysis of the union-find data structure (which is actually a little more tight than our $\log^* n$ bound) is due to Bob Tarjan. Finally, David Huffman discovered in 1952, while a graduate student, the encoding algorithm that bears his name.

Chapter 7

The simplex method was discovered in 1947 by George Danzig (1914–2005), and the min-max theorem for zero-sum games in 1928 by John von Neumann (who is also considered the father of the computer). A very nice book on linear programming is

> *V. Chvátal, Linear Programming. W. H. Freeman, 1983.*

And for game theory, see

Martin J. Osborne and Ariel Rubinstein, A course in game theory. M.I.T. Press, 1994.

Chapters 8 and 9

The notion of **NP**-completeness was first identified in the work of Steve Cook, who proved in 1971 that SAT is **NP**-complete; a year later Dick Karp came up with a list of 23 **NP**-complete problems (including all the ones proven so in Chapter 8), establishing beyond doubt the applicability of the concept (they were both given the Turing award). Leonid Levin, working in the Soviet Union, independently proved a similar theorem.

For an excellent treatment of **NP**-completeness see

M. R. Garey and D. S. Johnson, Computers and Intractability: A Guide to the Theory of **NP***-completeness. W. H. Freeman, 1979.*

And for the more general subject of Complexity see

C. H. Papadimitriou, Computational Complexity. Addison-Wesley, Reading Massachusetts, 1995.

Chapter 10

The quantum algorithm for primality was discovered in 1994 by Peter Shor. For a novel introduction to quantum mechanics for computer scientists see

http://www.cs.berkeley.edu/~vazirani/quantumphysics.html

and for an introduction to quantum computation see the notes for the course "Qubits, Quantum Mechanics, and Computers" at

http://www.cs.berkeley.edu/~vazirani/cs191.html

Index

This text, extensively class-tested over a decade at UC Berkeley and UC San Diego, explains the fundamentals of algorithms in a story line that makes the material enjoyable and easy to digest.

Emphasis is placed on understanding the crisp mathematical idea behind each algorithm, in a manner that is intuitive and rigorous without being unduly formal.

Features include:

- The use of boxes to strengthen the narrative: pieces that provide historical context, descriptions of how the algorithms are used in practice, and excursions for the mathematically sophisticated.
- Carefully chosen advanced topics that can be skipped in a standard one-semester course, but can be covered in an advanced algorithms course or in a more leisurely two-semester sequence.
- An accessible treatment of linear programming introduces students to one of the greatest achievements in algorithms. An optional chapter on the quantum algorithm for factoring provides a unique peephole into this exciting topic.

ISBN 978-0-07-352340-8
MHID 0-07-352340-2

EAN 9 780073 523408 90000

www.mhhe.com

The McGraw-Hill Companies

McGraw-Hill Higher Education