CHAPTER 1

A Sampling of Problems

The study of algorithms cannot be dissociated from the study of problems. The first step in the solution of any problem is the specification, in terms as clear as possible, of the problem's parameters and objectives. In spite of appearances, this step may prove quite challenging: the range of problems that algorithm designers have to face can be quite broad, and each problem may be initially presented in an extremely complex manner, encumbered by a myriad of details. In order to analyze the problem and develop algorithms for its solution, it is crucial that the problem be reduced to an abstract formulation, devoid of any details—however important in practice—that have no bearing on algorithmic development. Since optimization problems often spring from real situations—many tracing their sources to the planning of production and delivery schedules and the reduction of manufacturing costs—developing a suitable abstract formulation requires some thought and some experience. In this first chapter we propose to take the reader through a guided tour of typical combinatorial optimization problems, with three goals in mind: (i) to illustrate the development of formal problem specifications from informal English descriptions; (ii) to provide an overview of the type of problems that we shall examine (and thus demonstrate, albeit indirectly, the wide range of applicability of the methods that we shall study); and (iii) to instill in the reader a sense of adventure, a spirit of exploration, and a mood for play, all attitudes which we think are both elicited by and conducive to the study of algorithms.

We propose a tour of *combinatorial optimization* problems, not of all possible types of optimization problems. The common feature of all combinatorial problems is their discrete and finite nature: their parameters can only take on discrete values, and their solutions can only be drawn from a finite set of possibilities. The finiteness of these problems generally ensures the existence of some brute-force method of solution: simply generate all possible solution structures, always keeping the best one. Such methods, however, can only succeed with fairly small solutions sets. Thus the subject of these two volumes is the design of efficient algorithms for the solution of combinatorial optimization problems characterized by the enormous size of their set of potential solutions. All of the problems introduced in this chapter share this feature; for convenience of presentation, they have been further grouped into categories based on their fields of application.

1.1 Integer Programming Problems

Many situations of economic interest can be modelled as a set of linear equations in which the variables represent continuously varying quantities which are required to remain within certain limits. The objective in studying such systems is to maximize or minimize some aspect of the system—for example, profit, cost, or equitability of distribution of goods—without violating any of the constraints imposed on the variables by the linear equations. Problems of this nature are formalized as *linear programs*, and the study of their solution is called *linear programming*. However, the problems that we shall consider differ from the linear programming formalism in a significant way: the variables will be restricted to a finite set of values. We begin with an example known as the *knapsack* problem.

A hiker planning a backpacking trip feels that he can comfortably carry at most 20 kilograms. After laying out all the items that he wants to take and discovering that their total weight exceeds 20 kilograms, he assigns to each item a "value" rating, as shown in Table 1.1. Which items should he take to maximize the value of what he can carry without exceeding 20 kilograms?

Table 1.1: An Instance of the Knapsack Problem

Item	Tent	Canteen (filled)	Change of clothes	Camp stove	Sleeping bag	Dried food
Weight	11	7	5	4	3	3
Value	20	10	11	5	25	50

Item	First-aid kit	Mosquito repellent	Flashlight	Novel	Rain gear	Water purifier
Weight	3	2	2	2	2	1
Value	15	12	6	4	5	30

Although we do not yet know how to obtain the solution, the way to fill the knapsack to carry the most value is to take the sleeping bag, food, mosquito repellent, first-aid kit, flashlight, water purifier, and change of clothes, for a total value of 149 with a total weight of 19 kilograms. An interesting aspect of the solution is that it is not directly limited by the weight restriction. There are ways of filling the knapsack with exactly 20 kilograms, such as substituting for the change of clothes the camp stove and rain gear, but this decreases the total value.

Formally, the knapsack problem can be stated as follows.

Problem 1 [Knapsack] Given M, the capacity of the knapsack, $\{ w_i \mid w_i > 0, i = 1, 2, \ldots, n \}$, the weights of the n objects, and $\{ v_i \mid v_i > 0, i = 1, 2, \ldots, n \}$, their corresponding values,

$$\begin{aligned} \text{maximize} \quad & \sum_{i=1}^{n} v_i x_i \\ \text{subject to} \quad & \sum_{i=1}^{n} w_i x_i \leq M \\ \text{where} \quad & x_i \in \{0, 1\}. \end{aligned} \tag{1.1}$$

($x_i = 0$ means that item i should not be included in the knapsack, and $x_i = 1$ means that it should be included.) □

If x_i equals zero, then $v_i x_i$ and $w_i x_i$ will be zero, so that the value of the ith item will not be included in the sum that represents the total value and its weight will not be included in the sum of the weights. If, on the other hand, x_i equals one, then the value of the ith item contributes to the combined value of all things selected for inclusion and its weight similarly contributes to the combined weight. In consequence, the solution space has size 2^n, since there are two choices for each of n independent variables. This method of encoding is used over and over again for problems where something is being included or excluded, made true or false, etc.

A more realistic version of the knapsack problem would add volume restrictions to the weight restrictions, thereby obtaining a multidimensional version of the knapsack problem. However, modelling the volume constraints by a simple inequality ignores the fact that objects are three-dimensional solids; it models a soft knapsack to be filled with plastic objects (e.g., objects that come in instant, powdered form). An accurate model of the real-world knapsack problem requires large numbers of constraints to express the various restrictions placed on three-dimensional arrangements; moreover, a solution to such a problem should include not only a list of the chosen objects, but also packing instructions.

Problems of this type are called *optimization* problems, because there is some function, called the *objective function*, which is to be maximized or minimized. The restrictions placed on the variables, x_i, $i = 1, 2, \ldots, n$, in words or in formulae, are called *constraints*. When formalized in this way, the knapsack problem has an extremely simple form. The single "subject to" constraint in 1.1 is a linear inequality, and the objective function is also linear. A problem that can be formalized as

$$\begin{array}{rl}
\text{optimize} & \displaystyle\sum_{i=1}^{n} c_i x_i \\
\text{subject to} & \left\{\begin{array}{ccccccccc}
a_{11}x_1 & + & a_{12}x_2 & + \cdots + & a_{1n}x_n & \le & b_1 \\
a_{21}x_1 & + & a_{22}x_2 & + \cdots + & a_{2n}x_n & \le & b_2 \\
\vdots & & \vdots & \ddots & \vdots & & \vdots \\
a_{m1}x_1 & + & a_{m2}x_2 & + \cdots + & a_{mn}x_n & \le & b_m
\end{array}\right. \\
\text{where} & x_i \ge 0, \quad i = 1,\, 2,\, \ldots,\, n
\end{array}$$

is called a *linear programming* problem (LP). When specific numerical values are assigned to the constants in the above formulae (for c_i, a_{ij}, and b_i; or for v_i, w_i, and M), we say that we have an *instance* of the problem, so that names like "knapsack problem" actually refer to a family of similar problems. Unlike the x_i, the c_i, a_{ij}, and b_i may be negative. The point—or more precisely a point, since there can be more than one—that satisfies the constraints and makes the objective function achieve its maximum or minimum is called an *optimal solution*; we denote it by $(\hat{x}_1, \hat{x}_2, \ldots, \hat{x}_n)$. The direction of each inequality in the "subject to" constraints depends on the problem; strict equalities can also be used, since an equality can be replaced by two inequalities.

The formulation of the knapsack problem does differ from this prototype in an important way, however. In a linear programming problem, the x_i are not restricted to discrete values, but are continuous variables. Certainly the presence of a constraint of the form "x_i integral" can affect the optimal solution, since a formerly optimal solution, such as $(12^1\!/_4, 2^2\!/_3)$, no longer satisfies all the constraints. It is intuitively reasonable to expect that the introduction of integrality constraints will force the optimal solution to an all-integer point that is near the optimal solution that would otherwise be obtained. Unfortunately, as the following example shows, this intuition is incorrect.

A brewery makes two types of beer: lite and light. The key ingredients used, malt, hops, and yeast, are in limited supply, while other ingredients, like water, can be considered in infinite supply. It takes 2 units of malt, 4 of hops, and 2 of yeast to make 100 liters of lite and 3 units of malt, 1 of hops, and 9 of yeast to make 100 liters of light. (The unit of measurement is not the same for the three ingredients.) The brewery can sell all its output at a profit of \$21 per 100 liters of lite and \$31 per 100 liters of light. They are able to obtain 25 units of malt, 32 of hops, and 54 of yeast. How should they schedule their production to maximize profit?

This problem is embodied in the linear program

$$\begin{array}{rl} \text{maximize} & 21x + 31y \\ \text{subject to} & \left\{ \begin{array}{rcl} 2x + 3y & \leq & 25 \\ 4x + y & \leq & 32 \\ 2x + 9y & \leq & 54 \end{array} \right. \\ \text{where} & x,\, y \geq 0. \end{array}$$

In this linear program, the variables x and y represent the amount, in 100-liter units, of lite and light to make, respectively, and each of the constraints given by the linear inequalities says that the amount of one of the critical ingredients used cannot exceed the supply. "x, $y \geq 0$" is the obvious condition that a negative quantity cannot be produced. Constraints such as "x, $y \geq 0$" or "$x_i = 0$ or 1," which do not appear in the problem statement, which are generic to all of the problems that we shall be considering, and which can be imposed by a computer program without significant computation, are called *implicitconstraints*. The constraints that define the essence of the problem are called *explicitconstraints*, or often simply "the constraints," as if to say that the implicit constraints are so obvious as to make their statement redundant. The methods used today to solve linear programs are variations on the *simplex method*, developed by George Dantzig and a number of colleagues in the late 1940s. For our simple problem, however, it is possible to proceed graphically. Figure 1.1 indicates the regions of the first quadrant $(x, y \geq 0)$ for which each inequality holds. The points in the shaded region satisfy all of the constraints, both implicit and explicit, and are termed *feasible solutions*. The shaded region itself is called the *feasible region*. Points that satisfy the implicit constraints, especially if they are of the form "$x_i = 0$ or 1," define what is often called the *solution space*; the individual points are then referred to as *solutions*. This use of the term *solution* is an abuse of language, since a solution as just defined need not satisfy all explicit constraints and thus is not a solution at all in the normal sense of the word. What would normally be called a solution is what we have defined as a feasible solution.[1]

We can picture the line $21x + 31y = C$, which describes the set of solutions which have the same objective function value, being slowly moved toward the origin. As the value of C decreases, the line moves closer to the feasible region. At $(x, y) = (7^1\!/_{10}, 3^3\!/_5)$, where the objective function first crosses the feasible region, the objective function is maximized. The brewery achieves a maximum profit of \$260.70 on 710 liters of lite and 360 liters of light. Notice that all of the malt and hops available will be used but that there is an excess of yeast.

We now consider a problem that appears extremely similar. Instead of a brewery producing two types of beer, imagine a seamstress who plans to produce a limited number of copies of two suit styles. The suits require three fabrics, 2 meters of

[1] One reason for the terminology is that enumeration of only those solutions which are feasible is often not practical, since the distinction between a feasible and an infeasible solution requires a significant amount of computation and is best done after the solution has been generated.

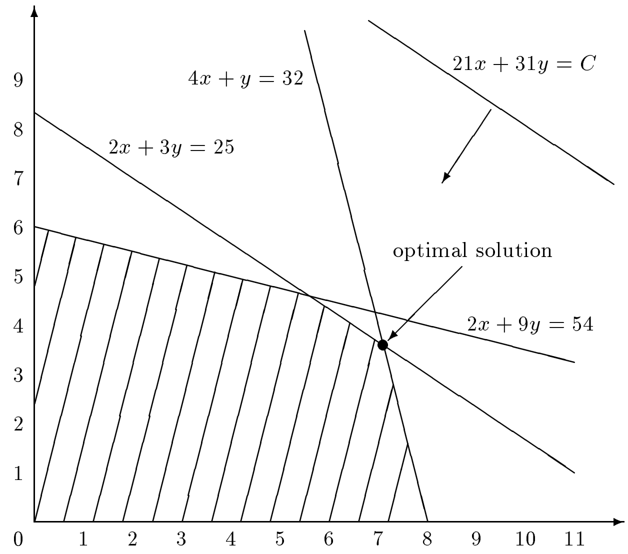


Figure 1.1: A Graphical Solution to the Problem of Planning Brewery Production

the first, 4 of the second, and 2 of the third being required for one style and 3 meters of the first, 1 of the second, and 9 of the third being required for the other style. If the seamstress has available the same amounts of material—her ingredients—that the brewery had available to it and if the slope of her objective function is also $-21/31$, how many suits of each type should she make? The explicit constraints are identical to what we had before, but the answer of $(x, y) = (7^1/_{10}, 3^3/_5)$ is unacceptable: a fraction of a suit cannot be sold. The two problems are not quite the same. The implicit constraint is now

$$x, y \geq 0, \quad x, y \text{ integer.}$$

Of the four points closest to $(7^1/_{10}, 3^3/_5)$ with integer components, three of them, $(7, 4)$, $(8, 3)$, and $(8, 4)$, violate at least one of the explicit constraints. The truncated solution of $(7, 3)$ is feasible, with an objective function value of \$2400, if we assume \$210 and \$310 profit per suit, respectively. It is not optimal, however, since $(6, 4)$ provides a profit of \$2500; $(4, 5)$, with a profit of \$2390, is not far behind. The situation is illustrated in Figure 1.2. A linear programming problem, that includes the additional constraint that all variables be integer valued, is called an *integer linear programming* problem (ILP). If the variables are further restricted to 0 or 1, the problem is called a *0-1 integer linear programming* problem, or just a *0-1 programming* problem for short. The feasible region of an ILP is a set of discrete points in n-dimensional space, those points with all integer components that satisfy the explicit and nonnegativity constraints.

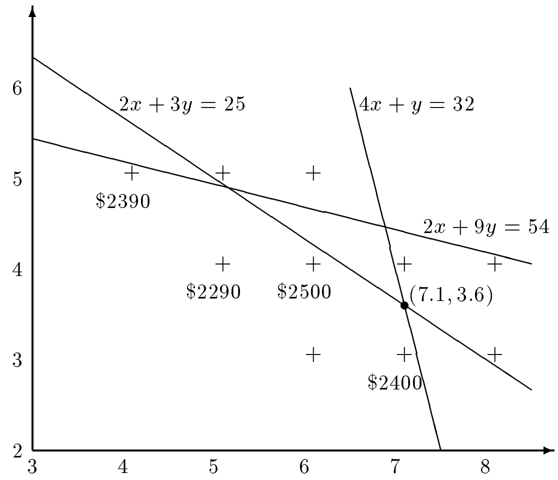


Figure 1.2: The Effect of Integrality Constraints on the Solution to a Linear Programming Problem

Definition 1.1 Let P_1 and P_2 be two LPs or ILPs over n variables with the same objective function. If the feasible region of P_1, $F(P_1)$, is a subset of the feasible region of P_2, $F(P_2)$, then P_2 is called a *relaxation* of P_1. □

An obvious result which will be very useful is the following:

Theorem 1.1 If P_2 is a relaxation of P_1, then the value of the objective function at the optimal feasible solution for problem P_2 is greater than or equal to (less than or equal to for minimization problems) the value of the objective function at the optimal feasible solution to P_1. □

The proof consists merely of noting that, in maximizing the objective function for P_2, the arguments to the objective function range over $F(P_2) \supseteq F(P_1)$. An immediate corollary is the following:

Corollary 1.1 Let P_2 be a relaxation of P_1; if the optimal feasible solution to P_2 is also contained in $F(P_1)$, then it optimizes P_1 as well. □

We have overlooked a subtle point in the proof of Theorem 1.1. If the feasible region is *bounded* (i.e., every variable satisfies $x_i \leq u_i$, the u_i constants) and the problem is an ILP, then the objective function will achieve its maximum, since then there are only finitely many feasible solutions. For the case of continuous variables, the existence of a definite point that is the optimal feasible solution (in certain circumstances there can be infinitely many optimal solutions) may not be apparent. The proof that such a point exists, while not complex, depends on some elementary real analysis, and we shall not present it. The u_i can be explicit in the problem or implied by the other constraints; for example, $2x + 3y \leq 25$ and $y \geq 0$ implies $x \leq 12.5$. If the feasible region is unbounded, the objective function may also be unbounded. It is also possible for an LP or ILP to have conflicting constraints that reduce the feasible region to the empty set; such a problem is called *infeasible*.

Exercise 1.1 Show that if an LP has two or more distinct optimal solutions, then it has infinitely many optimal solutions. *Hint*: Construct, pictorially, a two-dimensional LP problem with two distinct optimal solutions in order to see how the infinitely many solutions arise. □

A case of special interest arises from an ILP and the corresponding LP with the integrality constraints removed. The resulting LP is a relaxation of the ILP, since its feasible region is defined by the same explicit and nonnegativity constraints. The corollary becomes:

Corollary 1.2 Let an LP be formed from an ILP by dropping the integrality constraints; if the optimal solution to the LP is all integer, then it also optimizes the ILP. □

As we saw in the preceding example, merely truncating the fractional solution may lead to a feasible but nonoptimal solution. If the point formed by truncating the solution to the LP is feasible, then it does provide a useful bound for the potential error. If we let $\mathbf{x}_0$ be the optimal solution to the LP, $\lfloor \mathbf{x}_0 \rfloor$ be the truncated feasible solution to the ILP, and $\hat{\mathbf{x}}$ be the optimal solution to the ILP, then

$$f(\lfloor \mathbf{x}_0 \rfloor) \leq f(\hat{\mathbf{x}}) \leq f(\mathbf{x}_0),$$

where f is the objective function and we have assumed a maximization problem.

In many applications, where the solutions are fractional but have a large integral component, truncation may provide an acceptable solution. Consider an automobile manufacturer planning the production levels of several models of cars with the goal of maximizing its profit. Since the production figures are very large, the truncated solution, while not necessarily optimal, will show only a small percentage decrease in the value of the objective function. The values of some of the coefficients in the constraints will have enough uncertainty—some are likely to be based on economic forecasts or consumer surveys—that the true optimal solution is no more meaningful than any nearly optimal feasible solution. On the other hand, when the discretization is relatively coarse, as in a 0-1 programming problem, the use of the truncated solution in place of the true optimal is often unacceptable.

Other problems can also be formulated as 0-1 programming problems.

Problem 2 [Set Cover] Given S, a set containing m elements, and n subsets of S, $\{\, S_i \mid i = 1, 2, \ldots, n \,\}$, with $\bigcup_{i=1}^{n} S_i = S$, what is the smallest collection of the S_i, the union of which is S? (A collection of sets, the union of which is S, is called a *cover*. The problem is to find the cover consisting of the fewest subsets.) $\square$

As an example, consider the following subsets of a set of 16 elements:

$$
\begin{array}{ll}
S_1 = \{1,\ 5,\ 13\} & S_2 = \{4,\ 8,\ 12\} \\
S_3 = \{1,\ 2,\ 3,\ 4\} & S_4 = \{5,\ 6,\ 7,\ 9,\ 12\} \\
S_5 = \{9,\ 10,\ 12,\ 14\} & S_6 = \{8,\ 10,\ 11,\ 13,\ 14\} \\
S_7 = \{11,\ 12,\ 15,\ 16\} & S_8 = \{13,\ 14,\ 15,\ 16\} \\
S_9 = \{2,\ 3,\ 6,\ 7,\ 10,\ 11,\ 14,\ 15\} &
\end{array}
$$

The collection $\{S_3,\ S_4,\ S_6,\ S_8\}$ forms a minimum cover, since their union is S and there is no cover consisting of three subsets. (In this problem, the optimal cover is not unique.) The set cover problem can be formulated as a 0-1 programming problem in much the same manner as the knapsack problem. Again, $x_i = 0$ (or 1) is taken to mean that S_i is not (is) included in the cover. In addition, define a_{ij}, for $i = 1, 2, \ldots, m$ and $j = 1, 2, \ldots, n$, by

$$
a_{ij} = \begin{cases} 1, & \text{if element } i \in S_j \\ 0, & \text{otherwise.} \end{cases}
$$

The a_{ij} vary from instance to instance, but are constants once the instance is specified. The x_i, however, are truly variables. With these definitions, the set covering problem becomes

$$
\begin{array}{rl}
\text{minimize} & \displaystyle\sum_{i=1}^{n} x_i \\
\text{subject to} & \left\{ \begin{array}{ccccccccc}
a_{11}x_1 & + & a_{12}x_2 & + & \cdots & + & a_{1n}x_n & \geq & 1 \\
a_{21}x_1 & + & a_{22}x_2 & + & \cdots & + & a_{2n}x_n & \geq & 1 \\
\vdots & & \vdots & & \ddots & & \vdots & & \vdots \\
a_{m1}x_1 & + & a_{m2}x_2 & + & \cdots & + & a_{mn}x_n & \geq & 1
\end{array} \right. \\
\text{where} & x_i = 0 \text{ or } 1, \quad i = 1, 2, \ldots, n.
\end{array}
$$

The value of the objective function at the optimal solution is the number of sets in the minimum cover. Each linear inequality in the constraints guarantees that an element is in the union of the selected subsets. S_j contains element i and S_j is included in the cover if and only if $a_{ij}x_j$ equals 1. The value of the left-hand side of the ith inequality is the number of times the ith element is covered by the subcollection comprising the cover, and this number must be one or more.

Of all the problems considered, set covering appears to be the most theoretical, with no possible application. In fact, this problem does model important practical problems, however imperfectly, and has received considerable attention in the professional literature. The *airline crew scheduling problem* is an example. An airline has m flights scheduled every day; management wants to know the minimum number of crews needed to service all flights. In the terminology of the set covering problem, a flight is an element of the set S. A subset corresponds to a set of different flights that can be serviced by the same crew. The S_j are determined in advance by consulting the schedule—a crew can serve on flight k after serving on flight i only if flight k departs from the same airport that serves as the destination for flight i and the departure time for flight k is suitably later than the arrival time for flight i—and by heeding various regulations. To say that S must be covered is to say that every flight must have a crew assigned to it. The value of the objective function is the number of required crews.

In this problem, the number of subsets will greatly exceed the number of flights, since the number of flight-to-flight transfers possible at some destinations may be quite large, creating many options for any one crew. The number of crews actually required can be expected to be smaller than the number of flights by a factor of three or more, since one crew normally services several domestic flights in a day. The set covering problem only models the airline crew scheduling problem inexactly, since a number of realistic secondary constraints are not correctly handled. For example, if two or more crews are scheduled for the same flight, i.e., an element is contained in more than one subset that composes the cover, presumably all but one of the crews rest. The airline may consider this to be a penalty situation, since seats that could be sold to passengers must now be reserved to move crews to their next destination. The objective function can be modified to take this situation into account. Another difficulty that is easily handled is that wide-body aircraft may require staffing by more than one crew. Yet another aspect of the real problem not captured in the set covering formulation is that management presumably prefers schedules in which the crew does not change aircraft, since this prevents delays from propagating through the entire schedule and alleviates the need to call up reserve crews on short notice to handle flights for which the correct crew has not arrived. This can be handled partially by imposing additional rules on the formation of the S_j, effectively eliminating certain combinations of flights that fulfill all legal requirements, but are deemed to be too risky. An additional advantage of this formulation is that it decreases n, the number of subsets, and thus reduces the size of the instance.

1.2 Path Problems in Graphs

Many problems reduce to finding paths in a graph that are optimal under some measure. Probably the most widely known optimization problem, the travelling salesperson problem, falls into this category.

A salesman has customers in each of n cities, including that in which he lives. He is planning a round trip in order to visit each of his customers. In what order should he visit them to minimize the distance travelled? There are many abstract formulations of this problem, each version incorporating various constraints designed to model the problem with greater realism. In its most general form, the problem can be expressed as follows.

Problem 3 [Travelling Salesperson] Given a directed graph, $G = (V, A)$, of n vertices and a distance function, $d: V \times V \to \mathcal{R}^+ \cup \{\infty\}$, such that[2]

$$d(v_i, v_j) = \begin{cases} D_{ij} \in \mathcal{R}^+, & \text{if } (v_i, v_j) \in A \\ \infty, & \text{if } (v_i, v_j) \notin A, \end{cases} \tag{1.2}$$

find a simple cycle, $v_{i_0} \to v_{i_1} \to \cdots \to v_{i_n} = v_{i_0}$, which minimizes

$$\sum_{k=0}^{n-1} d(v_{i_k}, v_{i_{k+1}}). \qquad \square$$

A *simple cycle* is a path that passes through no vertex twice, except for the starting and ending vertices, which are the same. That the number of edges of the path is specified to be n, the same as the number of vertices, implies that each vertex is visited exactly once in the cycle. The size of the solution space can reach $(n-1)!$; that is, there can be up to $(n-1)!$ possible cycles from which the cycle of minimum length must be found. The worst case occurs in complete graphs where $(n-1)!$ is derived as follows. The starting and ending vertex can be considered to be v_1, since any vertex in a cycle is just as good a starting and ending point as any other; also, the first vertex on the path after v_1 can be chosen arbitrarily from the remaining $n-1$ vertices. There are then $n-2$ choices for the next vertex, given that v_1 and the vertex chosen already have been eliminated, etc. The size of the solution space to this problem is even larger than the 2^n size for the knapsack and set covering problems.

The distance function can be viewed in greatest generality by considering it as an $n \times n$ matrix, D, the entries of which are the values returned by the function. The values do not necessarily have to represent distances in a geometric sense: they can be travel time or travel cost ($\text{gasoline} + \text{tolls} + \cdots$). If the minimum-cost tour yields an objective function value of infinity, then no tour is possible, i.e.,

[2]Directed graphs are composed of vertices and arcs and will be denoted by (V, A), while undirected graphs are composed of vertices and edges and will be denoted by (V, E). $\mathcal{R}^+$ will generally be taken to include zero as well as the positive reals; whether zero is included can usually be told from context, and no special mention of zero will normally be made.

the problem is infeasible. The distance function and the corresponding distance matrix do not have to be symmetric, that is, $D_{ij} = D_{ji}$ need not hold for all i and j. When symmetry does hold, the graph can be thought of as undirected and the problem is known as the *symmetric travelling salesperson problem.* A more important additional constraint for modelling the original, informal statement of the problem is to insist that the *triangle inequality* hold:

$$d(v_i, v_k) \leq d(v_i, v_j) + d(v_j, v_k), \quad \forall\, i,\, j,\, k.$$

In other words, while it may be preferable, in the context of the entire problem, to visit v_j on the way from v_i to v_k, the direct route, when considered in isolation, cannot be bettered. The importance of this property derives from the fact that any distance matrix can be transformed into one that obeys the triangle inequality: it is enough to allow multiple visits to a vertex, thereby letting the new distance from v_i to v_k equal the length of the shortest path from v_i to v_k (instead of the length of the arc from v_i to v_k). In terms of a real salesperson travelling between cities, a multiple visit can be regarded as "passing through on the way to someplace without stopping," a liberty that one would not be denied in practice. Thus the optimal tour obtained from the new matrix, although it is a simple cycle in terms of the new "arcs," may in fact pass through the same vertex more than once, since each new arc corresponds to a path in the original graph. Indeed, this path will have to be stored for each new arc in order to print out the optimal tour in terms of the original graph, G. Figure 1.3 shows an instance of the symmetric travelling salesperson problem with triangle inequality. The graph at the top shows only the direct connections, with the matrix below formed from the shortest paths from each vertex to every other vertex. The optimal tour, starting at Washington, D.C., goes successively to Baltimore, Philadelphia, New York, Buffalo, Detroit, Cincinnati, Cleveland, Pittsburgh, and finally back to Washington, D.C., for a total distance of 1790 miles.

There is a related graph traversal problem, which those of us who always seem to be late to engagements and appointments are continually called upon to solve—the shortest-path problem.

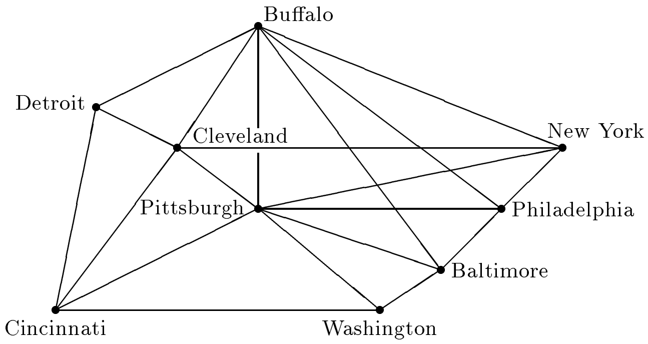
Problem 4 [Shortest Path] Given a directed graph, $G = (V, A)$, of n vertices, a distance function, $d\colon V \times V \rightarrow \mathcal{R}^+ \cup \{\infty\}$, and two vertices, $v_a, v_b \in V$, find a path, $v_a = v_{i_1} \rightarrow v_{i_2} \rightarrow \cdots \rightarrow v_{i_m} = v_b$, which minimizes

$$\sum_{j=1}^{m-1} d(v_{i_j}, v_{i_{j+1}}),$$

i.e., find a path of minimum length from v_a to v_b. □

Note that a path of minimum length need not be a path with the fewest arcs.

A variant on this problem is the *all-points shortest-paths problem,* which is simply the problem above applied to all pairs of vertices. We have already encountered this problem in converting the general travelling salesperson problem to one in which the triangle inequality holds.



(a) The graph, showing only direct connections

Baltimore	0	345	514	355	522	189	97	230	39
Buffalo	345	0	430	186	252	445	365	217	384
Cincinnati	514	430	0	244	265	670	589	284	492
Cleveland	355	186	244	0	167	507	430	125	356
Detroit	522	252	265	167	0	674	597	292	523
New York	189	445	670	507	674	0	92	386	228
Philadelphia	97	365	589	430	597	92	0	305	136
Pittsburgh	230	217	284	125	292	386	305	0	231
Washington	39	384	492	356	523	228	136	231	0

(b) The distance matrix

Figure 1.3: An Instance of the Symmetric Travelling Salesperson Problem with Triangle Inequality

Problem 5 [All-Points Shortest Paths] Given a directed graph, $G = (V, A)$, of n vertices and a distance function, $d: V \times V \rightarrow \mathcal{R}^+ \cup \{\infty\}$, find the shortest path from each vertex in V to every other vertex in V. □

Another problem of the same general type, but with a somewhat different twist, is the minimum spanning tree problem.

Problem 6 [Minimum Spanning Tree] Let $G = (V, E)$ be an undirected, connected graph of n vertices and let $d: E \rightarrow \mathcal{R}^+$ be a distance function defined on the edges. Find $E' \subseteq E$ such that $G' = (V, E')$ is still connected and $\sum_{e \in E'} d(e)$ is minimized. □

We claim that the edges composing the optimal E' form a tree.

The word *tree* is used here in a sense familiar to graph theorists and mathematicians, and not in the sense typically associated with computer science, where it is often equated with a binary tree, which has a designated root and directed edges.

Definition 1.2 A *free tree* (or just a *tree*) is an undirected, connected graph with no cycles. □

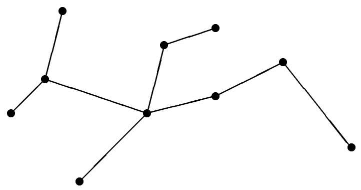


Figure 1.4: An Example of a Free Tree

Figure 1.4 highlights several aspects of this definition. Notice that there really is neither a root nor a parent-child relationship and that the number of edges incident upon a vertex is variable. The definition does not even require the number of vertices in the graph to be finite, although we shall always assume finiteness. With this assumption, we have the following result:

Theorem 1.2 The following are equivalent:

1. G is a free tree (with n vertices).

2. G is connected, but if any edge is deleted, G becomes disconnected.

3. G is connected and has $n - 1$ edges.

4. G has no cycles, but the addition of an edge between any two vertices of G creates a cycle.

5. G has no cycles and has $n - 1$ edges.

6. There is a unique simple path connecting any two vertices of G. $\square$

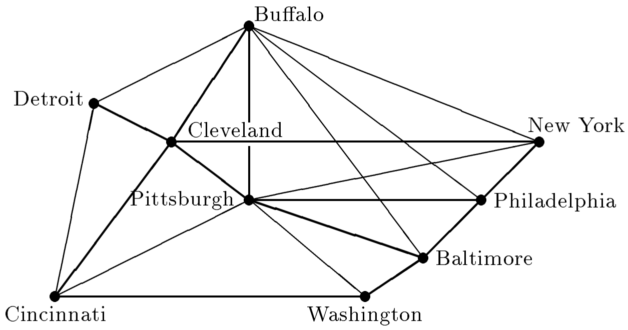


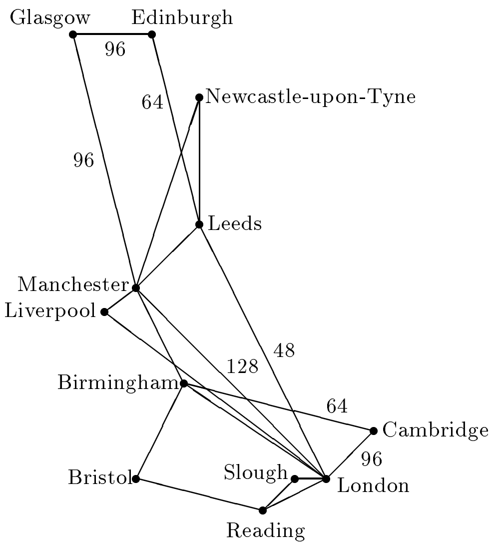
Figure 1.5: The Graph of Figure 1.3 with Its Minimum Spanning Tree

Proving this result would be too much of a digression at this point, but the reader is encouraged to do so, if only in an informal manner.

The optimal set of edges, E', must leave the vertices connected, as required by the problem statement, and cannot contain a cycle, since removing any edge from a cycle will produce a set of lower cost that nonetheless connects all vertices. Thus $G' = (V, E')$ is a tree by Definition 1.2. Any set of edges that connects all of the vertices of a graph is termed spanning. Taken together, these two observations explain the name given to the problem.

Figure 1.5 shows the graph of Figure 1.3 with the minimum spanning tree superimposed. While it is possible to go from any vertex to any other, the route can be quite circuitous, such as the path from Cincinnati to Washington, D.C., which has a length of 638 miles, compared with the direct route of 492 miles. Minimum spanning trees are effective models for certain applications—although road building is not one of them. Consider, for example, a chain of department stores located in the Mid-Atlantic region and the Ohio valley. The management has decided to implement a distributed database system with computing facilities located in the cities shown in Figure 1.5. They intend to lease telephone lines to connect the computing facilities, where the charge for the leased line is a linear function of the distance. By selecting the edges that compose the minimum spanning tree, they will connect all facilities at minimum cost.

Let us now change our viewpoint slightly. Consider the problem of conveying material or data from one point to another in a labelled graph. Each edge corresponds to a section of pipe or transmission line of limited carrying capacity; the labels, which are positive real numbers, represent these capacities. In its full generality, the problem statement allows for two different capacities to be associated with the same edge, one for each direction of flow. In most applications, these two capacities are the same (the graph is effectively undirected) or one of the capacities is zero (the graph is effectively directed). Each vertex is a *source* (generating materials or data), a *sink* (consuming material or data), or a simple interconnection point. Our problem is to determine how to maximize the flow of material or



(Capacities are 56 kbps unless otherwise indicated.)

Figure 1.6: The British Public Data Transmission Network (PSS) *circa* 1981

data from one given point to another. This problem is an appropriate model for gas, oil, and water distribution networks, as well as for many data transmission applications.

Most data transmission networks use a transmission technique known as packet switching, in which a message is broken into packets and the individual packets are sent independently. Each packet contains a sequence number (to merge it correctly with the other packets when reconstituting the message at the destination) and a destination code; routing is done by each node in the network, so that different packets may be sent concurrently along different paths. The question then arises: What is the maximum amount of data that can be transferred from one network node to another, making the best possible use of concurrency? To illustrate the problem, consider the British public data transmission network, PSS, in its early stages (*circa* 1981), as illustrated in Figure 1.6.[3] Assuming perfect routing, the maximum rate of data transmission from Cambridge to Glasgow is 160 kbps (kilobits per second), using several links in parallel, as shown in Figure 1.7. This problem can be formalized as follows.

Problem 7 [Network Flow] Let $G = (V, E)$ be an undirected, connected graph of n vertices and let $c\colon V \times V \to \mathcal{R}^+$ be a capacity function. (This formalism allows for different capacities for the two directions of flow through an edge; by

[3]The line transmission capacities have been altered somewhat to make the problem more interesting: as a rule, data networks use the same type of line throughout.

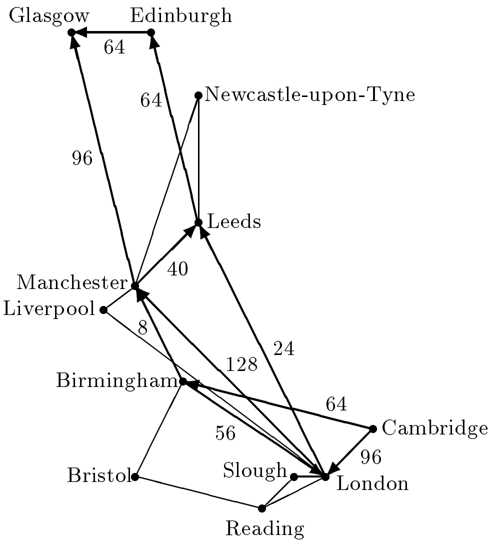


Figure 1.7: The Graph of Figure 1.6 with a Maximum Flow from Cambridge to Glasgow

convention we let $c(v,w) = 0$ when there is no edge connecting v and w.) Let $s, t \in V$ be distinguished vertices (the source and the sink). A flow from s to t is a real-valued function on pairs of vertices which obeys five conditions, for all vertices v and w:

1. The flow assigned to an arc does not exceed its capacity: $-c(w,v) \le f(v,w) \le c(v,w)$.

2. The flow through an edge is positive when seen from one end and negative when seen from the other: $f(v,w) = -f(w,v)$; in other words, f is skew-symmetric.

3. Material is conserved at all vertices except s and t: $\sum_w f(v,w) = 0$ for all vertices v other than s and t.

4. There is no flow into the source: $f(s,w) \ge 0$.

5. There is no flow out of the sink: $f(w,t) \ge 0$.

The *value* of a flow from s to t is simply the value of the net flow out of s, $\sum_w f(s,w)$. We want to find a flow from s to t of maximum value. □

In thinking about these conditions, it is best to think of $f(v,w)$ as the amount of flow from v to w. A negative value for $f(v,w)$ means that the actual, physical flow of material is really from w to v. This is why $f(v,w)$ can be as negative as

$-c(w, v)$ (the flow is at its maximum, but the direction of flow is from w to v); why $f(v,w) = -f(w,v)$ (it is just a change of perspective); and why $\sum_w f(v,w) = 0$ represents conservation of material. The fourth and fifth conditions can be enforced by the convention that $c(v, s) = c(t, w) = 0$ for all v and w.

Exercise 1.2 Prove that the net flow out of the source, $\sum_w f(s, w)$, is equal to the net flow into the sink, $\sum_v f(v, t)$. □

The path problem underlying the network flow problem is apparently complicated by the infinite number of possible flow assignments: this is the first problem we have encountered where the solution space is infinite.

1.3 The Efficiency of Elementary Calculations

How efficiently a certain task can be performed is a central theme of computer science. Here we consider problems, the origins of which are not rooted in classical computer science, but rather stem from elementary mathematics.

Problem 8 [Exponentiation of Integers] Let x and n be natural integers. How should x^n be computed in order to minimize the number of multiplications performed? □

The problem statement just given is sufficiently open-ended that it permits a number of solutions that are not really intended. First, since x is an integer, it is reasonable to presume that x^n is to be computed exactly. Methods for computing x^n approximately, appropriate when x is real, and possibly already represented inexactly inside the computer, such as executing the FORTRAN code

```
Y=EXP(N*LOG(X))
```

corresponding to $e^{n \ln x} = (e^{\ln x})^n = x^n$, are not being considered here. More in the spirit of integer arithmetic, but still not intended, is to compute x^{2047} as x^{2048}/x. This method is effective because x^{2048} can be computed using only 11 multiplications when constructed by the sequence

$$x, \quad x \cdot x = x^2, \quad x^2 \cdot x^2 = x^4, \quad x^4 \cdot x^4 = x^8, \quad x^8 \cdot x^8 = x^{16}, \ldots$$

How to judge the "cost" of computing x^{2047} in this manner is not clear; divisions are normally considered to be more time-consuming than multiplications. In general, for an exponent of the form $n = 2^m - 1$, this method takes m multiplications and one division. If methods like this, which are strictly integer (the division leaving no remainder), are to be considered invalid, a precise set of rules must be given. We shall permit only multiplications under the following rules:

1. A fixed pool of memory locations is available; initially, all memory locations have undefined values, with the exception of one location, which contains the value x.
2. Multiplication must be performed by taking the values from two defined memory locations, multiplying them, and putting the result into a memory location. The operands may come from the same defined memory location, and either a defined or an undefined memory location can receive the result of the multiplication.

If only two storage locations are allowed, one in which x itself is held and one in which the answer is being accumulated, then the optimal way to compute x^{15} is to form the sequence

$$x, \quad x \cdot x = x^2, \quad x^2 \cdot x = x^3, \quad x^3 \cdot x^3 = x^6,$$
$$x^6 \cdot x = x^7, \quad x^7 \cdot x^7 = x^{14}, \quad x^{14} \cdot x = x^{15}.$$

There really is very little choice in the matter. Each computation must correspond to

```
answer := answer*answer;    or
answer := answer*x;         or
answer := x*x;
```

where `answer` holds the result as it is being accumulated. This sequence requires six multiplications. The ability to store an indefinite number of intermediate results and to use them later with no cost penalty reduces the number of multiplications to five, since x^{15} can then be computed as

$$x, \quad x \cdot x = x^2, \quad x^2 \cdot x = x^3, \quad x^3 \cdot x^2 = x^5,$$
$$x^5 \cdot x^5 = x^{10}, \quad x^{10} \cdot x^5 = x^{15},$$

which requires that two intermediate values, x^2 for the computation of x^5 and x^5 for the computation of x^{15}, be retained. If x need not be preserved, then this method of computing x^{15} still requires only two memory locations. If x must be preserved for some other purpose, then three memory locations are required. This analysis raises the possibility of optimizing a secondary objective function: Among all ways to compute x^n that use the fewest number of multiplications, which uses the least additional storage?

Since multiplying x^p by x^q is equivalent to adding p and q, and since the value of x does not affect our evaluation of the cost, the original problem, after clarifying the legal operations, can be rephrased as shown in Problem 9.

Problem 9 [Shortest Sum Decomposition] Given n, find the shortest sequence of integers, $1 = p_1, p_2, \ldots, p_{k-1}, p_k = n$, such that each integer in the sequence, other than 1, is the sum of two integers already included in the sequence. □

x^{2047} can be computed in 15 multiplications, corresponding to the following sequence:

Value	1	2	3	5	7	14	21	42	47
from		$1+1$	$2+1$	$3+2$	$5+2$	$7+7$	$14+7$	$21+21$	$42+5$

Value	89	178	267	445	890	1780	2047
from	$47+42$	$89+89$	$178+89$	$267+178$	$445+445$	$890+890$	$1780+267$

If only x and one other storage location to accumulate the answer are permitted, then computing x^{2047} requires 20 multiplications. Notice that 15 multiplications would appear more expensive than the computation of x^{2048}/x, but that approach violates our ground rules.

Exercise 1.3 What is the optimal way to compute x^{77}? (The minimum length sequence requires eight multiplications, and it is not unique.) □

Why the ground rules, and where did this problem come from in the first place? In recent years, considerable attention has been paid to *public key encryption*, systems for sending coded messages that have several advantages over earlier cryptographic methods. These advantages include the following:

1. An encrypted communication between two people can be established without a clandestine meeting to exchange secret coding and decoding keys; in fact, the exchange of keys can be done over an insecure communications channel using plain language.

2. Bogus messages from third parties can be detected and disregarded; that is, the authenticity of the sender can be determined. (Not all public key encryption systems incorporate this feature.)

3. Since these systems are based on problems assumed to be intractable, it may be possible to prove that the code cannot be deciphered in any reasonable length of time.

The basic computation required to use the public key encryption system due to Rivest, Shamir, and Adelman is the rapid calculation of $x^e \bmod N$ for x, e, and N positive integers of up to 1000 digits in length. In this system, x is the message (or a portion of it) to be encoded; it can simply be the ASCII representation of the message viewed as one large binary number. Note that x changes with each transmission but that e and N are calculated once and for all. The large size of the numbers implies that an unbounded precision arithmetic package, that stores numbers in multiple words, will be needed, so that a multiplication will take considerable time. Division by x is disallowed for the following reason: if

Table 1.2: The Effect of the Order of Evaluation on the Number of Scalar Multiplications

Order of multiplication	Number of scalar multiplications
$\big((A_1 \times A_2) \times A_3\big) \times A_4$	320,000
$(A_1 \times A_2) \times (A_3 \times A_4)$	620,000
$A_1 \times \big(A_2 \times (A_3 \times A_4)\big)$	160,000
$\big(A_1 \times (A_2 \times A_3)\big) \times A_4$	212,000
$A_1 \times \big((A_2 \times A_3) \times A_4\big)$	52,000

the arithmetic is done modulo N at every stage, performing a division by x is really multiplying by x^{-1}, a value which must be determined at a fairly large cost and which might not even exist (because N is composite). On the other hand, performing the entire computation in $\mathcal{Z}^+$, which permits the division, and taking the result modulo N requires multiplying far larger numbers at each stage, which slows down the overall computation. Since e does not change, but $x^e \bmod N$ is being constantly computed, it pays to spend some time finding the optimal way to compute x^e.

Problem 10 [Matrix Chain Product] Given the matrix computation, $A_1 \times A_2 \times \cdots \times A_n$, where the matrices have compatible dimensions, but need not be square, in what order should the matrices be multiplied to minimize the computation time? □

To avoid some cumbersome details, we shall assume that the matrix multiplications are carried out in the "standard" manner, where if $C = A \times B$, c_{ij} is computed as $\sum_{k=1}^{s} a_{ik} \cdot b_{kj}$, where A is $r \times s$ and B is $s \times t$. This results in $r \cdot s \cdot t$ scalar multiplications for the matrix multiplication, which will be used as the measure of computation time, since the number of additions performed is essentially the same. If the matrices are all square, say $s \times s$, then the order of multiplication is irrelevant and $(n-1) \cdot s^3$ scalar multiplications are required. If the matrices are rectangular, on the other hand, the order of multiplication can make a significant difference. Consider, for example, four matrices with dimensions 100×4, 4×50, 50×20, and 20×100. Table 1.2 shows the five different orders of computation possible and their respective computation times. The worst order requires almost 12 times more work than the best.

If the order of multiplication is thought of as a list of instructions that states the order in which the multiplications are to be carried out, there are six orders of

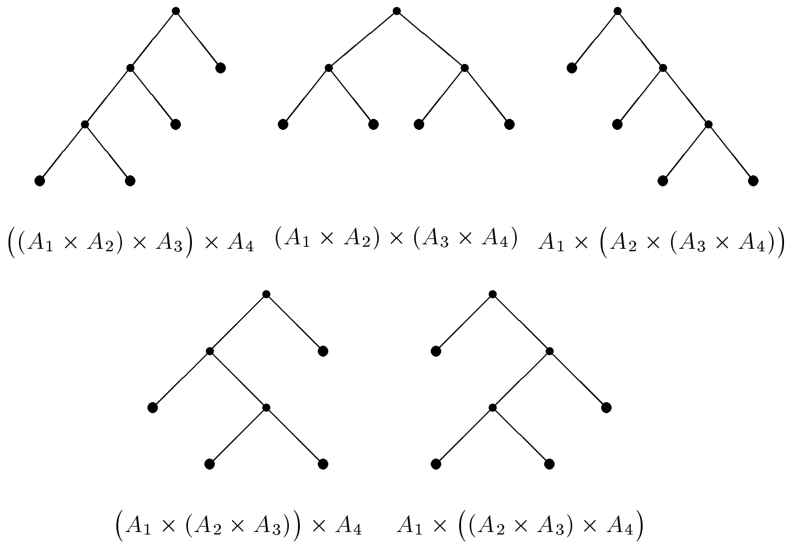


Figure 1.8: Tree Representations of Fully Parenthesized Matrix Products

multiplication, but

$$\begin{array}{lcl} T_1 \leftarrow A_1 \times A_2 & & T_1 \leftarrow A_3 \times A_4 \\ T_2 \leftarrow A_3 \times A_4 & \text{and} & T_2 \leftarrow A_1 \times A_2 \\ T_3 \leftarrow T_1 \times T_2 & & T_3 \leftarrow T_2 \times T_1 \end{array}$$

require the same number of computations, for both are linearizations of the order specified by

$$(A_1 \times A_2) \times (A_3 \times A_4).$$

In order to keep the solution space, each member of which is an order of multiplication, as small as possible, it makes sense to find a representation for an order of multiplication that treats these equivalent sequences as indistinguishable. The fully parenthesized form used in Table 1.2 is one way of representing the computation that ignores the ordering of logically independent subcomputations. Another way of achieving the same effect is to specify a computation as an *expression tree.* Each internal node of the tree is an operator, the only operator here being "matrix multiplication," and each leaf represents an operand, one of the matrices. The five orders of multiplication corresponding to the five parenthesizations are shown in Figure 1.8. Notice that the n leaves are always labelled in the same order, A_1, A_2, $\ldots$, A_n, when viewed from left to right, or more precisely when listed in the order in which they are encountered in a pre-, in-, or postorder traversal of the tree.

It should be apparent that there is a one-to-one correspondence among the fully parenthesized form for the specification of the order of multiplication, the use of expression trees to convey the order, and what we intuitively would call distinct orders of multiplication. Therefore the size of the solution space is equal to the number of binary trees with $n-1$ internal nodes (or n leaves).[4] Table 1.3 shows the number of binary trees with n leaves, b_n, for the first few values of n. Note that, while the values increase only slowly at first, they seem to grow exponentially after that, tripling or more at each step for $n \geq 6$.

While for some problems, such as the knapsack and travelling salesperson problems, only the most elementary principles of counting were needed to determine the size of the solution space, finding a closed form formula for b_n is considerably more difficult. The required mathematics, as well as the derivation itself, will be presented in the next chapter; we content ourselves here with quoting the result,

$$b_n = \frac{1}{n}\binom{2n-2}{n-1}.$$

1.4 Search and Identification Problems

Chances are good that, before you first heard the words *binary search* or wrote your first computer program, you had already discovered this technique for rapidly searching an ordered list, probably while playing the children's guessing game of 20 questions. In one version of this two-person game, one player picks a number between 1 and 1000 and the second player tries to guess the number by asking questions of the form, "Is it bigger than ...?" or "smaller than ...?" or "equal to ...?" If the second player manages to guess the number in 20 or fewer questions, he or she wins; otherwise, he or she loses. Children quickly tire of this game: they soon realize that the first question they ought to ask is, "Is it bigger than 500?"

[4] Different authors define *binary tree* differently. Some insist that every node have either zero or two children, as is the case for expression trees, while others allow zero, one, or two children, with the case of a single left child distinguished from that of a single right child. We use the term *binary tree* to designate either kind, with context or a brief comment specifying which definition obtains.

Table 1.3: The Number of Binary Trees with n Leaves

n	b_n
1	1
2	1
3	2
4	5
5	14
6	42

n	b_n
7	132
8	429
9	1,430
10	4,862
11	16,796
12	58,786

and, based on the answer, next ask about the relationship of the number to 250 or 750.

The code for binary search, given in Program 1.1,[5] is effective for searching large ordered lists since the length of the interval of search, $r - l + 1$, is reduced by a factor of two with every pass through the `while` loop. Thus, for a list of length n, the search time is proportional to $\lg n$. To see just how effective this algorithm really is, imagine a stack of punch cards with the name of every man, woman, and child in the United States, in alphabetical order, on a separate card. The stack would stretch for a distance of 25 miles, yet the card associated with any individual could be retrieved in just 28 fetches.

By placing the binary search technique in a more abstract and general setting, we can see more clearly what it does and perhaps how to improve on its performance. Let us first generalize the problem setting, using as a specific example a program to detect spelling errors in large documents. We begin with a key space, K, which is quite enormous or even infinite. K must have the property, however, that any two keys in K can be compared with the "$<$" relation. In the application we are considering, K would consist of all possible strings of letters, that is, wordlike objects, where $k_1 < k_2$ means that k_1 precedes k_2 in alphabetical order. From K we select a distinguished finite subset, $\hat{K}$, which defines the dictionary of correctly spelled words. The use to which we intend to put $\hat{K}$ is as follows: for every wordlike object in a document, determine whether or not the object is also in $\hat{K}$. If it is, ignore it, but if it is not, then print it out as a potential misspelling. Thus we anticipate searching $\hat{K}$ for different keys time and time again. (Of course, this scheme is not perfect: foreign or technical words and proper names may be absent from $\hat{K}$ and will be printed, and some misspellings will produce legitimate words and not be caught.)

We can apply binary search to the alphabetized $\hat{K}$, but this approach is not entirely satisfactory. With a bit of bad luck, the most common word in English, *the,* will fall into a position on the list that requires the maximum number of passes through the `while` loop of Program 1.1. Indeed, if the dictionary is 250,000 words long, a total of 118,929 words take the maximum number of passes; even if *the* is not among them, other common English words, such as *of*, *and*, and *if*, are sure to be. Conversely, a rare word like *numismatist* is likely to fall exactly at the halfway point, requiring only one pass. Unlike in the 20 questions game, where each integer had the same probability of being selected for the secret number, the elements of $\hat{K}$ in this example are looked up with unequal probabilities. We shall therefore assume that each key, $k_i \in \hat{K}$, for $1 \leq i \leq n$, where $|\hat{K}| = n$, has an associated probability, p_i, that it will be the object of the search. Another feature that distinguishes this search from the game of 20 questions is the possibility of requesting a key that is absent from $\hat{K}$. We define q_i as the probability of a search for a key, absent from

[5]Note that we pass the array `A` as a `var` parameter, in spite of the fact that it is not modified by the procedure; otherwise, the whole point of the algorithm would be lost, because the array would be copied on entry to the procedure, a linear-time operation. On the other hand, we choose to pass the objects to be compared by value, as we assume them to be small.

```
const MaxN = ...;
type datum = ...; (* anything comparable by (generalized) '<' *)
     data = array [1..MaxN] of datum;
function BinarySearch(var (* for speed *) A: data; n: integer;
                          k: datum): integer;
  (* Return location in A of k, if present, or 0 if absent.
     n is the location of the last valid datum in A. *)
  label 1;
  type order = (less, equal, greater);
  var l, m, r: integer; (* left, middle and right *)
        (* The value k, if present, is always located in A[l..r]. *)
  function Compare(first, second: datum): order;
    begin
      (* Determine relationship between first and second. *)
      ... not shown ...
    end; (* Compare *)
  begin (* BinarySearch *)
    l := 1; (* If present, surely in A[l..r] at this point. *)
    r := n;
    while l <= r do (* There still is a potential range. *)
      begin
        (* Find the middle element of A[l..r], more or less. *)
        m := (l+r) div 2;
        case Compare(k,A[m]) of
          less:    r := m - 1; (* If present, then to the left of m. *)
          equal:   begin (* found it *)
                     BinarySearch := m;
                     goto 1 (* return *)
                   end;
          greater: l := m + 1 (* If present, then to the right of m. *)
        end
      end;
    (* Not present in the list. *)
    BinarySearch := 0;
 1:
  end; (* BinarySearch *)
```

Program 1.1: Binary Search

$\hat{K}$, that falls between the adjacent keys k_i and k_{i+1}. The two boundary cases, q_0 and q_n, are the probability of searching for a key preceding k_1 or following k_n. In some problems, the p_i and q_i can be determined *a priori*; for the example at hand, these values would have to be determined empirically by gathering frequency statistics from a number of large documents. (We shall ignore the fact that many of the p_i will be so small as to be meaningless and that the q_i probably cannot be determined with any accuracy.) Since the p_i and q_i represent probabilities of successful and unsuccessful search, respectively, and between themselves account for all keys in K, we must have

$$\sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_i = 1.$$

Given that we have K, $\hat{K}$, the p_i, and the q_i, the question is: How do we organize the search to minimize the average search time? As with the problem of rapid exponentiation of integers, this question is also too open-ended. For instance, if hashing is permitted, we might achieve constant average search time. On the other hand, if we restrict ourselves to the binary search procedure, there is no problem at all because, given $\hat{K}$, there is only one way to apply this procedure. We need an appropriate generalization.

Definition 1.3 A *binary search tree* is a binary tree (of the zero, one, or two children variety) in which there is a one-to-one correspondence between the keys of $\hat{K}$ and the nodes of the tree and in which every node has the following property: if k is the key associated with the node, k_1 is any key associated with a node in its left subtree, and k_2 is any key associated with a node in its right subtree, then $k_1 < k < k_2$. □

A binary search tree containing the names of the original 13 states is shown in Figure 1.9. Notice that, if a binary search tree is traversed in inorder, the nodes are visited in increasing order with respect to the "$<$" relation.

The process of searching a binary search tree for a specified value is essentially the same as searching an array of ordered values with binary search, as illustrated in Program 1.2. In searching for Massachusetts, we first try the root; since Massachusetts precedes New Hampshire, we go down the left branch; then, since Massachusetts follows Connecticut, we go to the right, where we compare it with Maryland and go right again; finally, we find Massachusetts on the fourth probe of the data structure. While the process is the same, there is an important difference: in binary search we reduce the search space by half with each test. For an arbitrary binary search tree, the gain may be much smaller—in Figure 1.9, New Hampshire creates a 5 *vs.* 7 split, while Connecticut splits its two subtrees 0 *vs.* 4.

Not only is the search algorithm of Program 1.2 analogous to the search procedure of Program 1.1, but also the behavior of binary search is emulated exactly if the tree is constructed according to the following rule:

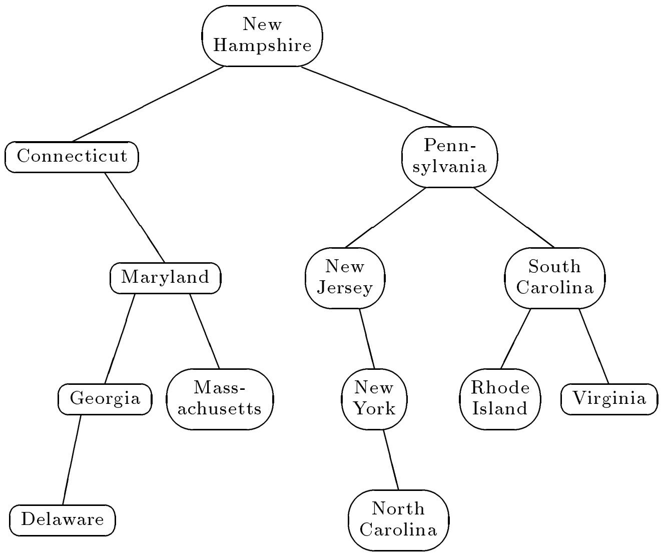


Figure 1.9: A Binary Search Tree for the Original 13 States

> Select as the root of the tree and every subtree the key which divides the remaining search interval most nearly into two equal pieces, giving the extra key to the right subtree in case the original search interval had an even number of keys.

Whenever a move is made down the right subtree in Program 1.2, binary search adjusts `l` in Program 1.1, etc. For the original 13 states, the tree that is equivalent to binary search is shown in Figure 1.10.

Binary search never performs poorly, because the associated binary search tree is balanced fairly evenly and the height of the tree is $\lfloor \lg n \rfloor + 1$. On the other

```
   p := root;
   while p <> nil do
     case Compare(k,p^.data) of
       less:    p := p^.left;  (* equivalent to r := m - 1 *)
       equal:   goto 1;
       greater: p := p^.right  (* equivalent to l := m + 1 *)
     end;
   (* p = nil; no node with key equal to k *)
1: ...
```

Program 1.2: Searching a Binary Tree

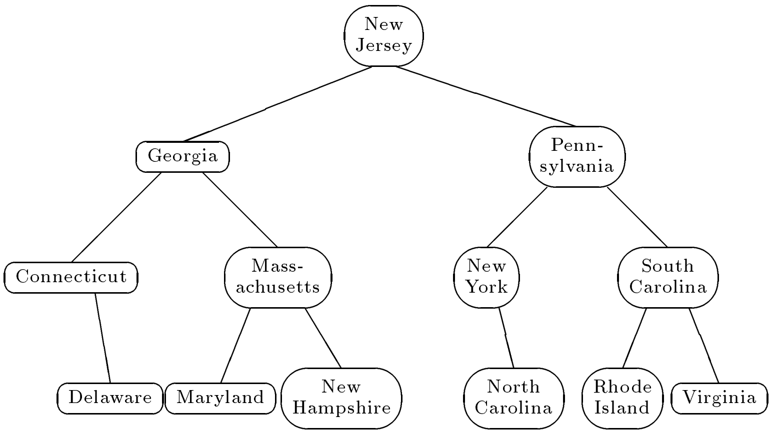


Figure 1.10: The Binary Search Tree Dictated by Binary Search

hand, as the spelling error detection program suggests, binary search is completely insensitive to the probability distribution of search requests inherent in the data—the rule just given for constructing the binary search tree equivalent to binary search concerns itself with only the number of data items. Therefore, it might be possible to find a somewhat skewed binary search tree that exhibits better performance characteristics by placing items that are frequently requested closer to the root.

So that we can be precise about what we mean by one binary search tree being better than another, we need to define an objective function. Intuitively, we want to minimize the average execution time of the search described in Program 1.2 under the probability distribution specified by the p_i and q_i. Exact analysis is difficult, because the time through the `while` loop depends on whether one or two tests (the one in the `while` and the one in the `case`) are performed. In order to simplify our work, we assign unit cost to the test of the `case`, but ignore the work done in the test of the `while`. This charging policy can be viewed as a crude model of the situation where the binary search tree is kept on disk. Checking whether a pointer is `nil` is done in main memory and so is very fast; similarly, once the data at `p^` have been retrieved, performing the `case` test is irrelevant. The disk access time dominates all other computations.

Let the root be at depth 1; if a node at depth j is the object of the search, we incur a cost of j. Since node k_i is the object of the search p_i percent of the time, it contributes an average cost of $p_i \cdot \text{depth}(k_i)$. An unsuccessful search presents a slight difficulty, because it does not correspond to nodes of the tree; to remedy this situation, we imagine that the `nil` pointers are replaced by a special type of node, the key of which matches anything in the range $k_i < k < k_{i+1}$ (or $k < k_1$ for q_0 and similarly for q_n). Calling such a node k'_i, the cost of an unsuccessful search is now $q_i \cdot (\text{depth}(k'_i) - 1)$, where the -1 is due to the fact that determining that we are at node k'_i is really the test `p <> nil`, for which we agreed not to incur any cost. Thus, for any binary search tree, the cost of the tree equals the average search length:

$$\sum_{i=1}^{n} p_i \cdot \text{depth}(k_i) + \sum_{i=0}^{n} q_i \cdot (\text{depth}(k'_i) - 1).$$

The binary search trees of Figures 1.9 and 1.10 have costs of 3.62 and 3.45, where we have used for p_i the population of the state relative to the population of all 50 states according to the 1980 census. The q_i are the percentages of people who lived in the states alphabetically between k_i and k_{i+1}. Since no state falls alphabetically between Maryland and Massachusetts, $q_4 = 0$. On the other hand, q_9 is 0.0757, since North Dakota, Ohio, Oklahoma, and Oregon fall between North Carolina and Pennsylvania.

We can now pose the following problem.

Problem 11 [Optimal Binary Search Tree] Given K, $\hat{K}$, $\{ p_i \mid 0 \leq p_i \leq 1, i = 1, 2, \ldots, |\hat{K}| \}$, and $\{ q_i \mid 0 \leq q_i \leq 1, i = 0, 1, \ldots, |\hat{K}| \}$, with $\sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_i = 1$,

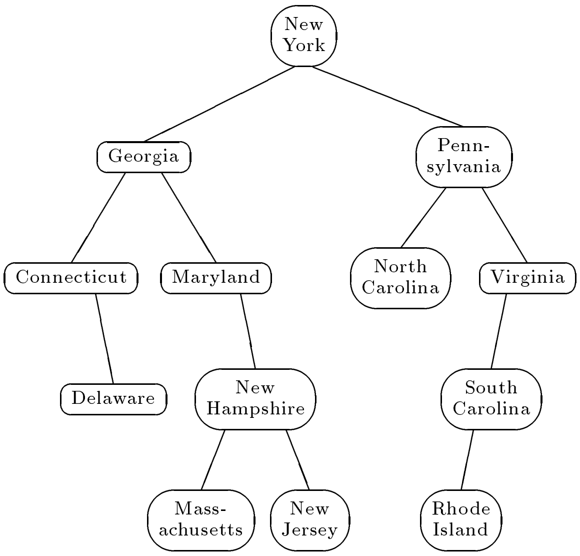


Figure 1.11: The Optimal Binary Search Tree with Probabilities Based on 1980 Populations

what is the binary search tree that minimizes

$$\sum_{i=1}^{n} p_i \cdot \text{depth}(k_i) + \sum_{i=0}^{n} q_i \cdot (\text{depth}(k_i') - 1)\,? \qquad \square$$

The tree of Figure 1.11 is optimal for the data described, with a cost of 3.27.

The use of trees to guide searches is not limited to searching lists, the elements of which come from a totally ordered domain. Consider the problem of a microbiologist attempting to determine which one of eight species from genus *Pasteurella* is present in a specimen brought into the laboratory for identification. He has at his disposal a collection of tests that he can apply to samples of the specimen, such as the motility of the bacteria at 22°C or the ability of a colony to grow on an agar medium in the presence of various chemical compounds. The results of applying these tests to control strains are published in the microbiological literature and can be summarized as in Table 1.4, where "+" indicates a positive response and "−" a negative response. Unless every row of such a table is distinct from every other row, there will be species that are inseparable, because these species behave identically under any and all tests applied to them. For the time being, we shall assume that any two species can always be separated.

Table 1.4: Test Results on Eight Species of *Pasteurella*

	Name of test									
Organism	Motility at 22° C	Haemo-lysis	Indole	Methyl red	Urease	Oxidase	Litmus	Orni-thine	Arab-inose	Cello-biose
P. pneumo-tropica	−	−	+	−	+	+	−	−	−	−
P. hemolytica var. *ureae*	−	+	+	+	+	+	+	−	−	−
Pasteurella multocida	−	−	+	−	−	+	−	+	−	−
P. hemolytica type A	−	+	−	−	−	+	−	−	+	−
P. hemolytica type T	−	+	−	−	−	+	−	−	−	+
Pasteurella X	+	−	−	+	+	−	−	+	+	+
Pasteurella pestis	−	−	−	+	−	−	+	−	+	−
P. pseudo-tuberculosis	+	−	−	+	+	−	+	−	+	−

The microbiologist could proceed by applying all of the tests and then matching the observed results against each row of the table until an exact match is found. However, this procedure is inefficient. Notice that applying any one test will divide the species into two disjoint subsets, based on the reaction of each species to the test. The result of performing the test on the unknown species will narrow the possibilities to those species in one subset or the other. The microbiologist can then select another test to apply, further restricting the possibilities for the unknown, and continue in this manner until the species is identified. There are two subtle points to notice about this procedure:

1. The tests are to be applied sequentially, with the knowledge gained on previous tests guiding the selection of subsequent tests. The test that the microbiologist chooses to apply as the $(i+1)$st test generally depends on the outcome of the first i tests.

2. Unlike when binary search is applied to an ordered list, the two subsets produced cannot always be forced to be of (approximately) equal size. The size of the subsets created by applying a test depends on the data in the published table, and nature does not always conveniently divide things along a 50–50 split.

A testing procedure can be specified by an *identification tree*, such as that shown in Figure 1.12. An identification tree differs from a binary search tree in a few relatively inconsequential ways. The interior nodes now correspond to tests and no longer contain a key value, so that the search cannot terminate before reaching

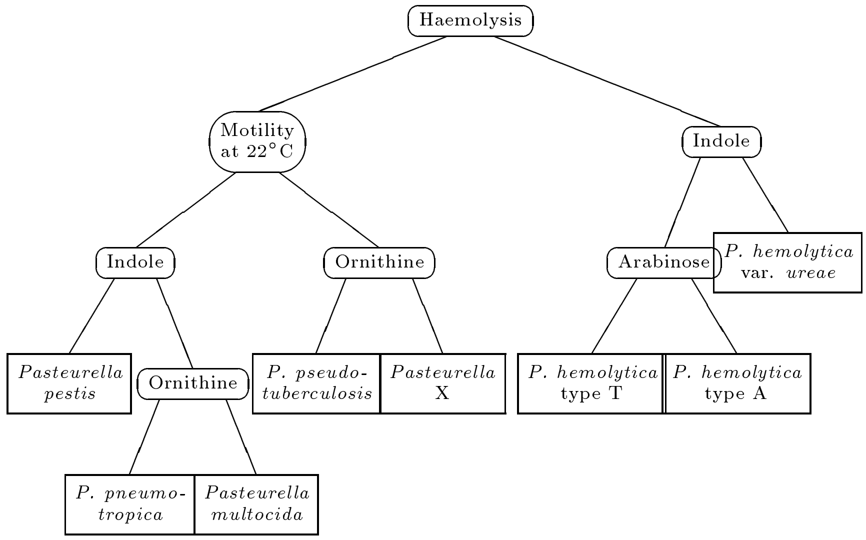


Figure 1.12: An Identification Tree for Eight Species of *Pasteurella*

a leaf. The `nil` pointers have been replaced by nodes that involve no testing but that specify the name of the identified object, here a species of *Pasteurella*.

While technically an interior node can have only one child, applying such a test would provide no useful information, since having only one child corresponds to not splitting a subcollection further. Just as in the case of binary search trees, we may associate a probability, p_i, with each species: the probability that this species will be brought into the laboratory for identification. These probabilities are presumably determined by gathering statistics over a period of time. There is no analogue to the q_i, as every identification will terminate at a leaf. As an unfortunate consequence, if the unknown species is not any of those listed in the table, a misdiagnosis will result. The code of Program 1.2 must undergo minor modifications to accommodate our new problem.

Problem 12 [Optimal Identification Tree] Given that objects drawn from m categories, $O_1, O_2, \ldots, O_m$, with relative frequencies $p_1, p_2, \ldots, p_m$ (with $\sum_{i=1}^{m} p_i = 1$) must be identified with the help of n dichotomous tests, each of which is applicable to any object and is specified by an $m \times 1$ vector of outcomes, find the identification tree that minimizes the average path length

$$\sum_{i=1}^{m} p_i \cdot (\mathrm{depth}(O_i) - 1),$$

where $\mathrm{depth}(O_i) - 1$ is the number of tests that must performed to identify an object in category O_i. $\square$

If the eight species of *Pasteurella* are equiprobable, the identification tree of Figure 1.12 is not optimal.

1.5 Puzzles and Decision Problems

Every so often combinatorial games and puzzles appear in toy stores and for a short period of time become the rage, only to be relegated to attics and garages when their owners become sufficiently frustrated with them. The number of puzzles of this type is truly staggering. We present here four such puzzles, which hold some theoretical interest and have the advantage that they can be expressed easily in the two-dimensional format of a book.

Problem 13 [n-queens] The queen in the game of chess attacks in the horizontal, vertical, and diagonal directions, as shown in Figure 1.13. How, if at all, can n queens be placed on an $n \times n$ chessboard so that no two queens are mutually attacking? □

A few minutes of thought should convince the reader that there are no solutions to the two-queens and three-queens problems. A solution to the four-queens problem is shown in Figure 1.14.

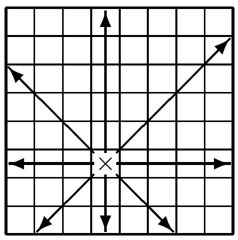


Figure 1.13: The Lines of Attack of a Queen on an 8×8 Chessboard

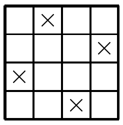
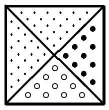


Figure 1.14: A Solution to the Four-Queens Problem

Problem 14 [Tiling a Square] A tile is a square with a side of unit length divided into four independently colored quarters, as shown below:

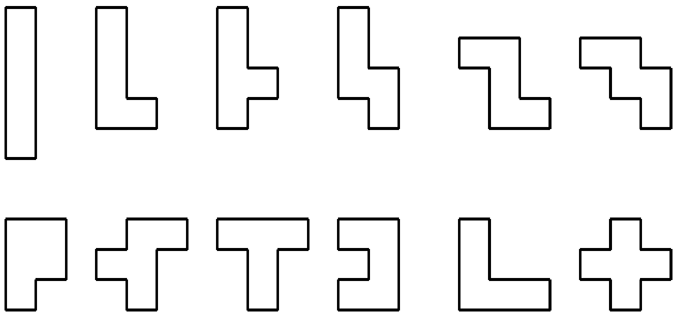


(The back of the tile is blank.) There are m different tile types, i.e., color arrangements, and t_i tiles of type i available, with

$$\sum_{i=1}^{m} t_i = n^2.$$

How, if at all, can the tiles be placed in an $n \times n$ square so that, whenever two tiles have a border in common, they have the same color along that border? The tiles may be rotated, they need not agree in any sense if they share only a point in common, and any colors can be on the boundary of the $n \times n$ square. □

Problem 15 [Pentominoes] Five unit squares can be connected into rigid shapes in exactly the following 12 ways:



Since $12 \times 5 = 60$, how, if at all, can the 12 shapes be placed into a 6×10 rectangle so that it is completely covered? □

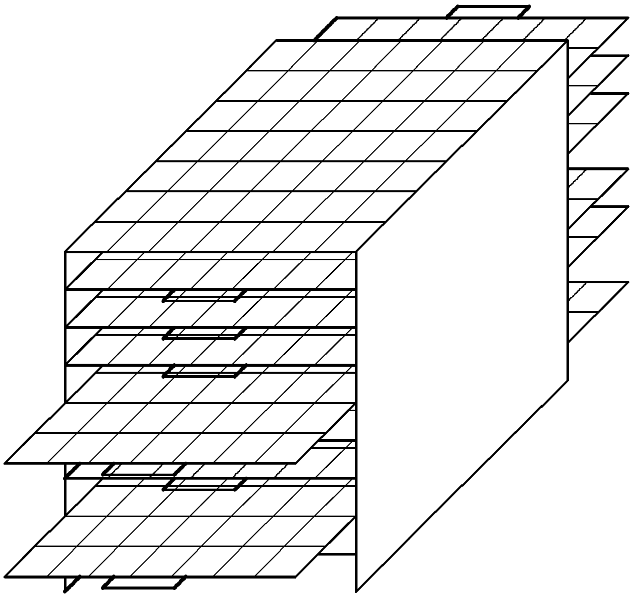
Problem 16 [Peek] This is a two-person game played with a stack of perforated plates placed in a fixed order in a rack. Each plate has one or more holes centered at integer coordinates. All plates have the same dimensions, with their length exceeding the length of the rack, and all holes have the same diameter. A plate can be in one of two positions in its slot: flush on one side or flush on the other. The top of the rack is itself a plate, but it is shorter and fixed in place. (Figure 1.15 shows a sample configuration for the game.) Each player knows the perforation pattern of all plates and can observe the other player's moves. Each plate is marked "first" or "second," and players can move only their own plates. Players take turns sliding their plates back and forth until one player succeeds in aligning holes so that one can "peek" clear through the stack. The problem is to determine, given an initial configuration, whether the first player can force a win. □

These problems, unlike the earlier ones in this chapter, are not usually viewed as optimization problems. What is desired here is to exhibit one or more solutions or, perhaps, to decide whether or not a solution exists. In general, we can formulate the same problem in several ways by slight rephrasings; we distinguish the following forms:

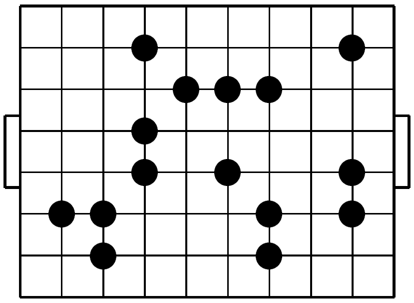
1. A *decision* problem has a yes or no answer. Typical questions are, "Does a solution exist?" or "Is there a sufficiently good solution?"
2. In a *search* problem, one must exhibit a solution. Puzzles are search problems. Optimization problems are search problems, too: it does not do the travelling salesperson much good to know the length of the optimal tour while remaining ignorant of which tour has this length.
3. In an *enumeration* problem, one must exhibit all solutions. Many enumeration problems may require long running times simply because the number of solutions is very large; to avoid this constraint, an enumeration problem is often defined as requiring that all distinct solutions only be counted. The latter requirement might be met implicitly—which could prove much faster than counting solutions one by one.

The initial reaction to decision problems is often to dismiss them as hidden search problems (and thus just as difficult). How, in general, are we to solve a decision problem, except to solve the corresponding search problem, then emit yes or no? While it is true that, for most decision problems, solving the corresponding search problem is the only technique that suggests itself, this need not always be the case: mathematics does allow nonconstructive existence proofs.

As an example of this distinction, consider the following two-person game, which is a generalization of the game of *hex*. (The reasoning to follow applies to any two-person game where an extra piece or an extra move can only help the player who has it available.) The game board consists of a connected, undirected graph, $G = (V, E)$, with two distinguished vertices, s and t. The players alternate turns, coloring a vertex from $V - \{s, t\}$ that has not already been colored, until all vertices



(a) A box with eight movable plates, six slid in and two slid out



(b) A typical plate

Figure 1.15: A Sample Configuration of the Peek Game

are colored. Let the first player color vertices green and the second color them red. If the first player can form a path from s to t passing through only green vertices and the second player cannot form a path from s to t passing through only red vertices, then the first player wins. A win for the second player is similarly defined. If neither or both players can form paths from s to t passing through only vertices of the proper color, then the game is a draw. Consider the decision problem: given G, s, and t, and assuming perfect play by the first player, can he or she force a win or a draw? We can prove that the answer to this question is always yes—although the proof is a little hard to accept on first reading. The argument, a proof by contradiction, establishes that the second player cannot win. *Ad absurdum*,[6] suppose that there exists an instance of the game in which the second player can win according to some strategy. The first player can effectively adopt the second player's strategy as follows. For the first move, he or she chooses a vertex randomly and "forgets" that he or she has it. The second player then moves. The first player then chooses his or her next vertex using the second player's strategy, treating the second player's first move as if it were the opening move of the game. This style of play continues until the first player, according to the second player's strategy, must pick the vertex that he or she originally chose and forgot. If this occurs, and there are any vertices left, he or she chooses his or her next vertex randomly, making it the new forgotten vertex. Play continues in this manner. At the end of the game, the first player holds all of the vertices that the second player would hold according to the latter's strategy, plus possibly one more. Having an extra vertex cannot harm a winning position, so the first player wins, contradicting our hypothesis. Thus the decision problem is solved, yet we do not have the slightest inkling as to what is the perfect strategy! Note that the fact that the answer to the decision problem is yes for all instances of the game plays no role in the proof; other games exist for which the decision problem is also solvable without search and where the answer does vary with the instance.

One of the most important decision problems concerns Boolean expressions; it has applications in automated theorem proving, logic circuit verification (since each circuit implements a Boolean function), and system testing (in which we are interested in only Boolean values: Does the system work or does it not?). It is also fundamental in the study of problem complexity (see Volume II).

Problem 17 [Satisfiability] Given a Boolean expression in *conjunctive normal form*—that is, as a conjunction (logical "and") of clauses, each of which is a disjunction (logical "or") of literals (variables or their complements)—can the variables be assigned truth values that make the expression evaluate to true? □

For instance, the expression

$$(x \vee z) \wedge (\bar{x} \vee \bar{y} \vee \bar{z}) \wedge (\bar{x} \vee y) \wedge (\bar{z}),$$

[6]Literally, "towards absurdity," i.e., in order to obtain a contradiction.

where $\bar{x}$ denotes the complement of x, is satisfiable (the truth assignment $x = true$, $y = true$, $z = false$ makes the expression evaluate to *true*), but the expression

$$(\bar{x} \vee y) \wedge (x \vee \bar{y}) \wedge (\bar{x} \vee \bar{y}) \wedge (x \vee y)$$

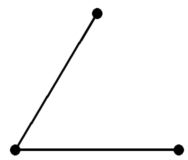
is not. Like generalized hex, it is conceivable that the satisfiability of an expression can be determined nonconstructively, i.e., without producing a satisfying truth assignment; however, once again, most of us would approach this problem as a search problem.

We also note that this problem, like many other decision problems, is easily rephrased as an optimization problem: find the truth assignment that maximizes the number of clauses that evaluate to *true*. In fact, this slight transformation is only one of several among the problem forms. Clearly, a search problem can be turned into an enumeration problem. An optimization problem is easily converted into a decision problem by adding one parameter, a bound assigned to the objective function, and asking whether there exists a solution for which the objective function exceeds the bound. A search problem can also be converted into an equivalent decision problem by means other than changing the question from "show a solution" to "does a solution exist?" For instance, one can convert the search version of satisfiability (find a truth assignment, if any, that makes the expression evaluate to *true*) into the pure decision problem of *tautology* (is a Boolean expression true under all possible truth assignments?) by transforming the original expression, (E), into the new Boolean expression $(E \rightarrow \mathit{false})$. The new expression is a tautology if and only if E can never be satisfied; in particular, a solution to the tautology problem would provide a (presumably) nonconstructive solution to the satisfiability problem.

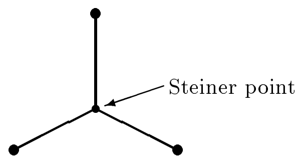
1.6 Geometric Problems

With the increasingly widespread use of graphics and the advent of computer-aided design and manufacturing (better known as CAD/CAM), geometric problems are receiving more and more attention. Most geometric problems consist of an underlying set or graph problem, where the set or graph is *embedded* in Euclidean space. An embedding of a set or graph in n-dimensional space is a function that assigns an n-tuple of coordinates to each element or vertex. The embedding automatically induces a distance measure between the set elements or the graph vertices. Thus most set or graph problems can give rise to geometric problems; however, we shall reserve the term for those problems that depend on geometry for their existence, such as the problems introduced in this section.

Let us consider once again the problem of a communication network, which we discussed in Section 1.2, but let us shift our focus from the use of such a network to its design. Given a collection of cities, we are to construct the least expensive network that will allow communication between any two cities. Let us define the



(a) A minimum spanning tree



(b) The optimal interconnection tree

Figure 1.16: Interconnection Trees for Three Cities

cost of any such network as the sum of the lengths of its communication links. From our discussion in Section 1.2, the reader may be tempted to propose using a minimum spanning tree. That the least expensive network is indeed a tree is easily proved: if it were not, then a cycle would exist; removing any edge of the cycle would produce a less expensive, and still connected, network. However, the minimum spanning tree only rarely provides an optimal solution. In most cases, it is advantageous to add extra nodes (relay stations) to the network. Figure 1.16 presents a simple example with three cities, where the length of the minimum spanning tree is $2/\sqrt{3}$ times that of a tree using an additional node. A tree that spans all the given vertices and possibly includes additional vertices is called a *Steiner tree* and the additional vertices are called *Steiner points*. We can now formulate our first geometric problem.

Problem 18 [Geometric Steiner Tree] Given a set of points, S, in the plane, find a Steiner tree of minimum total length for S. □

The search space for this problem appears infinite, because we can locate the Steiner points at any real coordinates. However, Steiner points must satisfy a number of strong conditions; as a result, the choice of locations remain finite (see Exercise 1.14).

Of particular importance in both linear programming and computational geometry is the notion of a convex domain or region.

Definition 1.4 A set of points in space is said to be *convex* if and only if, given any two points, p_1 and p_2, in the set, all points on the line segment $\overline{p_1 p_2}$ are in the set as well. □

Examples of convex domains include half-spaces, (hyper)planes, the interiors of (hyper)spheres, and the interiors of regular polygons. Since convex domains are closed under intersection, the feasible region of a linear programming problem is itself convex, since it is the intersection of half-spaces, each defined by one of the linear constraints. For the same reason, the concept of the smallest convex domain containing a set of points is well defined, because it is just the intersection of all convex domains that contain the given set. This minimal convex domain is called the *convex hull* of the set. Due to the finiteness constraints of computing, we are particularly interested in convex domains that can be specified finitely.

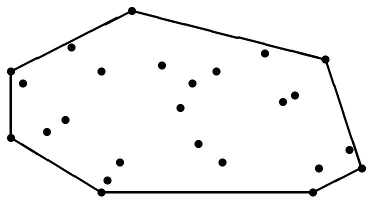


Figure 1.17: The Convex Hull of a Set of Points

Definition 1.5 Given a collection of points, $P = \{p_1, p_2, \ldots, p_k\}$, in n-dimensional space, a point, $p = \sum_{i=1}^{k} \alpha_i p_i$, where each $\alpha_i \geq 0$ and $\sum_{i=1}^{k} \alpha_i = 1$, is called a *convex combination* of the points in P. □

In two dimensions, the set of convex combinations is just the line segment connecting the two points; in general, the set of convex combinations is a convex domain, as a little algebra shows. Furthermore, any convex domain containing the points must contain all of their convex combinations, so that the points describe their own convex hull. The p_i are referred to as the *generators* of the convex set. Note that some of the points of P may themselves be convex combinations of other points in P; such points serve no useful purpose, since their contributions to a convex combination can always be absorbed into the α_i of the other points. Thus there is a minimal subset of generators, no element of which can be generated from any subset of the others. As suggested by Figure 1.17, the generators are the vertices of a polygon which forms the convex hull of the set of points. Although this picture offers only a two-dimensional illustration, we can see that, if the points lie in n-dimensional space, then the minimal set of generators form the vertices of a polytope (the n-dimensional version of a polygon), which is also the convex hull of the set of points. Thus, even though the convex hull of a finite set of points is infinite, it has a finite representation, so that the problem of finding the convex hull is basically a subset search problem.

Problem 19 [Convex Hull] Given a finite set of points in n-dimensional Euclidean space, find the summits of its convex hull. □

While we have discussed several problems concerned with connecting things as tightly as possible, we have not considered the reverse, that is, how to keep things as far apart as possible. One application of this problem derives from environmental considerations: if some pollution-producing plant is to be located within some political jurisdiction, it would be well to place it as far as possible from all human habitation, water sources, and other environmental resources. If each habitation and environmental resource is represented by a point on a map, then the problem becomes one of placing within specified boundaries a vertex that will maximize the distance to its closest neighbor. We formalize this problem as follows.

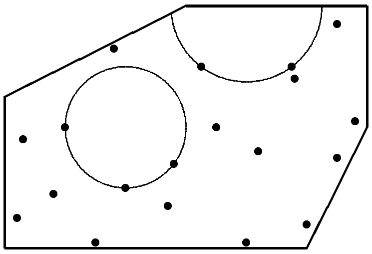


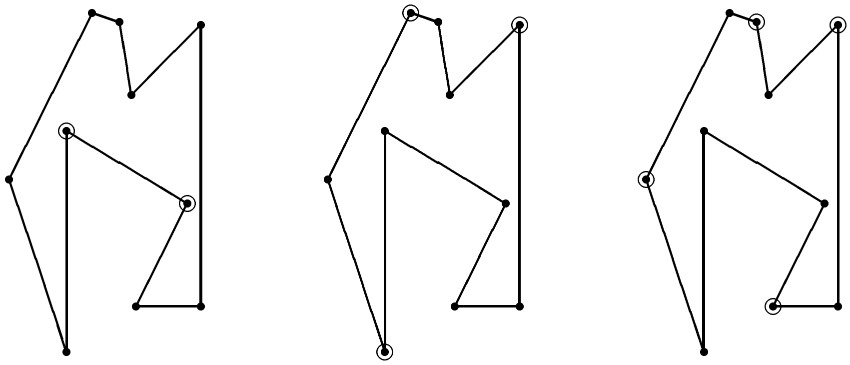
Figure 1.18: An Instance of the Largest Empty Circle Problem, Showing Two Large Empty Circles

Problem 20 [Largest Empty Circle] Given a collection of points in the plane and a boundary enclosing these points, find the largest circle containing none of the points, the center of which lies within the boundary. □

A sample instance is shown in Figure 1.18.

Although the search space may at first appear infinite, a basic result of Euclidean geometry serves to restrict it to a subset search: each circle is fully characterized by three points on its perimeter. Thus we can simply examine each subset of three points, find the center of the circle that they describe, and then determine whether or not that circle is empty. The state space reduces to $\binom{n}{3}$, and the obvious algorithm takes quartic time. However, the algorithm sketched above does not really work, because the center of the circle determined by three points might lie outside the boundary, and thus not be feasible. In particular, it follows that the largest empty circle with its center within the boundary might not touch three obstructions, so that our state space might not be limited after all. In fact, a little more geometric analysis indicates how to handle the cases where the largest empty circle touches only two, or even one, obstruction. Similar work takes care of more possible problems, such as nonconvex regions or even regions with holes (nature preserves, for example, where no development is allowed); the state space remains cubic in size, where n now counts both points and boundary edges. The challenge is to come up with a faster algorithm, that is, an algorithm that uses the geometric nature of the problem in order to consider only a small subset of the $\binom{n}{3}$ circles.

Given the presence of walls (segments) or more general obstructions (solid figures) in the plane, an astoundingly large array of interesting questions may be formulated. Of particular interest are problems of visibility. Consider the design of a prison or fortress; we can assimilate it to a simple polygon (roughly speaking, a polygon with no crossing edges). Both require guards: in the case of a fortress, in order to survey the outside of the polygon; in the case of a prison, in order to watch over both the inside and outside. Towers are commonly built at vertices of the polygonal perimeter, for structural reasons as well as for convenient placement of guards. For a given polygonal enclosure, what is the minimum number of guards, placed at vertices of the polygon, required to keep the entire area (inside or outside)



(a) Art gallery guards *(b) Fortress guards* *(c) Prison yard guards*

Figure 1.19: A Simple Polygon with Optimal Guard Emplacements

under watch? We formalize these problems as follows. A point, p, is *visible* from a vertex, v, of a simple polygon, P, whenever the line segment, $\overline{vp}$, does not cross the perimeter of P. The internal (external) *visibility area* associated with v is the set of all points inside (outside) of P visible from v.

Problem 21 [Art Gallery] Given a simple polygon, P, find a minimal set of vertices of P such that the union of their internal visibility areas covers the inside of the polygon. □

Problem 22 [Fortress] Given a simple polygon, P, find a minimal set of vertices of P such that the union of their external visibility areas covers the outside of the polygon. □

Problem 23 [Prison Yard] Given a simple polygon, P, find a minimal set of vertices of P such that the union of their internal and external visibility areas covers the plane. □

An immediate consequence of our definition of convexity is that a single guard is needed to watch over a convex art gallery; similarly, $\lceil n/2 \rceil$ guards are needed to take care of a convex fortress with n vertices. Thus the problem is of interest for only nonconvex enclosures. Figure 1.19 shows a simple polygon with optimal guard emplacements for all three problems.

1.7 Comments and Additional Problems

By now, the reader will have developed some intuition about the formulation of problems, as well as about the nature of what we have chosen to call combinatorial optimization problems. It may still be difficult to gauge accurately the importance of an abstract formulation; yet the value of a precise formulation should at least be clear. Although we have made an effort to present each problem in a natural manner, as it arises from its real-world constraints, we must acknowledge that we have described very few problems in a complete, realistic manner. The point of the model is the expression of the fundamental combinatorial question present within the problem: there can be no solution to the problem without consideration of this question, although solving this question alone may not be sufficient. The additional constraints and parameters typically complicate the problem (although they may so much constrain it that very few potential solutions are left, so that the design of a solution algorithm becomes easy); consequently, they are often best handled in an *ad hoc* manner through modifications to the algorithm designed to solve the basic underlying question.

In fact, not only do our abstract formulations deliberately omit many parameters and constraints, but they also omit many of the objectives. It is a rare situation in a company or government where a single, quantifiable objective is to be pursued. An airline is not only interested in scheduling its flight crews efficiently, but must also pursue a wide variety of other objectives: customer satisfaction, growth, shareholders' profits, and more. A time-dependent travelling salesperson tour problem may be used to model the flight path of a space shuttle on a satellite maintenance and repair mission; but, even with the inclusion of such parameters as payload, fuel consumption, human reliability, and insurance costs, the tour problem remains a small part of the goals pursued by the astronautic agency in charge of the mission. Other objectives include production of more craft, testing and development of new parts, training of personnel, image building, and advertising with potential contributors and customers. The clear impossibility of modelling all of these constraints and objectives together justifies our insistence on the extraction, from each real-world problem, of the combinatorial core, i.e., of a greatly streamlined version which retains the essential flavor of the original, captures as well as possible the difficulties involved in its solution, and yet lends itself to a concise and precise mathematical formulation.

Providing abstract formulations of problems has been a major concern of mathematicians and scientists throughout the ages; even in its few years of existence, the field of computer science has accumulated a large collection of such abstract problems. In order to round out the reader's impression of the field of combinatorial optimization, we present a few more problems, but this time only in their formalized, purified versions. We encourage the reader to relate each of these problems to possible real-world applications.

Problem 24 [Vertex Cover] Given an undirected graph, $G = (V, E)$, a *vertex cover* is a subset of the vertices, $V' \subseteq V$, such that, for each edge, $(u, v) \in E$, at least one of u and v is contained in V'. Find a vertex cover of minimum cardinality. □

Problem 25 [Chromatic Number] A *vertex coloring* of an undirected graph is an assignment of colors to the vertices of the graph such that no edge of the graph connects vertices of the same color. The *chromatic number* of a graph is the minimum number of colors needed for a legal vertex coloring. □

Problem 26 [Chromatic Index] An *edge coloring* of an undirected graph is an assignment of colors to the edges of the graph such that no two edges of the same color share an endpoint. The *chromatic index* of a graph is the minimum number of colors needed for a legal edge coloring. □

Problem 27 [Monochromatic Triangle] Given an undirected graph, $G = (V, E)$, is it possible to partition E into two subsets, E_1 and E_2, such that neither $G_1 = (V, E_1)$ nor $G_2 = (V, E_2)$ contains a "triangle," i.e., a complete subgraph of three vertices?

The problem is called *monochromatic triangle*, because the edges in E_1 can be thought of as being blue and the edges in E_2 can be thought of as being red. The question is then: Can the edges of the graph be colored in such a way that the resulting graph contains neither a blue nor a red triangle? □

Problem 28 [Minimum Testing Collection] Let us take another look at our microbiologist. While the optimal identification tree affords, on the average, the minimum number of tests, it may prove very costly in terms of time. Because many of the tests require a day to perform (the specimen has to incubate), the sequential nature of the identification procedure induces unacceptable delay when a medical diagnosis must be made as rapidly as possible. Even in this situation, it is possible to improve on the procedure of applying all of the tests and then looking for a match between the results and a row of a table similar to Table 1.4. As already pointed out, what is required is that no two rows be identical. Certain columns, i.e., tests, can be eliminated from such a table, and the resulting collection of tests can still separate all bacteria. For example, in Table 1.4, the tests labelled haemolysis, urease, litmus, and arabinose suffice to separate the eight species of *Pasteurella*. Given a table of test results, the minimum testing collection problem is to find the smallest subset of tests that still separates all object categories. □

Problem 29 [Bipartite Matching or Marriage] Recall that an undirected graph is *bipartite* if and only if its vertex set can be partitioned into two subsets, U and V, such that each edge of the graph has one endpoint in U and the other in V. A *matching* in an undirected graph, $G = (V, E)$, is a subset of the edges, $M \subseteq E$, with the property that no two edges in M share an endpoint (one endpoint of each edge in M is "matched" with the other endpoint). A bipartite graph can be considered as giving a list of possible matches between elements of U and elements of V. The maximum bipartite matching problem is to find a matching of maximum cardinality. The alternate name of marriage derives from another interpretation: the vertices of one subset are women, those of the other subset are men, the edges denote marriageable (compatible) pairs, and the problem is to produce as many married couples as possible.

When the two subsets U and V have equal size, we can formulate the decision problem variant known as *perfect matching* by asking whether or not all of the vertices can be matched. □

Problem 30 [Bin Packing] Given a set of elements, S, a size function, $s\colon S \rightarrow \mathcal{R}^+$, and given a bin capacity, $B \in \mathcal{R}^+$, determine the minimum number of bins necessary to pack all the elements of S, where the bin capacity cannot be exceeded and elements are indivisible. This problem can be generalized to a finite inventory of bin capacities by asking that the total amount of wasted space be minimized. □

Problem 31 [Clustering] Given a set of elements, S, with a "dissimilitude" function, $d\colon S \times S \rightarrow \mathcal{R}^+$ (which we can regard as a distance measure, as an incompatibility measure, as a cost measure, etc.—this diversity helps explain why clustering problems are a frequent occurrence in many diverse fields), and given a positive integer, $k < |S|$, partition S into k nonempty subsets—which we shall call *clusters*—such that a suitable objective function is optimized. Two optimization criteria suggest themselves:

1. Minimize the total *intracluster* dissimilitude, that is,

$$\text{minimize} \sum_{i=1}^{k} \sum_{x,y \in C_i} d(x,y),$$

where C_i is the ith cluster. This criterion gives rise to the problem known as *k-min clustering.*

2. Maximize the total *intercluster* dissimilitude, that is,

$$\text{maximize} \sum_{i=1}^{k} \sum_{\substack{j=1 \\ j \neq i}}^{k} \sum_{\substack{x \in C_i \\ y \in C_j}} d(x,y).$$

This criterion gives rise to the problem known as *k-max cut.*

□

1.8 Exercises

Exercise 1.4 Rank all problems presented in this chapter (including the additional problems) in order from most difficult to solve to least difficult to solve. (You will want to form groups of approximately equally difficult problems.) We shall refer back to this list on occasion as new techniques are covered—it will be interesting to see how your initial intuition corresponds to your intuition after reading this book! □

Exercise 1.5 Recast the vertex cover, monochromatic triangle, and minimum testing collection problems as 0-1 programming problems. How many constraints do your formulations produce in terms of the original problem size? (Note that the monochromatic triangle problem provides an example of formulating a decision problem as an integer linear programming problem. In such a case, there is no natural objective function; what matters is whether there are any feasible solutions at all.) □

Exercise 1.6 Show that, if all inequalities are of the "$\leq$" variety and $a_{ij} \geq 0$ for all i, then truncation of the optimal solution of the LP obtained by dropping the integrality constraints from the ILP yields a feasible solution to the ILP. □

Exercise 1.7 Show that, if the conditions of Exercise 1.6 do not hold, then it is possible that an optimal all-integer solution exists, but that neither truncating nor rounding up the variables yields a feasible solution to the ILP. □

Exercise 1.8 Explain why the following 0-1 programming formulation for the travelling salesperson problem is incorrect.

$$\text{minimize} \quad \sum_{\substack{1 \leq i,j \leq n \\ i \neq j}} d_{ij} x_{ij}$$

$$\text{subject to} \quad \begin{cases} \displaystyle\sum_{\substack{j=1 \\ i \neq j}}^{n} x_{ij} = 1, & \text{for } i = 1, 2, \ldots, n \\ \displaystyle\sum_{\substack{i=1 \\ i \neq j}}^{n} x_{ij} = 1, & \text{for } j = 1, 2, \ldots, n \end{cases}$$

$$\text{where} \quad x_{ij} = 0 \text{ or } 1$$

There is an x_{ij} for each ordered pair (i, j) with $i \neq j$. In the optimal solution, x_{ij} equals one whenever the edge from i to j is included in the minimum-cost tour of the graph; x_{ij} equals zero otherwise. The first set of n constraints states that each vertex will be left exactly once, since i is fixed and j varies over all potential next vertices. The second set of n constraints states that every vertex will be entered once. □

Exercise 1.9 If $G = (V, E)$ is a free tree, then $\sum_{i=1}^{|V|} d_i = 2(|V| - 1)$, where d_i is the degree of vertex i. Show that the converse is true, namely: given n positive integers with $\sum_{i=1}^{n} d_i = 2(n-1)$, there is a free tree of n vertices such that the d_i describe the degrees of its vertices. □

Exercise 1.10* Present an informal argument to show that, if all capacities in a network flow problem are integral, then there is an integral flow that is also a maximum flow. (Thus the search space for integral problems is at least finite.) □

Exercise 1.11 Construct an example to show that the collection of tests composing the minimum testing collection can be of lower cardinality than the collection of those tests that occur (somewhere) in the optimal identification tree. This highlights the difference between the sequential searching process embodied in identification trees and the parallel nature of the testing procedure described here—the identification based on the optimal identification tree performs, on average, fewer tests, but requires more capability in the arsenal of tests that may potentially be applied. □

Exercise 1.12 Prove that the following recursive procedure calculates the cost of an identification tree when called externally with a pointer to the root of the tree.

```
const NumObjects = ...;
      NumTests = ...;
type class = (object, test);
     ptr = ^node;
     node = record
              case kind: class of
                object: (who: 1..NumObjects);
                test:   (what: 1..NumTests; left, right: ptr)
            end;
var prob: array [1..NumObjects] of real;
      (* probabilities of request for identification for individual
         object categories *)
procedure CalcCost(p: ptr; var cost, weight: real);
  (* On exit, weight equals sum of the probabilities of request for
     identification of all objects in the subtree rooted at p. *)
  var lcost, rcost, lweight, rweight: real;
  begin
    if p^.kind = object
      then begin
             weight := prob[p^.who];
             cost := 0.0
           end
      else begin
             CalcCost(p^.left,lcost,lweight);
             CalcCost(p^.right,rcost,rweight);
             weight := lweight + rweight;
             cost := lcost + rcost + weight
```

```
        end
end; (* CalcCost *)
```

□

Exercise 1.13 [P] Study and compare Steiner trees and minimal spanning trees; in particular, attempt to estimate how much can be gained by using a Steiner tree rather than a minimal spanning tree. □

Exercise 1.14* Prove that any additional vertices present in the Steiner tree have degree three and that the three edges incident upon such vertices form a regular star (i.e., the angle between two consecutive edges is exactly $120°$). *Hints*: Consider splitting Steiner points of degree higher than 3 into two or more closely spaced vertices and compute the length of the resulting trees. Consider small motions of the Steiner points and compute the lengths of the resulting trees. □

Exercise 1.15 Prove that a partition is an optimal solution to k-min clustering whenever it is an optimal solution to k-max cut. Thus the two clustering problems are very closely related. □

Exercise 1.16* Each of the combinatorial optimization problems that follow is presented in a deliberately vague manner and sometimes also obscured by algorithmically irrelevant details. For each problem, devise one (or more, as a problem may contain more than one combinatorial core) abstract formulation in the spirit of those presented in this chapter.

1. (Database optimization) An airline reservation system must be distributed across many computer systems for purposes of efficiency and reliability. The information to be stored is fairly well known: the list of flights changes neither much nor often. However, updates and queries are frequent: schedule queries, seat reservations, rate requests, and similar operations emanate from all nodes with predictable frequencies. How should the airline pick its computing nodes, connect them, distribute the data between them (including duplications, of course), and organize the flow of queries and updates in order to maximize the efficiency and reliability of the system and minimize its cost?

2. (The MX missile shuffle) Missiles loaded on railroad cars must be moved on a network of tracks in such a manner as to: (i) make it difficult to predict their next location; and (ii) make it impossible to destroy more than one missile with the same shot.

3. (City planning) The city of Albuquerque has spent many years gathering statistics about commercial and residential fires in terms of time and place of occurrence, severity, demands placed on the city's emergency services, and additional damage resulting from the time taken by the emergency crews to get to the scene. Now that it is about to renovate much of its vehicle inventory, it also wants to relocate its emergency facilities so as to make the entire system as effective as possible.

4. (The copy editor) You are writing a book, a software manual, or a very large, well-documented program. You have several versions stored on disk and are working on the latest version, which is to be amended according to a collection of notes. You would very much appreciate a program which would take as inputs the current and the new versions and produce as output as succinct and organized a list of the changes as possible. This program might recognize not only simple word alterations, but also data movement at any structure level (such as moving words within a sentence, sentences within a paragraph or section, paragraphs within a section or chapter), addition and deletion of data (words, sentences, paragraphs), and substitution of words within a pattern (such as use of synonyms).

□

1.9 Bibliography

Linear and integer programming have long been the main object of study of operations research and thus predate computer science; most of the texts discussing these methods have a markedly different flavor from that encountered in texts on algorithms, with the exception of the graduate text of Papadimitriou and Steiglitz [1982], which takes its inspiration from both fields. The simplex method for linear programming was invented by Dantzig in 1947, and his text [1965] remains one of the best sources for linear programming problems and methods; for a more modern exposition, the reader should consult the text of Chvátal [1983]. Readers who want to learn more about integer programming techniques should consult the aforementioned text of Papadimitriou and Steiglitz [1982] and the text of Nemhauser and Wolsey [1988]. Formalized problems appear in all writings dealing with algorithmic design; particularly interesting sources include the monographs on graph algorithms of Christofides [1975] and of Even [1980]; the three volumes of Knuth [1968, 1969, 1973] on general combinatorial problems as well as on seminumerical problems and on search problems; the two-volume work of Berlekamp, Conway, and Guy [1982] on games; the three-volume monograph of Mehlhorn [1984] on search problems, graph problems, geometric problems, and general combinatorial optimization problems; and the text of Pearl [1985] on difficult combinatorial problems. In addition, the text of Garey and Johnson [1979] contains a list, organized by areas of application, of over 300 combinatorial problems (all hard to solve) stated in a very succinct, clear, and precise manner. Whole books have been devoted to some particularly important problems: examples include the travelling salesperson problem (Lawler *et al.* [1985]), the job scheduling problem (Brucker [1981]), the network flow problem (Ford and Fulkerson [1962], the original study of the problem), and the guard problems (O'Rourke [1987]).

The airline crew scheduling problem is discussed in detail by Giannessi and Nicoletti [1979]; they model the problem as closely as possible to reality—in fact, Nicoletti actually worked for an airline (Alitalia). The game of hex was invented by Piet Hein; the argument demonstrating that the first player has a forced win is a special case of a class of arguments first formalized by Hales and Jewett [1963]. Hwang and Du [1990] have proved that the ratio of the lengths of a minimum spanning tree to a Steiner tree cannot exceed $\sqrt{3}/2$ (so that our example of Figure 1.16 is a worst-case example), thereby solving in part Exercise 1.13.