CHAPTER 23

Integer Programming

Many real-world problems could be modeled as linear programs except that some or all of the variables are constrained to be integers. Such problems are called *integer programming problems*. One might think that these problems wouldn't be much harder than linear programming problems. For example, we saw in Chapter 14 that for network flow problems with integer data, the simplex method automatically produces integer solutions. But that was just luck. In general, one can't expect to get integer solutions; in fact, as we shall see in this chapter, integer programming problems turn out to be generally much harder to crack than linear ones.

There are many important real-world problems that can be formulated as integer programming problems. The subject is so important that several monographs are devoted entirely to it. In this chapter, we shall just present a few favorite applications that can be modeled as integer programming problems and then we will discuss one technique for solving problems in this class, called *branch-and-bound*.

1. Scheduling Problems

There are many problems that can be classified as *scheduling problems*. We shall consider just two related problems of this type: the equipment scheduling and crew scheduling problems faced by large airlines. Airlines determine how to route their planes as follows. First, a number of specific flight *legs* are defined based on market demand. A leg is by definition one flight taking off from somewhere at some time and landing somewhere else (hopefully). For example, a leg could be a flight from New York directly to Chicago departing at 7:30 A.M. Another might be a flight from Chicago to San Francisco departing at 1:00 P.M. The important point is that these legs are defined by market demand, and it is therefore not clear a priori how to put these legs together in such a way that the available aircraft can cover all of them. That is, for each airplane, the airline must put together a *route* that it will fly. A route, by definition, consists of a sequence of flight legs for which the destination of one leg is the origin of the next (and, of course, the final destination must be the origin of the first leg, forming a closed loop).

The airline scheduling problems are generally tackled in two stages. First, reasonable routes are identified that meet various regulatory and temporal constraints (you can't leave somewhere before you've arrived there—time also must be reserved for dropping off and taking on passengers). This route-identification problem is by no means trivial, but it isn't our main interest here, so we shall simply assume that a collection of reasonable routes has already been identified. Given the potential routes, the second stage is to select a subset of them with the property that each leg is covered by exactly one route. If the collection of potential routes is sufficiently rich, we would expect there to be several feasible solutions. Therefore, as always, our goal is to pick an optimal one, which in this case we define as one that minimizes the total cost. To formulate this problem as an integer program, let

$$x_j = \begin{cases} 1 & \text{if route } j \text{ is selected,} \\ 0 & \text{otherwise,} \end{cases}$$

$$a_{ij} = \begin{cases} 1 & \text{if leg } i \text{ is part of route } j, \\ 0 & \text{otherwise,} \end{cases}$$

and

$$c_j = \text{cost of using route } j.$$

With these notations, the *equipment scheduling problem* is to

$$\begin{array}{lll} \text{minimize} & \sum_{j=1}^{n} c_j x_j & \\ \text{subject to} & \sum_{j=1}^{n} a_{ij} x_j = 1 & i = 1, 2, \ldots, m, \\ & x_j \in \{0, 1\} & j = 1, 2, \ldots, n. \end{array}$$

This model is often called a *set-partitioning problem*, since the set of legs gets divided, or partitioned, among the various routes.

The flight crews do not necessarily follow the same aircraft around a route. The main reason is that the constraints that apply to flight crews differ from those for the aircraft (for example, flight crews need to sleep occasionally). Hence, the problem has a different set of potential routes. Also, it is sometimes reasonable to allow crews to ride as passengers on some legs with the aim of getting in position for a subsequent flight. With these changes, the *crew scheduling problem* is

$$\begin{array}{lll} \text{minimize} & \sum_{j=1}^{n} c_j x_j & \\ \text{subject to} & \sum_{j=1}^{n} a_{ij} x_j \geq 1 & i = 1, 2, \ldots, m, \\ & x_j \in \{0, 1\} & j = 1, 2, \ldots, n. \end{array}$$

This model is often referred to as a *set-covering problem*, since the crews are assigned so as to cover each leg.

2. The Traveling Salesman Problem

Consider a salesman who needs to visit each of n cities, which we shall enumerate as $0, 1, \ldots, n-1$. His goal is to start from his home city, 0, and make a tour visiting each of the remaining cities once and only once and then returning to his home. We assume that the "distance" between each pair of cities, c_{ij}, is known (distance does not necessarily have to be distance—it could be travel time or, even

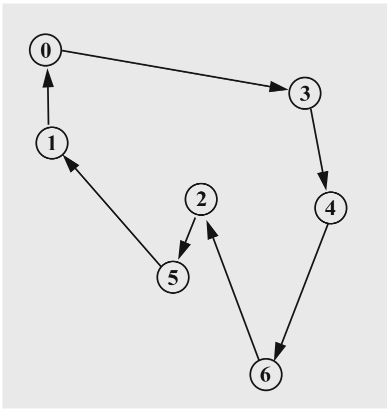

FIGURE 23.1. A feasible tour in a seven-city traveling salesman problem.

better, the cost of travel) and that the salesman wants to make the tour that minimizes the total distance. This problem is called the *traveling salesman problem*. Figure 23.1 shows an example with seven cities. Clearly, a tour is determined by listing the cities in the order in which they will be visited. If we let s_i denote the ith city visited, then the tour can be described simply as

$$s_0 = 0, s_1, s_2, \ldots, s_{n-1}.$$

The total number of possible tours is equal to the number of ways one can permute the $n-1$ cities, i.e., $(n-1)!$. Factorials are huge even for small n (for example, $50! = 3.041 \times 10^{64}$). Hence, enumeration is out of the question. Our aim is to formulate this problem as an integer program that can be solved more quickly than by using enumeration.

It seems reasonable to introduce for each (i, j) a decision variable x_{ij} that will be equal to one if the tour visits city j immediately after visiting city i; otherwise, it will be equal to zero. In terms of these variables, the objective function is easy to write:

$$\text{minimize} \sum_i \sum_j c_{ij} x_{ij}.$$

The tricky part is to formulate constraints to guarantee that the set of nonzero x_{ij}'s corresponds exactly to a bonafide tour. Some of the constraints are fairly obvious. For example, after the salesman visits city i, he must go to one and only one city next. We can write these constraints as

$$\sum_j x_{ij} = 1, \qquad i = 0, 1, \ldots, n-1 \tag{23.1}$$

(we call them the *go-to constraints*). Similarly, when the salesman visits a city, he must have come from one and only one prior city. That is,

$$\sum_i x_{ij} = 1, \qquad j = 0, 1, \ldots, n-1 \tag{23.2}$$

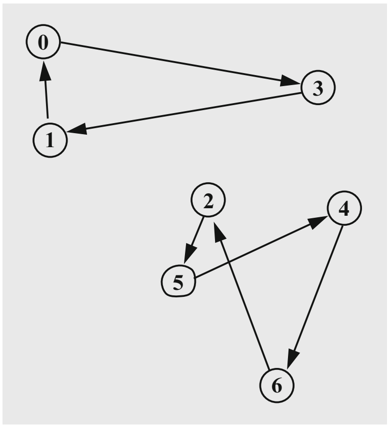

FIGURE 23.2. Two disjoint subtours in a seven-city traveling salesman problem.

(by analogy we call these the *come-from constraints*). If the go-to and the come-from constraints are sufficient to ensure that the decision variables represent a tour, the traveling salesman problem would be quite easy to solve because it would just be an assignment problem, which can be solved efficiently by the simplex method. But unfortunately, these constraints are not sufficient, since they do not rule out the possibility of forming disjoint subtours. An example is shown in Figure 23.2.

We need to introduce more constraints to guarantee connectivity of the graph that the tour represents. To see how to do this, consider a specific tour

$$s_0 = 0, s_1, s_2, \ldots, s_{n-1}.$$

Let t_i for $i = 0, 1, \ldots, n$ be defined as the number of the stop along the tour at which city i is visited; i.e., "when" city i is visited along the tour. From this definition, we see that $t_0 = 0,\ t_{s_1} = 1,\ t_{s_2} = 2$, etc. In general,

$$t_{s_i} = i, \qquad i = 0, 1, \ldots, n-1,$$

so that we can think of the t_j's as being the inverse of the s_i's. For a bonafide tour,

$$t_j = t_i + 1, \qquad \text{if } x_{ij} = 1.$$

Also, each t_i is an integer between 0 and $n-1$, inclusive. Hence, t_j satisfies the following constraints:

$$t_j \geq \begin{cases} t_i + 1 - n & \text{if } x_{ij} = 0, \\ t_i + 1 & \text{if } x_{ij} = 1. \end{cases}$$

(Note that by subtracting n in the $x_{ij} = 0$ case, we have effectively made the condition always hold.) These constraints can be written succinctly as

$$t_j \geq t_i + 1 - n(1 - x_{ij}), \qquad i \geq 0, j \geq 1, i \neq j. \tag{23.3}$$

Now, these constraints were derived based on conditions that a bonafide tour satisfies. It turns out that they also force a solution to be a bonafide tour. That is, they rule out subtours. To see this, suppose to the contrary that there exists a solution to (23.1), (23.2), and (23.3) that consists of at least two subtours. Consider a subtour that does not include city 0. Let r denote the number of legs on this subtour. Clearly, $r \geq 2$. Now, sum (23.3) over all arcs on this subtour. On the left, we get the sum of the t_j's over each city visited by the subtour. On the right, we get the same sum plus r. Cancelling the sums from the two sides, we get that

$$0 \geq r,$$

which is a contradiction. Hence, the traveling salesman problem can be formulated as the following integer programming problem:

$$\begin{array}{lll} \text{minimize} & \displaystyle\sum_{i,j} c_{ij} x_{ij} & \\ \text{subject to} & \displaystyle\sum_{j=1}^{n} x_{ij} = 1, & i = 0, 1, \ldots, n-1, \\ & \displaystyle\sum_{i=1}^{n} x_{ij} = 1, & j = 0, 1, \ldots, n-1, \\ & \quad t_j \geq t_i + 1 - n(1 - x_{ij}), & i \geq 0, j \geq 1, i \neq j, \\ & \quad t_0 = 0, & \\ & \quad x_{ij} \in \{0, 1\}, & \\ & \quad t_i \in \{0, 1, 2, \ldots\}. & \end{array}$$

Note that, for the n-city problem, there are $n^2 + n$ variables in this formulation.

3. Fixed Costs

The terms in an objective function often represent costs associated with engaging in an activity. Until now, we've always assumed that each of these terms is a linear function such as cx. However, it is sometimes more realistic to assume that there is a fixed cost for engaging in the activity plus a linear variable cost. That is, one such term might have the form

$$c(x) = \begin{cases} 0 & \text{if } x = 0 \\ K + cx & \text{if } x > 0. \end{cases}$$

If we assume that there is an upper bound on the size of x, then it turns out that such a function can be equivalently modeled using strictly linear functions at the expense of introducing one integer-valued variable. Indeed, suppose that u is an upper bound on the x variable. Let y denote a $\{0, 1\}$-valued variable that is one when and only when $x > 0$. Then

$$c(x) = Ky + cx.$$

Also, the condition that y is one exactly when $x > 0$ can be guaranteed by introducing the following constraints:

$$\begin{aligned} x &\leq uy \\ x &\geq 0 \\ y &\in \{0, 1\}. \end{aligned}$$

Of course, if the objective function has several terms with associated fixed costs, then this trick must be used on each of these terms.

4. Nonlinear Objective Functions

Sometimes the terms in the objective function are not linear at all. For example, one such term could look like the function shown in Figure 23.3. In Chapter 25, we will discuss efficient algorithms that can be used in the presence of nonlinear objective functions—at least when they have appropriate convexity/concavity properties. In this section, we will show how to formulate an integer programming approximation to a general nonlinear term in the objective function. The first step is to approximate the nonlinear function by a continuous piecewise linear function.

The second step is to introduce integer variables that allow us to represent the piecewise linear function using linear relations. To see how to do this, first we decompose the variable x into a sum,

$$x = x_1 + x_2 + \cdots + x_k,$$

where x_i denotes how much of the interval $[0, x]$ is contained in the ith linear segment of the piecewise linear function (see Figure 23.4). Of course, some of the initial segments will lie entirely within the interval $[0, x]$, one segment will lie partially in and partially out, and then the subsequent segments will lie entirely outside of the interval. Hence, we need to introduce constraints to guarantee that the initial x_i's are equal to the length of their respective segments and that after the straddling segment the subsequent x_i's are all zero. A little thought reveals that the following constraints do the trick:

$$\begin{aligned} L_j w_j \leq x_j &\leq L_j w_{j-1} && j = 1, 2, \ldots, k \\ w_0 &= 1 \\ w_j &\in \{0, 1\} && j = 1, 2, \ldots, k \\ x_j &\geq 0 && j = 1, 2, \ldots, k. \end{aligned}$$

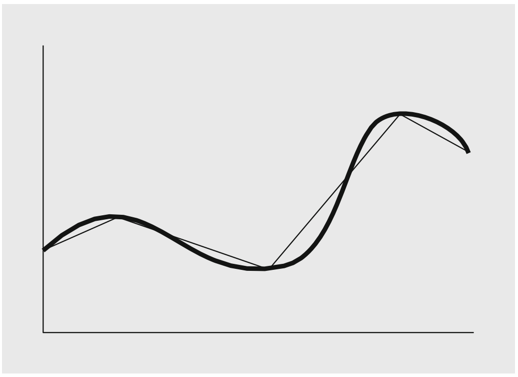

FIGURE 23.3. A nonlinear function and a piecewise linear approximation to it.

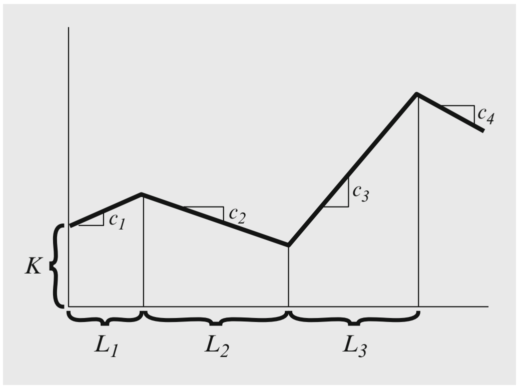

FIGURE 23.4. A piecewise linear function.

Indeed, it follows from these constraints that $w_j \leq w_{j-1}$ for $j = 1, 2, \ldots, k$. This inequality implies that once one of the w_j's is zero, then all the subsequent ones must be zero. If $w_j = w_{j-1} = 1$, the two-sided inequality on x_j reduces to $L_j \leq x_j \leq L_j$. That is, $x_j = L_j$. Similarly, if $w_j = w_{j-1} = 0$, then the two-sided inequality reduces to $x_j = 0$. The only other case is when $w_j = 0$ but $w_{j-1} = 1$. In this case, the two-sided inequality becomes $0 \leq x_j \leq L_j$. Therefore, in all cases, we get what we want. Now with this decomposition we can write the piecewise linear function as

$$K + c_1x_1 + c_2x_2 + \cdots + c_kx_k.$$

5. Branch-and-Bound

In the previous sections, we presented a variety of problems that can be formulated as integer programming problems. As it happens, all of them had the property that the integer variables took just one of two values, namely, zero or one. However, there are other integer programming problems in which the integer variables can be any nonnegative integer. Hence, we define the standard *integer programming problem* as follows:

$$\begin{array}{ll} \text{maximize} & c^Tx \\ \text{subject to} & Ax \leq b \\ & x \geq 0 \\ & x \text{ has integer components.} \end{array}$$

In this section, we shall present an algorithm for solving these problems. The algorithm is called *branch-and-bound*. It involves solving a (potentially) large number of (related) linear programming problems in its search for an optimal integer solution. The algorithm starts out with the following wishful approach: *first ignore the constraint that the components of x be integers, solve the resulting linear programming problem, and hope that the solution vector has all integer components.* Of course, hopes are almost always unfulfilled, and so a backup strategy is needed.

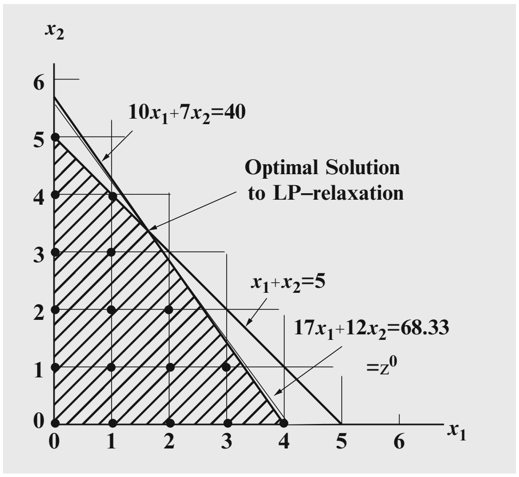

FIGURE 23.5. An integer programming problem. The dots represent the feasible integer points, and the shaded region shows the feasible region for the LP-relaxation.

The simplest strategy would be to round each resulting solution value to its nearest integer value. Unfortunately, this naive strategy can be quite bad. In fact, the integer solution so obtained might not even be feasible, which shouldn't be surprising, since we know that the solution to a linear programming problem is at a vertex of the feasible set and so it is quite expected that naive movement will go outside of the feasible set.

To be concrete, consider the following example:

$$\begin{array}{lrcrcl}
\text{maximize} & 17x_1 & + & 12x_2 & & \\
\text{subject to} & 10x_1 & + & 7x_2 & \leq & 40 \\
& x_1 & + & x_2 & \leq & 5 \\
& & & x_1,\ x_2 & \geq & 0 \\
& & & x_1,\ x_2 & & \text{integers.}
\end{array}$$

The linear programming problem obtained by dropping the integrality constraint is called the *LP-relaxation*. Since it has fewer constraints, its optimal solution provides an upper bound ζ^0 on the the optimal solution ζ^* to the integer programming problem. Figure 23.5 shows the feasible points for the integer programming problem as well as the feasible polytope for its LP-relaxation. The solution to the LP-relaxation is at $(x_1, x_2) = (5/3, 10/3)$, and the optimal objective value is $205/3 = 68.33$. Rounding each component of this solution to the nearest integer, we get $(2, 3)$, which is not even feasible. The feasible integer solution that is closest to the LP-optimal solution is $(1, 3)$, but we can see from Figure 23.5 that this solution is not the optimal solution to the integer programming problem. In fact, it is easy to see from the figure that the optimal integer solution is either $(1, 4)$ or $(4, 0)$. To make the

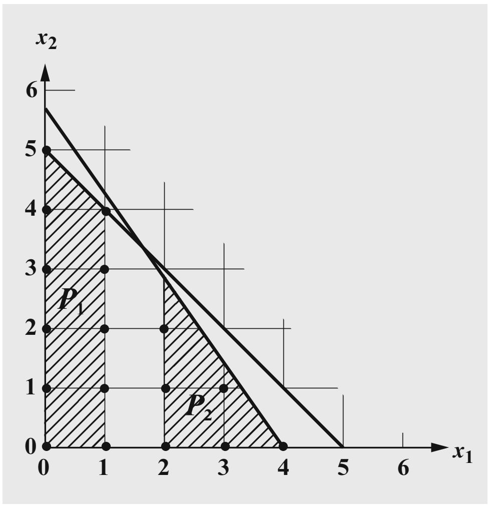

FIGURE 23.6. The feasible subregions formed by the first branch.

problem interesting, we've chosen the objective function to make the more distant point $(4, 0)$ be the optimal solution.

Of course, we can solve only very small problems by the graphical method: to solve larger problems, an algorithm is required, which we now describe. Consider variable x_1 in the optimal solution to the LP-relaxation. Its value is $5/3$. In the optimal solution to the integer programming problem, it will be an integer. Hence, it will satisfy either $x_1 \leq 1$ or $x_1 \geq 2$. We consider these two cases separately. Let P_1 denote the linear programming problem obtained by adding the constraint $x_1 \leq 1$ to the LP-relaxation, and let P_2 denote the problem obtained by including the other possibility, $x_1 \geq 2$. The feasible regions for P_1 and P_2 are shown in Figure 23.6. Let us study P_1 first. It is clear from Figure 23.6 that the optimal solution is at $(x_1, x_2) = (1, 4)$ with an objective value of 65. Our algorithm has found its first feasible solution to the integer programming problem. We record this solution as the *best-so-far*. Of course, better ones may (in this case, will) come along later.

Now let's consider P_2. Looking at Figure 23.6 and doing a small amount of calculation, we see that the optimal solution is at $(x_1, x_2) = (2, 20/7)$. In this case, the objective function value is $478/7 = 68.29$. Now if this value had turned out to be less than the best-so-far value, then we'd be done, since any integer solution that lies within the feasible region for P_2 would have a smaller value yet. But this is not the case, and so we must continue our systematic search. Since $x_2 = 20/7 = 2.86$, we divide P_2 into two subproblems, one in which the constraint $x_2 \leq 2$ is added and one with $x_2 \geq 3$ added.

Before considering these two new cases, note that we are starting to develop a tree of linear programming subproblems. This tree is called the *enumeration tree*. The tree as far as we have investigated is shown in Figure 23.7. The double box

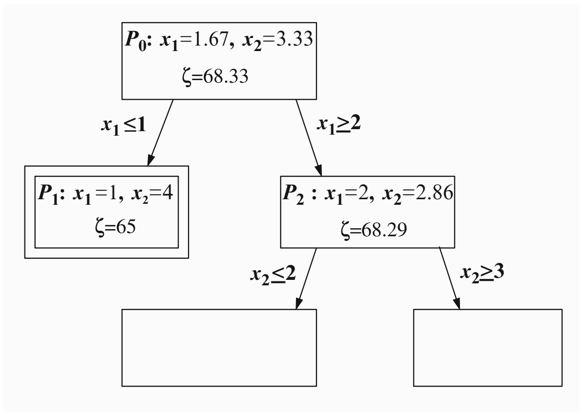

FIGURE 23.7. The beginnings of the enumeration tree.

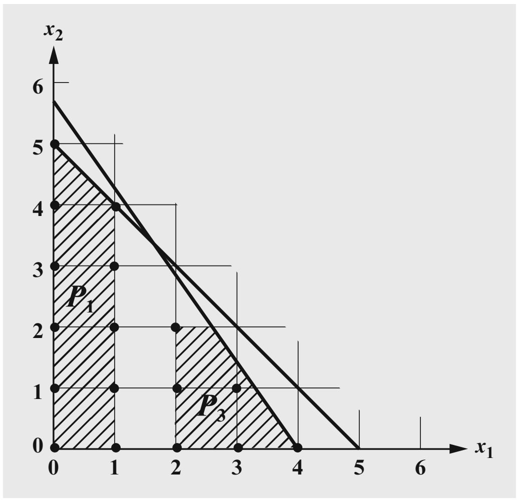

FIGURE 23.8. The refinement of P_2 to P_3.

around P_1 indicates that that part of the tree is done: i.e., there are no branches emanating from P_1—it is a leaf node. The two empty boxes below P_2 indicate two subproblems that have yet to be studied. Let's proceed by looking at the left branch, which corresponds to adding the constraint $x_2 \leq 2$ to what we had before. We denote this subproblem by P_3. Its feasible region is shown in Figure 23.8, from which we see that the optimal solution is at $(2.6, 2)$. The associated optimal objective value is 68.2. Again, the solution is fractional. Hence, the process of subdividing must continue. This time we subdivide based on the values of x_1. Indeed, we consider two cases: either $x_1 \leq 2$ or $x_1 \geq 3$.

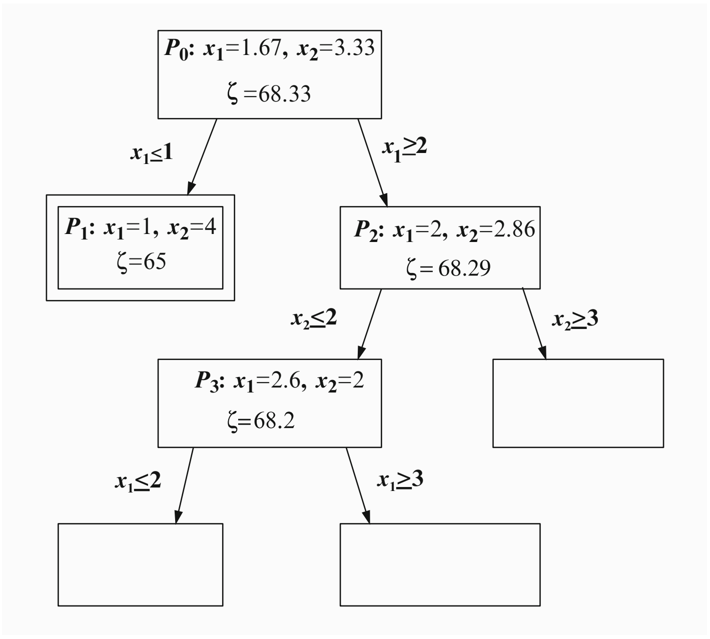

FIGURE 23.9. The enumeration tree after solving P_3.

Figure 23.9 shows the enumeration tree as it now stands. At this juncture, there are three directions in which we could proceed. We could either study the other branch under P_2 or work on one of the two branches sitting under P_3. If we were to systematically solve all the problems on a given level of the tree before going deeper, we would be performing what is referred to as a *breadth-first search*. On the other hand, going deep before going wide is called a *depth-first search*. For reasons that we shall explain later, it turns out to be better to do a depth-first search. And, to be specific, let us always choose the left branch before the right branch (in practice, there are much better rules that one can employ here). So our next linear programming problem is the one that we get by adding the constraint that $x_1 \le 2$ to the constraints that defined P_3. Let us call this new problem P_4. Its feasible region is shown in Figure 23.10. It is easy to see that the optimal solution to this problem is $(2, 2)$, with an objective value of 58. This solution is an integer solution, so it is feasible for the integer programming problem. But it is not better than our best-so-far. Nonetheless, we do not need to consider any further subproblems below this one in the enumeration tree.

Since problem P_4 is a leaf in the enumeration tree, we need to work back up the tree looking for the first node that has an unsolved problem sitting under it. For the case at hand, the unsolved problem is on the right branch underneath P_3. Let us call this problem P_5. It too is depicted in Figure 23.10. The optimal solution is $(3, 1.43)$, with an optimal objective function value of 68.14. Since this objective

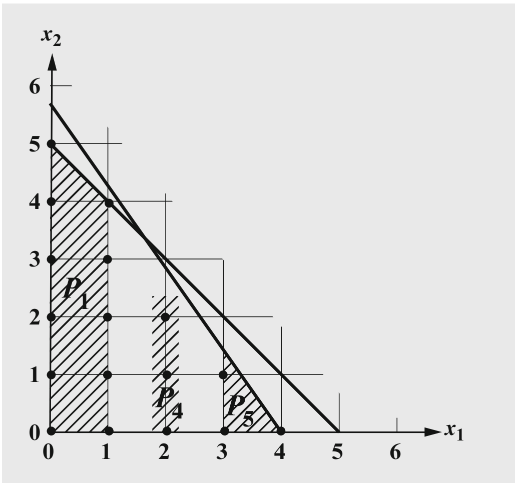

FIGURE 23.10. The refinement of P_3 to P_4.

function value is larger than the value of the best-so-far integer solution, we must further investigate by dividing into two possibilities, either $x_2 \leq 1$ or $x_2 \geq 2$. At this point, the enumeration tree looks like that shown in Figure 23.11.

Let P_6 denote the linear programming problem that we get on the left branch under P_5. Its feasible region is shown in Figure 23.12. The optimal solution is $(3.3, 1)$, and the associated objective value is 68.1. Again, the solution is fractional and has a higher objective value than the best-so-far integer solution. Hence, it must be subdivided based on $x_1 \leq 3$ as opposed to $x_1 \geq 4$. Denoting these two problems by P_7 and P_8, their feasible regions are as depicted in Figure 23.13. The solution to P_7 is $(3, 1)$, and the objective value is 63. This is an integer solution, but it is not better than the best-so-far. Nonetheless, the node becomes a leaf, since the solution is integral. Hence, we move on to P_8. The solution to this problem is also integral, $(4, 0)$. Also, the objective value associated with this solution is 68, which is a new record for feasible integer solutions. Hence, this solution becomes our best-so-far. The enumeration tree at this point is shown in Figure 23.14.

Now we need to go back and solve the problems under P_5 and P_2 (and any subproblems thereof). It turns out that both these subproblems are infeasible, and so no more subdivisions are needed. The enumeration tree is now completely fathomed and is shown in Figure 23.15. We can now assert that the optimal solution to the original integer programming problem was found in problem P_8. The solution is $(x_1, x_2) = (4, 0)$, and the associated objective function value is 68.

There are three reasons why depth-first search is generally the preferred order in which to fathom the enumeration tree. The first is based on the observation that most integer solutions lie deep in the tree. There are two advantages to finding integer feasible solutions early. The first is simply the fact that it is better to have a feasible solution than nothing in case one wishes to abort the solution process

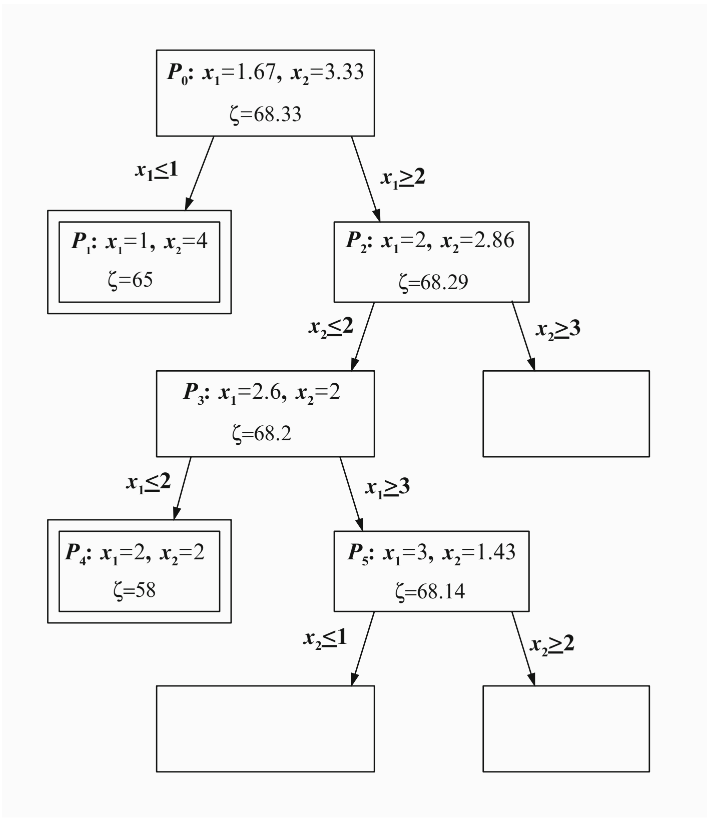

FIGURE 23.11. The enumeration tree after solving P_5. The double box around P_4 indicates that it is a leaf in the tree.

early. But more importantly, identifying an feasible integer solution can result in subsequent nodes of the enumeration tree being made into leaves simply because the optimal objective function associated with that node is lower than the best-so-far integer solution. Making such nodes into leaves is called *pruning* the tree and can account for tremendous gains in efficiency.

A second reason to favor depth-first search is the simple fact that it is very easy to code the algorithm as a recursively defined function. This may seem trite, but one shouldn't underestimate the value of code simplicity when implementing algorithms that are otherwise quite sophisticated, such as the one we are currently describing.

The third reason to favor depth-first search is perhaps the most important. It is based on the observation that as one moves deeper in the enumeration tree, each subsequent linear programming problem is obtained from the preceding one by simply adding (or refining) an upper/lower bound on one specific variable. To see why this is an advantage, consider for example problem P_2, which is a refinement of P_0. The optimal dictionary for problem P_0 is recorded as

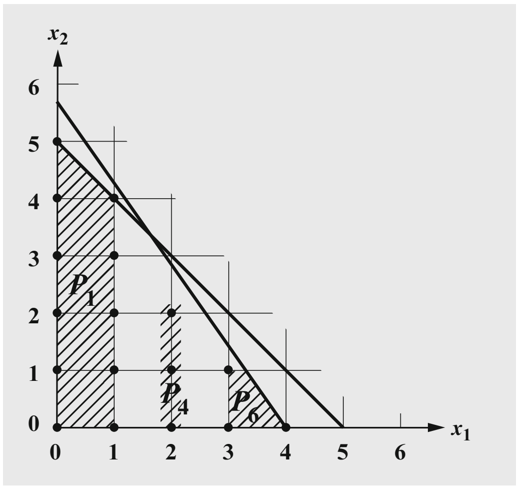

FIGURE 23.12. The refinement of P_5 to P_6.

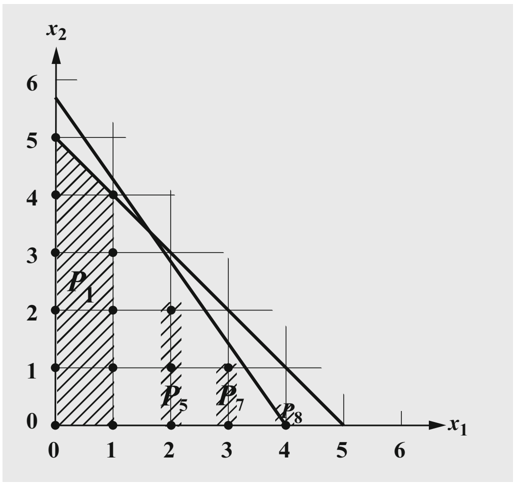

FIGURE 23.13. The refinement of P_6 to P_7 and P_8.

$$\begin{array}{rcrcrcr}
\zeta & = & \frac{205}{3} & - & \frac{5}{3}w_1 & - & \frac{1}{3}w_2 \\
\hline
x_1 & = & \frac{5}{3} & - & \frac{1}{3}w_1 & + & \frac{7}{3}w_2 \\
x_2 & = & \frac{10}{3} & + & \frac{1}{3}w_1 & - & \frac{10}{3}w_2.
\end{array}$$

Problem P_2 is obtained from P_0 by adding the constraint that $x_1 \geq 2$. Introducing a variable, g_1, to stand for the difference between x_1 and this lower bound and using the dictionary above to write x_1 in terms of the nonbasic variables, we get

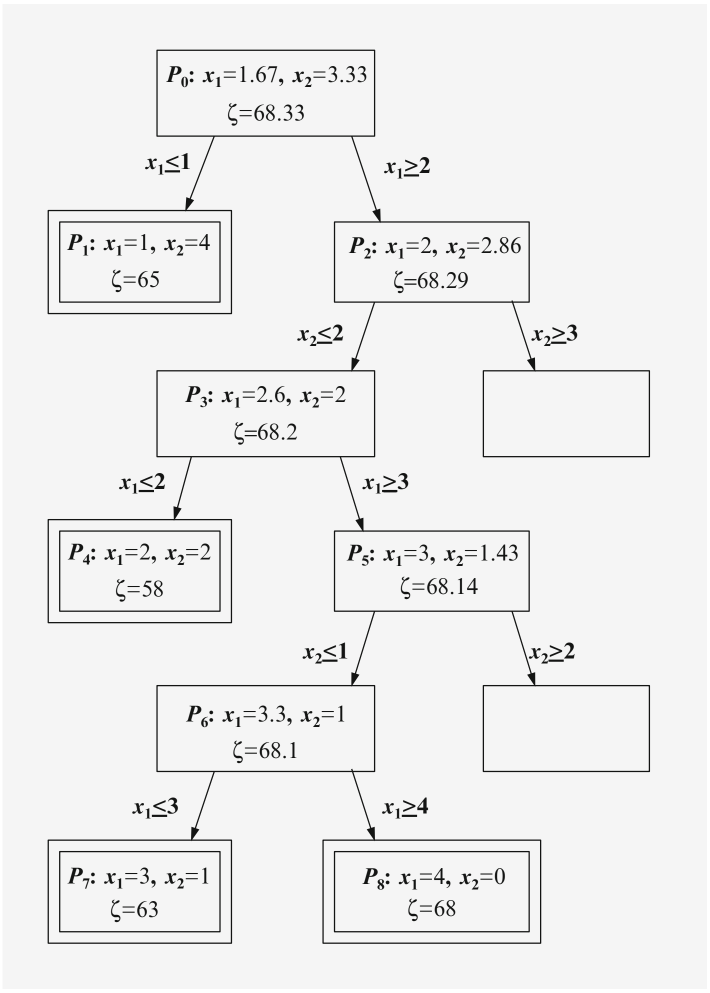

FIGURE 23.14. The enumeration tree after solving P_6, P_7, and P_8.

$$g_1 = x_1 - 2 = -\frac{1}{3} - \frac{1}{3}w_1 + \frac{7}{3}w_2.$$

Therefore, we can use the following dictionary as a starting point for the solution of P_2:

$$\begin{array}{rrrr}
\zeta = & \frac{205}{3} & - \frac{5}{3}w_1 & - \frac{1}{3}w_2 \\
\hline
x_1 = & \frac{5}{3} & - \frac{1}{3}w_1 & + \frac{7}{3}w_2 \\
x_2 = & \frac{10}{3} & + \frac{1}{3}w_1 & - \frac{10}{3}w_2 \\
g_1 = & -\frac{1}{3} & - \frac{1}{3}w_1 & + \frac{7}{3}w_2.
\end{array}$$

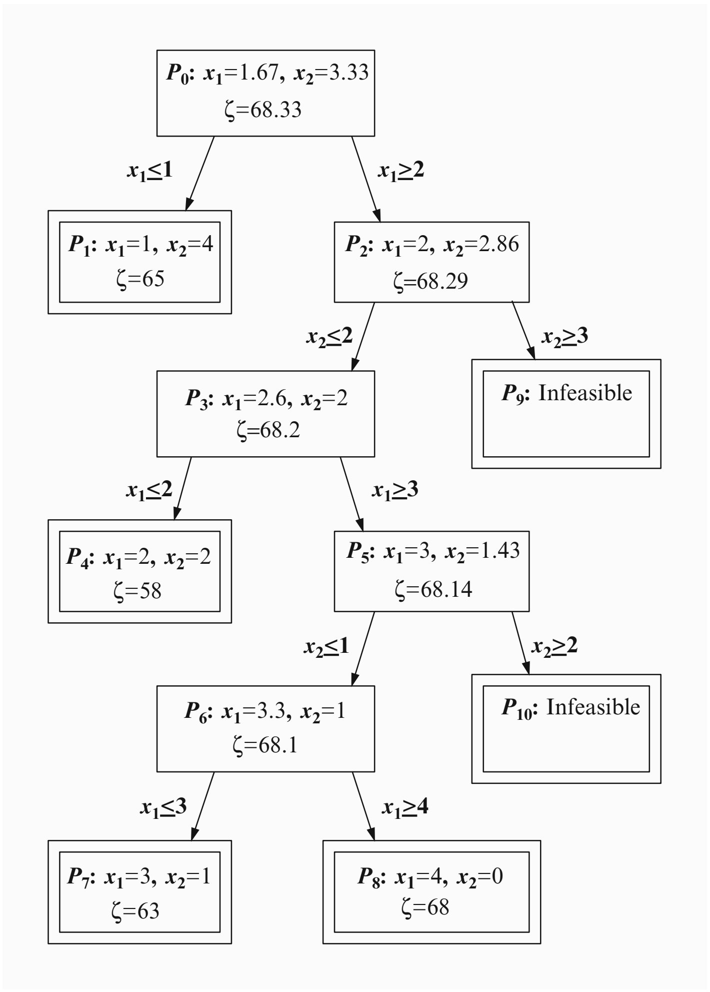

FIGURE 23.15. The complete enumeration tree.

This dictionary is dual feasible but primal infeasible. Therefore, the dual simplex method is likely to find a new optimal solution in very few iterations. According to the dual simplex method, variable g_1 is the leaving variable and w_2 is the corresponding entering variable. Making the pivot, we get the following dictionary:

$$\begin{array}{rcrcrcr}
\zeta & = & \frac{478}{7} & - & \frac{12}{7}w_1 & - & \frac{1}{7}g_1 \\
\hline
x_1 & = & 2 & & & + & g_1 \\
x_2 & = & \frac{20}{7} & - & \frac{1}{7}w_1 & - & \frac{10}{7}g_1 \\
w_2 & = & \frac{1}{7} & + & \frac{1}{7}w_1 & + & \frac{3}{7}g_1 .
\end{array}$$

This dictionary is optimal for P_2. In general, the dual simplex method will take more than one iteration to reoptimize, but nonetheless, one does expect it to get to a new optimal solution quickly.

We end this chapter by remarking that many real problems have the property that some variables must be integers but others can be real valued. Such problems are called *mixed integer programming problems*. It should be easy to see how to modify the branch-and-bound technique to handle such problems as well.

Exercises

23.1 *Knapsack Problem*. Consider a picnicker who will be carrying a knapsack that holds a maximum amount b of "stuff." Suppose that our picnicker must decide what to take and what to leave behind. The jth thing that might be taken occupies a_j units of space in the knapsack and will bring c_j amount of "enjoyment." The knapsack problem then is to maximize enjoyment subject to the constraint that the stuff brought must fit into the knapsack:

$$\begin{array}{ll} \text{maximize} & \displaystyle\sum_{j=1}^{n} c_j x_j \\ \text{subject to} & \displaystyle\sum_{j=1}^{n} a_j x_j \leq b \\ & \qquad x_j \in \{0, 1\} \qquad j = 1, 2, \ldots, n. \end{array}$$

This apparently simple problem has proved difficult for general-purpose branch-and-bound algorithms. To see why, analyze the special case in which each thing contributes the same amount of enjoyment, i.e., $c_j = c$ for all j, and takes up exactly two units of space, i.e., $a_j = 2$ for all j. Suppose also that the knapsack holds n units of stuff.

(a) What is the optimal solution when n is even? when n is odd?

(b) How many subproblems must the branch-and-bound algorithm consider when n is odd?

23.2 *Vehicle Routing*. Consider the dispatching of delivery vehicles (for example, mail trucks, fuel-oil trucks, newspaper delivery trucks, etc.). Typically, there is a fleet of vehicles that must be routed to deliver goods from a depot to a given set of n drop-points. Given a set of feasible delivery routes and the cost associated with each one, explain how to formulate the problem of minimizing the total delivery cost as a set-partitioning problem.

23.3 Explain how to modify the integer programming reformulation of continuous piecewise linear functions so that it covers piecewise linear functions having discontinuities at the junctions of the linear segments. Can fixed costs be handled with this approach?

Notes

Standard references for integer programming include the classic text by Garfinkel and Nemhauser (1972) and the more recent text by Nemhauser and Wolsey (1988). Bill Cook's recent book Cook (2012) on the traveling salesman problem gives an entertaining historical and mathematical perspective on this particular application of integer programming.