time algorithm for the matching problem and showed that the convex hull of the integer feasible solutions to the matching problem is given by P_{matching}. For a textbook exposition of matching algorithms see Papadimitriou and Steiglitz (1982), and Nemhauser and Wolsey (1988). Much more information on the traveling salesman problem can be found in Lawler et al. (1985).

Chapter 11

Integer programming methods

Contents

Unlike linear programming problems, integer programming problems are very difficult to solve. In fact, no efficient general algorithm is known for their solution. In this chapter, we review algorithms for integer programming problems, we develop a duality theory that facilitates algorithmic development, and discuss evidence suggesting that that these problems are inherently hard.

There are three main categories of algorithms:

(a) **Exact algorithms** that are guaranteed to find an optimal solution, but may take an exponential number of iterations. They include cutting plane (Section 11.1), branch and bound and branch and cut (Section 11.2), and dynamic programming methods (Section 11.3).

(b) **Approximation algorithms** that provide in polynomial time a suboptimal solution together with a bound on the degree of suboptimality (Section 11.5).

(c) **Heuristic algorithms** that provide a suboptimal solution, but without a guarantee on its quality. Although the running time is not guaranteed to be polynomial, empirical evidence suggests that some of these algorithms find a good solution fast. As examples we introduce local search methods (Section 11.6), and simulated annealing (Section 11.7).

Duality theory is central to linear programming. Integer programming also has a duality theory, presented in Section 11.4, which provides bounds on the optimal cost. These bounds are very useful in exact algorithms, as they can be used to avoid enumerating too many feasible solutions, and in approximation algorithms, as they provide performance guarantees.

Given our inability to solve integer programming problems efficiently, it is natural to ask whether such problems are inherently hard. Complexity theory, reviewed in Section 11.8, offers some insights on this question. It provides us with a class of problems with the following property: if a polynomial time algorithm exists for any problem in this class, then all integer programming problems can be solved by an efficient algorithm, but this is considered unlikely.

11.1 Cutting plane methods

We consider the integer programming problem

$$\begin{array}{ll} \text{minimize} & \mathbf{c}'\mathbf{x} \\ \text{subject to} & \mathbf{A}\mathbf{x} = \mathbf{b} \\ & \mathbf{x} \geq \mathbf{0} \\ & \mathbf{x} \text{ integer,} \end{array} \tag{11.1}$$

and its linear programming relaxation

$$\begin{array}{ll} \text{minimize} & \mathbf{c}'\mathbf{x} \\ \text{subject to} & \mathbf{A}\mathbf{x} = \mathbf{b} \\ & \mathbf{x} \geq \mathbf{0}. \end{array} \tag{11.2}$$

The main idea in cutting plane methods is to solve the integer programming problem (11.1) by solving a sequence of linear programming problems, as follows. We first solve the linear programming relaxation (11.2) and find an optimal solution $\mathbf{x}^*$. If $\mathbf{x}^*$ is integer, then it is an optimal solution to the integer programming problem (11.1). If not, we find an inequality that all integer solutions to (11.1) satisfy, but $\mathbf{x}^*$ does not. We add this inequality to the linear programming problem to obtain a tighter relaxation, and we iterate this step.

A generic cutting plane algorithm

1. Solve the linear programming relaxation (11.2). Let $\mathbf{x}^*$ be an optimal solution.
2. If $\mathbf{x}^*$ is integer stop; $\mathbf{x}^*$ is an optimal solution to (11.1).
3. If not, add a linear inequality constraint to (11.2) that all integer solutions to (11.1) satisfy, but $\mathbf{x}^*$ does not; go to Step 1.

Note that this method is just a variation of the cutting plane algorithm introduced in Section 6.3. As in that section, the main idea is to generate a violated constraint, whenever the relaxed problem gives rise to an infeasible solution. The performance of a cutting plane method depends critically on the choice of the inequality used to "cut" $\mathbf{x}^*$. We review next ways to introduce cuts that give rise to particular cutting plane algorithms.

Example 11.1 (An example of a cut) Let $\mathbf{x}^*$ be an optimal basic feasible solution to (11.2) with at least one fractional basic variable. Let N be the set of indices of the nonbasic variables. Consider any solution to the integer programming problem such that $x_i = 0$ for all $i \in N$. Then, it is a solution to the linear programming problem as well, and it must be the same as the basic feasible solution $\mathbf{x}^*$. Since $\mathbf{x}^*$ is not feasible for the integer programming problem, then all feasible integer solutions satisfy

$$\sum_{j \in N} x_j \geq 1.$$

This is the inequality that we add to the relaxation (11.2). Note that all integer solutions to (11.1) satisfy it, while the optimal solution $\mathbf{x}^*$ to the relaxation violates it.

The Gomory cutting plane algorithm

The first finitely terminating algorithm for integer programming was a cutting plane algorithm proposed by Gomory in 1958, which uses some detailed information from the optimal simplex tableau.

We solve the standard form linear programming problem (11.2) with the simplex method. Let $\mathbf{x}^*$ be an optimal basic feasible solution and let $\mathbf{B}$ be an associated optimal basis. We partition $\mathbf{x}$ into a subvector $\mathbf{x}_B$ of basic variables and a subvector $\mathbf{x}_N$ of nonbasic variables. Recall from Chapter 3 that a tableau provides us with the coefficients of the equation $\mathbf{B}^{-1}\mathbf{A}\mathbf{x} = \mathbf{B}^{-1}\mathbf{b}$. Let N be the set of indices of nonbasic variables. Let $\mathbf{A}_N$ be the submatrix of $\mathbf{A}$ with columns $\mathbf{A}_i$, $i \in N$. From the optimal tableau we obtain the coefficients of the constraints

$$\mathbf{x}_B + \mathbf{B}^{-1}\mathbf{A}_N\mathbf{x}_N = \mathbf{B}^{-1}\mathbf{b}.$$

Let $\bar{a}_{ij} = \left(\mathbf{B}^{-1}\mathbf{A}_j\right)_i$ and $\bar{a}_{i0} = \left(\mathbf{B}^{-1}\mathbf{b}\right)_i$. We consider one equality from the optimal tableau, in which $\bar{a}_{i0}$ is fractional:

$$x_i + \sum_{j \in N} \bar{a}_{ij} x_j = \bar{a}_{i0}.$$

Since $x_j \geq 0$ for all j, we have

$$x_i + \sum_{j \in N} \lfloor \bar{a}_{ij} \rfloor x_j \leq x_i + \sum_{j \in N} \bar{a}_{ij} x_j = \bar{a}_{i0}.$$

Since x_i should be integer, we obtain

$$x_i + \sum_{j \in N} \lfloor \bar{a}_{ij} \rfloor x_j \leq \lfloor \bar{a}_{i0} \rfloor.$$

This inequality is valid for all integer solutions, but it is not satisfied by $\mathbf{x}^*$. The reason is that $x_i^* = \bar{a}_{i0}$, $x_j^* = 0$ for all nonbasic $j \in N$, and $\lfloor \bar{a}_{i0} \rfloor < \bar{a}_{i0}$ (since $\bar{a}_{i0}$ was assumed fractional).

It has been shown that by systematically adding these cuts, and using the dual simplex method with appropriate anticycling rules, we obtain a finitely terminating algorithm for solving general integer programming problems. See Section 5.1 on how to apply the dual simplex method, when new inequality constraints are added. In practice, however, this method has not been particularly successful.

Example 11.2 (Illustration of the Gomory cutting plane algorithm) We consider the integer programming problem

$$\begin{array}{lrcl} \text{minimize} & x_1 - 2x_2 & & \\ \text{subject to} & -4x_1 + 5x_2 & \leq & 9 \\ & x_1 + x_2 & \leq & 4 \\ & x_1, x_2 \geq 0 & & \\ & x_1, x_2 \text{ integer.} & & \end{array}$$

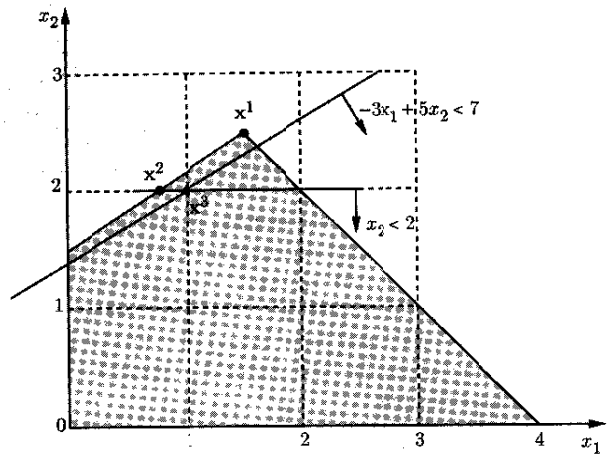

Figure 11.1: The Gomory cutting plane algorithm for Example 11.2. The shaded region is the feasible set of the linear programming relaxation.

We transform the problem in standard form

$$\begin{array}{lrcl} \text{minimize} & x_1 - 2x_2 & & \\ \text{subject to} & -4x_1 + 6x_2 + x_3 & = & 9 \\ & x_1 + x_2 + x_4 & = & 4 \\ & x_1, \ldots, x_4 \geq 0 & & \\ & x_1, \ldots, x_4 \text{ integer.} & & \end{array}$$

We solve the linear programming relaxation, and the optimal solution (in terms of the original variables) is $\mathbf{x}^1 = (15/10, 25/10)$. One of the equations in the optimal tableau is

$$x_2 + \frac{1}{10}x_3 + \frac{1}{10}x_4 = \frac{25}{10}.$$

We apply the Gomory cutting plane algorithm, and we find the cut

$$x_2 \leq 2.$$

We augment the linear programming relaxation by adding the constraints $x_2 + x_5 = 2$, $x_5 \geq 0$, and we find that the new optimal solution is $\mathbf{x}^2 = (3/4, 2)$. One of the equations in the optimal tableau is

$$x_1 - \frac{1}{4}x_3 + \frac{6}{4}x_5 = \frac{3}{4}.$$

We add a new Gomory cut

$$x_1 - x_3 + x_5 \leq 0,$$

which, in terms of the original variables x_1, x_2, is

$$-3x_1 + 5x_2 \leq 7.$$

We add this constraint, together with the previously added constraint $x_2 \leq 2$, and find that the new optimal solution is $\mathbf{x}^3 = (1, 2)$. Since the solution $\mathbf{x}^3$ is integer, it is an optimal solution to the original problem; see Figure 11.1.

A difficulty with general purpose cutting plane algorithms is that the added inequalities cut only a very small piece of the feasible set of the linear programming relaxation. As a result, the practical performance of such algorithms has not been impressive. For this reason, cutting plane algorithms with deeper cuts have been designed. These cuts utilize the structure of the particular integer programming problem. We illustrate such methods with an example.

Example 11.3 (The weighted independent set problem) Given an undirected graph $G = (\mathcal{N}, \mathcal{E})$ and weights w_i for each $i \in \mathcal{N}$, the weighted independent set problem asks for a collection of nodes S of maximum weight, so that no two nodes in S are adjacent. We let $x_i = 1$ if node i is selected in the independent set, and $x_i = 0$, otherwise. The problem can then be formulated as follows:

$$\begin{aligned} \text{maximize} \quad & \sum_{i=1}^{n} w_i x_i \\ \text{subject to} \quad & x_i + x_j \leq 1, \qquad (i,j) \in \mathcal{E}, \\ & x_i \in \{0,1\}, \qquad i \in \mathcal{N}. \end{aligned}$$

A collection of nodes U such that for any $i, j \in U$ we have $(i,j) \in \mathcal{E}$, is called a *dique*. Clearly the following inequality is valid for all feasible solutions to the incependent set problem:

$$\sum_{i \in U} x_i \leq 1, \quad \text{for any clique } U.$$

A set of nodes $U = \{i_1, \ldots, i_k\}$ is called a *cycle* if the only edges joining nodes in U are $\{i_1\, i_2\}, \{i_2, i_3\}, \ldots, \{i_k, i_1\}$. For any cycle U of odd cardinality, there can be no more that $\big(|U| - 1\big)/2$ nodes in an independent set; otherwise, two of these nodes will be adjacent. Therefore, the inequality

$$\sum_{i \in U} x_i \leq \frac{|U| - 1}{2}, \quad \text{for any cycle } U \text{ such that } |U| \text{ is odd},$$

must hold.

The inequalities we derived above utilize the particular combinatorial structure of the maximum independent set problem. If we use these inequalities in the generic cutting plane method we described, the performance of the algorithm is greatly enhanced. However, given an $\mathbf{x}^*$, we must search for a violated inequality of either type, which can be difficult.

11.2 Branch and bound

Branch and bound uses a "divide and conquer" approach to explore the set of feasible integer solutions. However, instead of exploring the entire feasible set, it uses bounds on the optimal cost to avoid exploring certain parts of the set of feasible integer solutions.

Let F be the set of feasible solutions to the problem

$$\begin{aligned} \text{minimize} \quad & \mathbf{c}'\mathbf{x} \\ \text{subject to} \quad & \mathbf{x} \in F. \end{aligned}$$

[For example, F could be the set of integer feasible solutions to the problem (11.1).] We partition the set F into a finite collection of subsets $F_1, \ldots, F_k$, and solve separately each one of the subproblems

$$\begin{aligned} \text{minimize} \quad & \mathbf{c}'\mathbf{x} \\ \text{subject to} \quad & \mathbf{x} \in F_i, \quad i = 1, \ldots, k. \end{aligned}$$

We then compare the optimal solutions to the subproblems, and choose the best one. Each subproblem may be almost as difficult as the original problem and this suggests trying to solve each subproblem by means of the same method; that is, by splitting it into further subproblems, etc. This is the branching part of the method and leads to a tree of subproblems; see Figure 11.2.

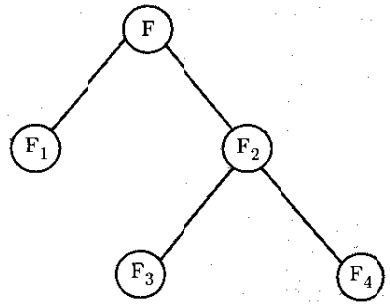

Figure 11.2: A tree of subproblems: the feasible set F is partitioned into F_1 and F_2; also, F_2 is partitioned into F_3 and F_4.

We also assume that there is a fairly efficient algorithm, which for every F_i of interest, computes a *lower bound* $b(F_i)$ to the optimal cost of the corresponding subproblem; that is,

$$b(F_i) \leq \min_{\mathbf{x} \in F_i} \mathbf{c}'\mathbf{x}.$$

The basic idea is that while the optimal cost in a subproblem may be difficult to compute exactly, a lower bound might be a lot easier to obtain.

A popular method to obtain such a bound is to use the optimal cost of the linear programming relaxation.

In the course of the algorithm, we will also occasionally solve certain subproblems to optimality, or simply evaluate the cost of certain feasible solutions. This allows us to maintain an upper bound U on the optimal cost, which could be the cost of the best feasible solution encountered thus far.

The essence of the method lies in the following observation. If the lower bound $b(F_i)$ corresponding to a particular subproblem satisfies $b(F_i) \geq U$, then this subproblem need not be considered further, since the optimal solution to the subproblem is no better than the best feasible solution encountered thus far.

The following is a high-level description of the resulting algorithm. At any point, the algorithm keeps in memory a set of outstanding (active) subproblems and the cost U of the best feasible solution so far. Initially, U is set either to ∞ or to the cost of some feasible solution, if one happens to be available. A typical stage of the algorithm proceeds as follows.

A generic branch and bound algorithm

1. Select an active subproblem F_i.
2. If the subproblem is infeasible, delete it; otherwise, compute $b(F_i)$ for the corresponding subproblem.
3. If $b(F_i) \geq U$, delete the subproblem.
4. If $b(F_i) < U$, either obtain an optimal solution to the subproblem, or break the corresponding subproblem into further subproblems, which are added to the list of active subproblems.

There are several "free parameters" in this algorithm. The best choices are usually dictated by experience.

(a) There are different ways of choosing an active subproblem. Two extreme choices are "breadth-first search" and "depth-first search."

(b) There may be several ways of obtaining a lower bound $b(F_i)$ on the optimal cost of a subproblem. One possibility that we have already mentioned is to consider the linear programming relaxation. We consider other possibilities in Section 11.4.

(c) There are usually several ways of breaking a problem into subproblems.

As an illustration, we use as a lower bound $b(F_i)$ the optimal cost of the linear programming relaxation whereby the integrality constraints are ignored. If an integer optimal solution to the relaxation is obtained, then it is automatically an optimal solution to the corresponding integer programming problem as well, and there is no need for expanding into

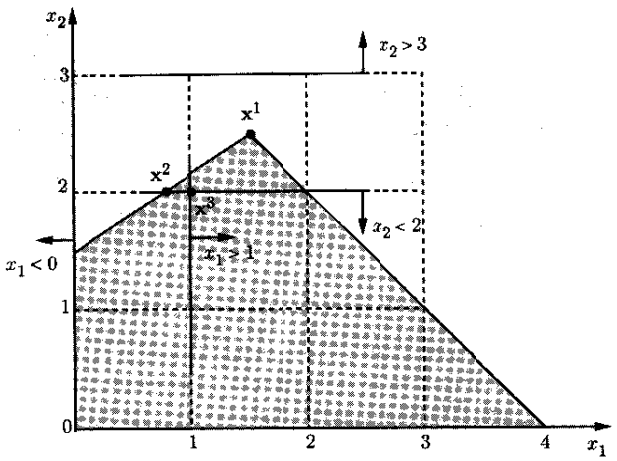

Figure 11.3: Branch and bound in Example 11.4.

further subproblems. We only need to update U (if the cost of this optimal solution is better than the previous value of U), and we can delete the current subproblem. If the optimal solution $\mathbf{x}^*$ to the linear programming relaxation is not integer, we choose a component x_i for which x_i^* is not integer and create two subproblems, by adding either of the constraints

$$x_i \leq \lfloor x_i^* \rfloor, \qquad \text{or} \qquad x_i \geq \lceil x_i^* \rceil.$$

(Note that both constraints are violated by $\mathbf{x}^*$. If $\mathbf{x}^*$ is the unique optimal solution to the linear programming relaxation, then the optimal cost in the relaxation of either of the new subproblems will be strictly larger.) Given that a subproblem differs from its parent only in the fact that a single new constraint has been added, we can solve the linear programming relaxation of a subproblem by means of the dual simplex method, starting from $\mathbf{x}^*$. We may then expect than an optimal solution to the new linear programming problem will be obtained after only a small number of iterations.

Example 11.4 (Illustration of branch and bound) We solve the problem of Example 11.2 by branch and bound; see Figure 11.3. Initially, $U = \infty$. We solve the linear programming relaxation and the optimal solution is $\mathbf{x}^1 = (1.5, 2.5)$. Then, $b(F)$ is the optimal cost of the relaxation, i.e., $b(F) = -3.5$. We create two subproblems, by adding the constraints $x_2 \geq 3$ (subproblem F_1), or $x_2 \leq 2$ (subproblem F_2). The active list of subproblems is $\{F_1, F_2\}$. The linear programming relaxation of subproblem F_1 is infeasible and, therefore, we can delete this subproblem from the active list. The optimal solution to the linear programming

relaxation of subproblem F_2 is $\mathbf{x}^2 = (3/4, 2)$, and thus $b(F_2) = -3.25$. We further decompose subproblem F_2 into two subproblems, since either $x_1 \geq 1$ (subproblem F_3), or $x_1 \leq 0$ (subproblem F_4). The active list of subproblems is now $\{F_3, F_4\}$. The optimal solution to the linear programming relaxation of Subproblem F_3 is $\mathbf{x}^3 = (1, 2)$, which is integer and therefore, $U = -3$. We delete subproblem F_3 from the active list. The optimal solution to the linear programming relaxation of subproblem F_4 is $\mathbf{x}^4 = (0, 3/2)$, and thus $b(F_4) = -3$. Since $b(F_4) \geq U$, we do not need to further explore subproblem F_4. Since the active list of subproblems is empty, we terminate. The optimal integer solution is $\mathbf{x}^3 = (1, 2)$.

Example 11.5 (A branch and bound method for the directed traveling salesman problem) Given a directed graph $G = (\mathcal{N}, \mathcal{A})$ with n nodes, and a cost c_{ij} for every arc, we want to solve the traveling salesman problem on G using branch and bound. The objective is to find a tour (a directed cycle that visits all nodes) of minimum cost. We let x_{ij} equal to 1, if i and j are consecutive nodes in a tour, and 0, otherwise. The optimal cost in the problem

$$\begin{aligned} \text{minimize} \quad & \sum_{i=1}^{n} \sum_{j=1}^{n} c_{ij} x_{ij} \\ \text{subject to} \quad & \sum_{i=1}^{n} x_{ij} = 1, \qquad j = 1, \ldots, n, \\ & \sum_{j=1}^{n} x_{ij} = 1, \qquad i = 1, \ldots, n, \\ & x_{ij} \in \{0, 1\}, \end{aligned}$$

provides a lower bound on the cost of an optimal tour, because every tour must satisfy the above constraints. We recognize this as an assignment problem. However, not every feasible solution to the assignment problem corresponds to a tour, and for this reason the optimal costs for the two problems are not the same. In particular an optimal solution to the assignment problem may correspond to a collection of "subtours"; see Figure 11.4.

Suppose now that we use the assignment problem to obtain a lower bound on the cost of the traveling salesman problem. If the optimal solution to the assignment problem corresponds to a tour, such a tour is optimal for the traveling salesman problem. If not, we split the problem into subproblems. Each additional subproblem involves a single additional constraint of the form $x_{ij} = 0$. This is equivalent to prohibiting (i, j) from being consecutive nodes in a tour, and can be also accomplished by setting c_{ij} to a prohibitively high value. Note that adding such a constraint to the traveling salesman or to the assignment problem, still leaves us with a traveling salesman or assignment problem, respectively. Therefore, all subproblems constructed in the course of the branch and bound algorithm will also correspond to instances of the traveling salesman problem and lower bounds can be obtained by solving the related assignment problems. The only remaining issue is how to decide which constraints $x_{ij} = 0$ to add. A natural alternative is to choose one or more subtours and let each subproblem

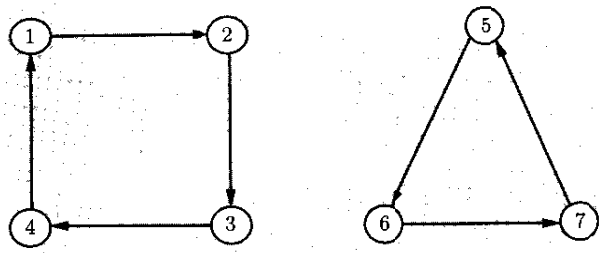

Figure 11.4: Consider a directed traveling salesman problem with seven nodes. The vector x corresponding to these two subtours is a feasible solution to the assignment problem.

prohibit one of their arcs. For example, if the optimal solution to the assignment problem is as in Figure 11.4, we can create subproblems by adding one of the constraints $x_{12} = 0$, $x_{23} = 0$, $x_{34} = 0$, $x_{41} = 0$, $x_{56} = 0$, $x_{67} = 0$, $x_{75} = 0$. If the current assignment problem has a unique optimal solution, this solution is made infeasible by the constraints that are added during branching. For this reason, the optimal cost in each subproblem is strictly larger, and improved lower bounds are obtained.

It should be clear that the success of branch and bound methods depends critically on the availability of tight lower bounds. (In Section 11.4 we introduce a duality theory for integer programming that leads to such bounds.) While the branch and bound algorithm may take exponential time in the worst case (see Exercise 11.4), it often produces acceptable solutions in a reasonably short amount of time, especially when tight lower bounds are available.

Branch and cut

Another variant of the method, often called *branch and cut*, utilizes cuts when solving the subproblems. In particular, we augment the formulation of the subproblems with additional cuts, in order to improve the bounds obtained from the linear programming relaxations. We illustrate the method with an example.

Example 11.6 (Illustration of branch and cut) We solve the problem of Example 11.2 by branch and cut. We first solve the linear programming relaxation and find the optimal solution $\mathbf{x}^1 = (1.5, 2.5)$. As before, we create subproblems F_1 (corresponding to $x_2 \geq 3$) and F_2 (corresponding to $x_2 \leq 2$). We delete subproblem F_1, because its linear programming relaxation is infeasible. In order to solve subproblem F_2, we add the constraint $-x_1 + x_2 \leq 1$ which is satisfied by all integer solutions to subproblem F_2. The optimal solution to the linear programming relaxation is now $\mathbf{x} = (1, 2)$, which is integer, and thus we terminate

with the optimal solution. Note that by adding the cut $-x_1 + x_2 \leq 1$, we avoide further enumeration. This is typical in branch and cut. If we can add "deep" cuts, we can accelerate branch and bound considerably. However, finding such cuts is nontrivial.

11.3 Dynamic programming

In the previous section, we introduced branch and bound, which is an exact, intelligent enumerative technique that attempts to avoid enumerating a large portion of the feasible integer solutions. In this section, we introduce another exact technique, called dynamic programming, that solves integer programming problems sequentially.

We illustrate the method by deriving a dynamic programming algorithm for the traveling salesman problem. We will then discuss general principles on how to develop dynamic programming algorithms for other integer programming problems.

Example 11.7 (A dynamic programming algorithm for the traveling salesman problem) Let $G = (\mathcal{N}, \mathcal{A})$ be a directed graph with n nodes and let c_{ij} be the cost of arc (i, j). We view the choice of a tour as a sequence of choices: we start at node 1; then, at each stage, we choose which node to visit next. After a number of stages, we have visited a subset S of $\mathcal{N}$ and we are at a current node $k \in S$. Let $C(S, k)$ be the minimum cost over all paths that start at node 1, visit all nodes in the set S exactly once, and end up at node k. If we call (S, k) a *state*, this state can be reached from any state of the form $\big(S \setminus \{k\}, m\big)$, with $m \in S \setminus \{k\}$, at a transition cost of c_{mk}. Thus, $C(S, k)$ can be interpreted as the least possible sum of transition costs, over all sequences of transitions that take us from state $\big(\{1\}, 1\big)$ to state (S, k). Therefore, we have the recursion

$$C(S,k) = \min_{m \in S \setminus \{k\}} \Big(C\big(S \setminus \{k\}, m\big) + c_{mk} \Big), \qquad k \in S, \tag{11.3}$$

and $C\big(\{1\}, 1\big) = 0$. There are 2^n choices for S, $O(n)$ choices for k, and a total of $O\big(n2^n\big)$ states (S, k). Each time that $C(S, k)$ is evaluated for some new state according to Eq. (11.3), $O(n)$ arithmetic operations are needed. Therefore, with $O\big(n^2 2^n\big)$ operations, we can obtain $C\big(\{1, \ldots, n\}, k\big)$ for all k. The length of an optimal tour is then given by

$$\min_k \Big(C\big(\{1, \ldots, n\}, k\big) + c_{k1} \Big).$$

This algorithm, although exponential, is much better than exhaustive enumeration of all $n!$ tours. Realistically, it can only be used to solve instances of the traveling salesman problem involving up to 20 nodes.

More generally, devising a dynamic programming algorithm for an integer programming problem involves the following steps.

Guidelines for constructing dynamic programming algorithms

1. View the choice of a feasible solution as a sequence of decisions occurring in stages, and so that the total cost is the sum of the costs of individual decisions.
2. Define the state as a summary of all relevant past decisions.
3. Determine which state transitions are possible. Let the cost of each state transition be the cost of the corresponding decision.
4. Write a recursion on the optimal cost from the origin state to a destination state.

The most crucial step is usually the definition of a suitable state. Let us apply the method to another problem.

A dynamic programming algorithm for the zero-one knapsack problem

Let us consider the version of the zero-one knapsack problem we introduced in Example 10.1:

$$\begin{aligned} \text{maximize} \quad & \sum_{j=1}^{n} c_j x_j \\ \text{subject to} \quad & \sum_{j=1}^{n} w_j x_j \leq K \\ & x_j \in \{0, 1\}. \end{aligned}$$

We assume that K and all c_j, w_j are positive integers. We derive a dynamic programming algorithm for the zero-one knapsack problem by decomposing it into stages. Instead of picking a vector $(x_1, \ldots, x_n)$ all at once, we visualize the problem as one in which decisions are made for one item at a time. After i decisions, we have decided which ones out of the first i items are to be included in the knapsack, and have therefore determined values for the variables $x_1, \ldots, x_i$. At that point, the value accumulated is $\sum_{j=1}^{i} c_j x_j$ and the weight accumulated is $\sum_{j=1}^{i} w_j x_j$.

Let $W_i(u)$ be the least possible weight that has to be accumulated in order to attain a total value of u using only items in the set $\{1, \ldots, i\}$. Let $W_i(u) = \infty$, if it is impossible to accumulate a total value of u using only the first i items. We use the convention $W_0(0) = 0$, and $W_0(u) = \infty$, if $u \neq 0$, which reflects the fact that the value accumulated using no items is zero. We then have the following recursion:

$$W_{i+1}(u) = \min \big\{ W_i(u),\ W_i(u - c_{i+1}) + w_{i+1} \big\}. \tag{11.4}$$

In words, this recursion means the following. If we wish to accumulate a total value of u, using some of the first $i + 1$ items, while accumulating as

little weight as possible, there are two alternatives depending on whether item $i+1$ is used or not. If item $i+1$ is not used, then the best we can do is to accumulate a total value of u, while using only some of the first i items and do that with the least possible accumulated weight, which is $W_i(u)$. Alternatively, if item $i+1$ is used, since it has a value of c_{i+1}, we must have accumulated a total value of $u - c_{i+1}$ using the first i items. Of course, the first i decisions should be made so that the value $u - c_{i+1}$ is accumulated with the least possible weight, which is $W_i(u - c_{i+1})$, and to which we must then add the weight of item $i+1$. We can now interpret recursion (11.4) as stating that $W_{i+1}(u)$ is given by the best of the two alternatives that we have just described.

We continue with a slightly different interpretation of recursion (11.4). Let us say that we are at *state* (i, u) if we have considered the first i items, have picked some of them, and have accumulated a total value of u. We then build a state transition diagram indicating which states can be reached from which other state. Notice that when in state (i, u) we can either decide to pick item $i+1$ and move to state $(i+1, u+c_{i+1})$ or we can decide to skip item $i+1$ and move to state $(i+1, u)$. We represent states as nodes and possible transitions by directed arcs; see Figure 11.5.

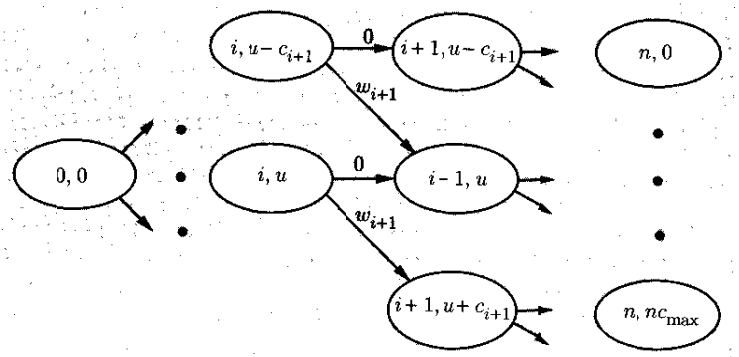

Figure 11.5: The state transition diagram for the dynamic programming approach to the zero-one knapsack problem.

In addition, we associate a weight to each arc (or transition) which is the additional weight added in the course of this transition. Thus, the transition from (i, u) to $(i+1, u)$ carries zero weight, while the transition from (i, v) to $(i+1, u+c_{i-1})$ carries weight w_{i+1}.

Initially, we are at state $(0, 0)$; no item has been considered and no value has been accumulated. A sequence of decisions, involving items $1, \ldots, i$ corresponds to a directed path from node $(0, 0)$ to some node of the form (i, u). Furthermore, the sum of the weights along the path corresponds to the accumulated weight. We conclude that $W_i(u)$ is equal to the least weight of all paths from node $(0, 0)$ to node (i, u), and is equal

to infinity if no such path exists. We may then recognize Eq. (11.4) as the Bellman equation associated with this shortest path problem (see Section 7.9).

Let

$$c_{\max} = \max_{i=1,\ldots,n} c_i.$$

If $u > nc_{\max}$, then no state of the form (i, u) is reachable. By restricting to states of the form (i, u) with $u \leq nc_{\max}$, we see that the total number of states of interest is of the order of $n^2 c_{\max}$. Using recursion (11.4), the value of $W_i(u)$ for all states of interest, can be computed in time $O(n^2 c_{\max})$. Once this is done, the optimal value u^* is given by

$$u^* = \max\{u \mid W_n(u) \leq K\},$$

which can be determined with only an additional $O(nc_{\max})$ effort. Optimal values for the variables $x_1, \ldots, x_n$ are then determined by an optimal path from node $(0, 0)$ to node (n, u^*). We have thus proved the following result.

Theorem 11.1 *The zero-one knapsack problem can be solved in time* $O(n^2 c_{\max})$.

An alternative, and somewhat more natural, dynamic programming algorithm for the same problem could be obtained by defining $C_i(w)$ as the maximum value that can be accumulated using some of the first i items subject to the constraint that the total accumulated weight is equal to w. We would then obtain the recursion

$$C_{i+1}(w) = \max\{C_i(w),\ C_i(w - w_{i+1}) + c_{i+1}\}.$$

By considering all states of the form (i, w) with $w \leq K$, an algorithm with complexity $O(nK)$ would be obtained. However, our previous algorithm is better suited to the purposes of developing an approximation algorithm, which will be done in Section 11.5.

The algorithm of Theorem 11.1 is an exponential time algorithm. This is because the size of an instance of the zero-one knapsack problem is

$$O\Big(n\big(\log c_{\max} + \log w_{\max}\big) + \log K\Big),$$

where $w_{\max} = \max_i w_i$. However, it becomes polynomial if $c_{\max}$ is bounded by some polynomial in n. More formally, for any integer d, we can define the problem KNAPSACK(d), as the problem consisting of all instances of the zero-one knapsack problem with $c_i \leq n^d$ for all i. According to Theorem 11.1, KNAPSACK(d) can be solved in time $O(n^{d+2})$, which is polynomial for every fixed d.

11.4 Integer programming duality

In this section, we develop the duality theory of integer programming. This in turn leads to a method for obtaining tight bounds, that are particularly useful for branch and bound. The methodology is closely related to the subject of Section 4.10, but our discussion here is self-contained.

We consider the integer programming problem

$$\begin{array}{ll} \text{minimize} & \mathbf{c}'\mathbf{x} \\ \text{subject to} & \mathbf{A}\mathbf{x} \geq \mathbf{b} \\ & \mathbf{D}\mathbf{x} \geq \mathbf{d} \\ & \mathbf{x} \text{ integer}, \end{array} \tag{11.5}$$

and assume that $\mathbf{A}$, $\mathbf{D}$, $\mathbf{b}$, $\mathbf{c}$, $\mathbf{d}$ have integer entries. Let Z_{IP} the optimal cost and let

$$X = \{\mathbf{x} \text{ integer} \mid \mathbf{D}\mathbf{x} \geq \mathbf{d}\}.$$

In order to motivate the method, we assume that optimizing over the set X can be done efficiently; for example X may represent the set of feasible solutions to an assignment problem. However, adding the constraints $\mathbf{A}\mathbf{x} \geq \mathbf{b}$ to the problem, makes the problem difficult to solve. We next consider the idea of introducing a dual variable for every constraint in $\mathbf{A}\mathbf{x} \geq \mathbf{b}$. Let $\mathbf{p} \geq \mathbf{0}$ be a vector of dual variables (also called *Lagrange multipliers*) that has the same dimension as the vector $\mathbf{b}$. For a fixed vector $\mathbf{p}$, we introduce the problem

$$\begin{array}{ll} \text{minimize} & \mathbf{c}'\mathbf{x} + \mathbf{p}'(\mathbf{b} - \mathbf{A}\mathbf{x}) \\ \text{subject to} & \mathbf{x} \in X, \end{array} \tag{11.6}$$

and denote its optimal cost by $Z(\mathbf{p})$. We will say that we *relax* or *dualize* the constraints $\mathbf{A}\mathbf{x} \geq \mathbf{b}$. For a fixed $\mathbf{p}$, the above problem can be solved efficiently, as we are optimizing a linear objective over the set X. We next observe that $Z(\mathbf{p})$ provides a bound on Z_{IP}.

Lemma 11.1 *If the problem (11.5) has an optimal solution and if $\mathbf{p} \geq \mathbf{0}$, then $Z(\mathbf{p}) \leq Z_{\text{IP}}$.*

Proof. Let $\mathbf{x}^*$ denote an optimal solution to (11.5). Then, $\mathbf{b} - \mathbf{A}\mathbf{x}^* \leq \mathbf{0}$ and, therefore,

$$\mathbf{c}'\mathbf{x}^* + \mathbf{p}'(\mathbf{b} - \mathbf{A}\mathbf{x}^*) \leq \mathbf{c}'\mathbf{x}^* = Z_{\text{IP}}.$$

Since $\mathbf{x}^* \in X$,

$$Z(\mathbf{p}) \leq \mathbf{c}'\mathbf{x}^* + \mathbf{p}'(\mathbf{b} - \mathbf{A}\mathbf{x}^*),$$

and therefore, $Z(\mathbf{p}) \leq Z_{\text{IP}}$. □

Since problem (11.6) provides a lower bound to the integer programming problem (11.5) for all $\mathbf{p} \geq \mathbf{0}$, it is natural to consider the tightest such

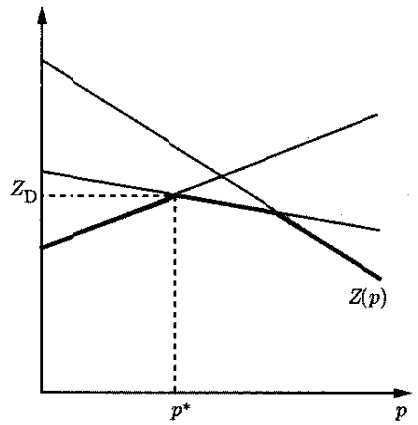

Figure 11.6: The function $Z(\mathbf{p})$ is concave and piecewise linear.

bound. For this reason, we introduce the problem

$$\begin{array}{ll} \text{maximize} & Z(\mathbf{p}) \\ \text{subject to} & \mathbf{p} \geq \mathbf{0}. \end{array} \tag{11.7}$$

We will refer to problem (11.7) as the *Lagrangean dual*. Let

$$Z_{\text{D}} = \max_{\mathbf{p} \geq \mathbf{0}} Z(\mathbf{p}).$$

Suppose for instance, that $X = \{\mathbf{x}^1, \ldots, \mathbf{x}^m\}$. Then $Z(\mathbf{p})$ can be also written as

$$Z(\mathbf{p}) = \min_{i=1,\ldots,m} \left(\mathbf{c}'\mathbf{x}^i + \mathbf{p}'(\mathbf{b} - \mathbf{A}\mathbf{x}^i)\right). \tag{11.8}$$

The function $Z(\mathbf{p})$ is concave and piecewise linear, since it is the minimum of a finite collection of linear functions of $\mathbf{p}$ (see Theorem 1.1 in Section 1.3 and Figure 11.6). As a consequence, the problem of computing Z_{D} [namely, problem (11.7)] can be recast as a linear programming problem, but with a very large number of constraints.

It is clear from Lemma 11.1 that weak duality holds:

Theorem 11.2 *We have $Z_{\text{D}} \leq Z_{\text{IP}}$.*

The previous theorem represents the weak duality theory of integer programming. Unlike linear programming, integer programming does not have a strong duality theory. (Compare with Theorem 4.18 in Section 4.10.)

Indeed in Example 11.8, we show that it is possible to have $Z_D < Z_I$. The procedure of obtaining bounds for integer programming problems b calcuating Z_D is called *Lagrangean relaxation*. We next investigate t quality of the bound Z_D, in comparison to the one provided by the line programming relaxation of problem (11.5).

On the strength of the Lagrangean dual

The characterization (11.8) of the Lagrangean dual objective does not pr vide particular insight into the quality of the bound. A more revealin characterization is developed in this subsection. Let $\mathrm{CH}(X)$ be the conve hull of the set X. We need the following result, whose proof is outline in Exercise 11.8. Since we already know that the convex hull of a finit set is a polyhedron, this result is of interest when the set $\{\mathbf{x} \mid \mathbf{Dx} \geq \mathbf{d}\}$ unbounded and the set X is infinite.

Theorem 11.3 *We assume that the system of linear inequalities* $\mathbf{Dx} \geq \mathbf{d}$ *has a feasible solution, and that the matrix* $\mathbf{D}$ *and the vector* $\mathbf{d}$ *hav integer entries. Let*

$$X = \{\mathbf{x} \text{ integer} \mid \mathbf{Dx} \geq \mathbf{d}\}.$$

Then $\mathrm{CH}(X)$ *is a polyhedron.*

The next theorem, which is the central result of this section, characterizes the Lagrangean dual as a linear programming problem.

Theorem 11.4 *The optimal value* Z_D *of the Lagrangean dual is equal to the optimal cost of the following linear programming problem:*

$$\begin{array}{ll} \text{minimize} & \mathbf{c}'\mathbf{x} \\ \text{subject to} & \mathbf{Ax} \geq \mathbf{b} \\ & \mathbf{x} \in \mathrm{CH}(X). \end{array} \tag{11.9}$$

Proof. By definition,

$$Z(\mathbf{p}) = \min_{\mathbf{x} \in X} \left(\mathbf{c}'\mathbf{x} + \mathbf{p}'(\mathbf{b} - \mathbf{Ax})\right).$$

Since the objective function is linear in $\mathbf{x}$, the optimal cost remains the same if we allow convex combinations of the elements of X. Therefore,

$$Z(\mathbf{p}) = \min_{\mathbf{x} \in \mathrm{CH}(X)} \left(\mathbf{c}'\mathbf{x} + \mathbf{p}'(\mathbf{b} - \mathbf{Ax})\right),$$

and hence, we have

$$Z_D = \max_{\mathbf{p} \geq 0} \min_{\mathbf{x} \in \mathrm{CH}(X)} \left(\mathbf{c}'\mathbf{x} + \mathbf{p}'(\mathbf{b} - \mathbf{Ax})\right).$$

Let $\mathbf{x}^k$, $k \in K$, and $\mathbf{w}^j$, $j \in J$, be the extreme points and a complete set of extreme rays of $\mathrm{CH}(X)$, respectively. Then, for any fixed $\mathbf{p}$, we have

$$Z(\mathbf{p}) = \begin{cases} -\infty, & \text{if } (\mathbf{c}' - \mathbf{p}'\mathbf{A})\mathbf{w}^j < 0, \\ & \text{for some } j \in J, \\ \min\limits_{k \in K} \left(\mathbf{c}'\mathbf{x}^k + \mathbf{p}'(\mathbf{b} - \mathbf{Ax}^k)\right), & \text{otherwise.} \end{cases}$$

Therefore, the Lagrangean dual is equivalent to and has the same optimal value as the problem

$$\begin{array}{lll} \text{maximize} & \min\limits_{k \in K} \left(\mathbf{c}'\mathbf{x}^k + \mathbf{p}'(\mathbf{b} - \mathbf{Ax}^k)\right) & \\ \text{subject to} & (\mathbf{c}' - \mathbf{p}'\mathbf{A})\mathbf{w}^j \geq 0, & j \in J, \\ & \mathbf{p} \geq \mathbf{0}, & \end{array}$$

or equivalently,

$$\begin{array}{lllll} \text{maximize} & y & & & \\ \text{subject to} & y + \mathbf{p}'(\mathbf{Ax}^k - \mathbf{b}) & \leq \mathbf{c}'\mathbf{x}^k, & & k \in K, \\ & \mathbf{p}'\mathbf{Aw}^j & \leq \mathbf{c}'\mathbf{w}^j, & & j \in J, \\ & \mathbf{p} \geq \mathbf{0}. & & & \end{array}$$

Taking the linear programming dual of the above problem, and using strong duality, we obtain that Z_D is equal to the optimal cost of the problem

$$\begin{array}{ll} \text{minimize} & \mathbf{c}' \left(\sum\limits_{k \in K} \alpha_k \mathbf{x}^k + \sum\limits_{j \in J} \beta_j \mathbf{w}^j \right) \\ \text{subject to} & \sum\limits_{k \in K} \alpha_k = 1 \\ & \mathbf{A} \left(\sum\limits_{k \in K} \alpha_k \mathbf{x}^k + \sum\limits_{j \in J} \beta_j \mathbf{w}^j \right) \geq \mathbf{b} \\ & \alpha_k, \beta_j \geq 0, \qquad k \in K, \ j \in J. \end{array}$$

Since,

$$\mathrm{CH}(X) = \left\{ \sum_{k \in K} \alpha_k \mathbf{x}^k + \sum_{j \in J} \beta_j \mathbf{w}^j \quad \sum_{k \in K} \alpha_k = 1, \ \alpha_k, \beta_j \geq 0, \ k \in K, \ j \in J \right\},$$

the result follows. □