6

Complexity and Problem Reductions

6.1 COMPLEXITY

If we consider a list of the problems we have examined so far, we have either shown or it can be shown that the following have the Efficient Optimization Property:

The Uncapacitated Lot-Sizing Problem (Chapter 5)
The Maximum Weight Tree Problem (Chapter 3)
The Maximum Weight Matching Problem
The Shortest Path Problem (Chapter 5)
The Max Flow Problem (Chapter 3)
The TU Integer Programming Problem (Chapter 3)
The Assignment Problem (Chapter 4)

Below we make this more precise: there is a polynomial algorithm for these optimization problems.

On the other hand, no one to date has found an efficient (polynomial) algorithm for any of the following optimization problems:

The 0–1 Knapsack Problem (Chapter 1)
The Set Covering Problem (Chapter 1)
The Traveling Salesman Problem (Chapter 1)
The Uncapacitated Facility Location Problem (Chapter 1)
The Integer Programming Problem (Chapter 1)
The Steiner Tree Problem (Chapter 3)

The remainder of this book will in large part be devoted to examining how to tackle problems in this second group. However, it is first useful to discuss the distinction (real or imaginary) between these two groups, so that when we encounter a new optimization problem we have an idea how we might classify and then attempt to solve it.

To develop such a method of classification, we need just four concepts:

A class $\mathcal{C}$ of legitimate problems to which the theory applies
A nonempty subclass $\mathcal{C}_A \subseteq \mathcal{C}$ of "easy" problems
A nonempty subclass $\mathcal{C}_B \subseteq \mathcal{C}$ of "difficult" problems
A relation "not more difficult than" between pairs of legitimate problems.

This immediately leads to:

Proposition 6.1 *(Reduction Lemma) Suppose that P and Q are two legitimate problems.*
If Q is "easy" and P is "not more difficult than" Q, then P is "easy".
If P is "difficult" and P is "not more difficult than" Q, then Q is "difficult".

We have already used the first part of the lemma implicitly in Chapter 4. There we show that the maximum weight bipartite matching problem is "easy" by showing that it is reducible to the assignment problem. Also Exercise 4.4 involves showing that the Chinese postman problem is reducible to maximum weight matching. The goal of the rest of this chapter is to somewhat formalize these notions. In the next section we introduce the class of legitimate problems and the "easy" class, and in Section 6.3 we discuss the concept of problem reduction which allows us to then define the "difficult" class. By the end of the chapter we will then have another tool at our disposal: namely a way to show that certain problems are "difficult" by using the second part of the reduction lemma.

6.2 DECISION PROBLEMS, AND CLASSES $\mathcal{NP}$ AND $\mathcal{P}$

Unfortunately, the theory does not exactly address optimization problems in the form we have posed them so far. To define the class of legitimate problems, it is appropriate to pose decison problems having YES-NO answers. Thus an optimization problem:

$$\max\{cx : x \in S\}$$

for which an instance consists of: {c and a "standard" representation of S} is replaced by the *decision problem*:

$$\text{Is there an } x \in S \text{ with value } cx \geq k?$$

for which an instance consists of {c, a "standard" representation of S, and an integer k}.

So for the rest of this chapter (unless explicitly stated), when we refer to an optimization problem *TSP*, *UFL*, and so on, we have in mind the corresponding decision problem.

Next we give a slightly more formal definition of the running time of an algorithm than that given at the start of Chapter 3. It is not just the number of variables and constraints or nodes and edges that defines the input length, but also the size of the numbers occurring in the data.

Definition 6.1 For a problem instance X, the *length of the input* $L = L(X)$ is the length of the binary representation of a "standard" representation of the instance.

Definition 6.2 Given a problem P, an algorithm A for the problem, and an instance X, let $f_A(X)$ be the number of elementary calculations required to run the algorithm A on the instance X. $f_A^*(l) = \sup_X\{f_A(X) : L(X) = l\}$ is the *running time* of algorithm A. An algorithm A is *polynomial* for a problem P if $f_A^*(l) = O(l^p)$ for some positive integer p.

Now we can define the class of "legitimate" problems.

Definition 6.3 $\mathcal{NP}$ is the class of decision problems with the property that: for any instance for which the answer is YES, there is a "short" (polynomial) proof of the YES.

We note immediately that if a decision problem associated to an optimization problem is in $\mathcal{NP}$, then the optimization problem can be solved by answering the decision problem a polynomial number of times (by using bisection on the objective function value).

Proposition 6.2 *For each optimization problem in the two lists in Section 6.1, the associated decision problem: "Does there exist a primal solution of value as good as or better than k?" lies in* $\mathcal{NP}$.

Now we can define the class of "easy" problems.

Definition 6.4 $\mathcal{P}$ is the class of decision problems in $\mathcal{NP}$ for which there exists a polynomial algorithm.

Example 6.1 (i) Uncapacitated Lot Sizing. Consider *ULS* for which a dynamic programming algorithm is presented in Chapter 5. For an instance X with integral data (n, d, p, h, f, k), the input has length $L(X) = \sum_{j=1}^n \lceil \log d_j \rceil + \sum_{j=1}^n \lceil \log p_j \rceil + \sum_{j=1}^n \lceil \log h_j \rceil + \sum_{j=1}^n \lceil \log f_j \rceil + \lceil \log k \rceil$.

The DP algorithm requires only $O(n^2)$ additions and comparisons of the numbers occurring in the data, and hence the size of numbers required to give a YES answer, and the running time are certainly $O(L^2)$. Thus *ULS* is in $\mathcal{P}$.

(ii) 0–1 Knapsack. For an instance X of 0–1 *KNAPSACK*: $\{\sum_{j=1}^{n} c_j x_j \geq k, \sum_{j=1}^{n} a_j x_j \leq b, x \in \{0,1\}^n\}$, the length of the input is $L(X) = \sum_{j=1}^{n} \lceil \log c_j \rceil + \sum_{j=1}^{n} \lceil \log a_j \rceil + \lceil \log b \rceil + \lceil \log k \rceil$.

For an instance for which the answer is YES, it suffices to (a) read a solution $x^* \in \{0,1\}^n$, and (b) check that $ax^* \leq b$ and $cx^* \geq k$. Both (a) and (b) can be carried out in time polynomial in L, so the associated decision problem is in $\mathcal{NP}$.

From Section 5.4, dynamic programming provides an $O(nb)$ algorithm. As b is not equal to $(\log b)^p$ for any fixed p, this algorithm is not polynomial, and in fact no polynomial algorithm is known for 0–1 *KNAPSACK*.

(iii) Symmetric Traveling Salesman. For *STSP* with instance $(G, c^{|E|}, k)$, it suffices to check that a proposed set of edges forms a tour and that its length does not exceed k. The argument for most other problems on the second list is similar.

(iv) Integer Programming. Problem *IP* requires a little more work, because one needs to show that there always exists an optimal solution x^* whose description length $\sum_{j=1}^{n} \lceil \log x_j^* \rceil$ is polynomial in L. ∎

Do the second set of optimization problems listed in Section 6.1 above have anything in common apart from the fact that their decision problems are in $\mathcal{NP}$, and that nobody has yet discovered a polynomial algorithm for any of them?

Surprisingly, they have a second property in common: their decision problems are all among the *most difficult problems* in $\mathcal{NP}$.

6.3 POLYNOMIAL REDUCTION AND THE CLASS $\mathcal{NPC}$

This is the formal definition of "is not more difficult than" that we need.

Definition 6.5 If P, $Q \in \mathcal{NP}$, and if an instance of P can be converted in polynomial time to an instance of Q, then P is *polynomially reducible* to Q.

Note that this means that if we have an algorithm for problem Q, it can be used to solve problem P with an overhead that is polynomial in the size of the instance. We now define the class of "most difficult" problems.

Definition 6.6 $\mathcal{NPC}$, the class of $\mathcal{NP}$*-complete* problems, is the subset of problems $P \in \mathcal{NP}$ such that for all $Q \in \mathcal{NP}$, Q is polynomially reducible to P.

It is a remarkable fact not only that $\mathcal{NPC}$ is nonempty, but that all of the decision problems in our second list are in $\mathcal{NPC}$. So how can one prove that a problem is in $\mathcal{NPC}$?

The most important step is to prove that $\mathcal{NPC}$ is nonempty. Written as an 0–1 integer program, *SATISFIABILITY* is the decision problem:

Given $N = \{1, \ldots, n\}$, and $2m$ subsets $\{C_i\}_{i=1}^m$ and $\{D_i\}_{i=1}^m$ of N, does the 0–1 integer program:

$$\sum_{j \in C_i} x_j + \sum_{j \in D_i} (1 - x_j) \geq 1 \text{ for } i = 1, \ldots, m$$
$$x \in B^n$$

have a feasible solution?

It is obvious that this problem is in $\mathcal{NP}$. Cook showed in 1970 that *SATISFIABILITY* is in $\mathcal{NPC}$.

Now we indicate how the reduction lemma can be used to show that all the problems of the second list and many others are in $\mathcal{NPC}$.

For example, to see that *BIP* is in $\mathcal{NPC}$, all we need to observe is that

(i) $BIP \in \mathcal{NP}$, (which is immediate) and

(ii) *SATISFIABILITY* reduces to *BIP*. (Above we actually described *SATISFIABILITY* as a *BIP*, and so this is also immediate).

We now restate the Reduction Lemma (Proposition 6.1) more formally.

Proposition 6.3 *Suppose that problems $P, Q \in \mathcal{NP}$.*
(i) If $Q \in \mathcal{P}$ and P is polynomially reducible to Q, then $P \in \mathcal{P}$.
(ii) If $P \in \mathcal{NPC}$ and P is polynomially reducible to Q, then $Q \in \mathcal{NPC}$.

Proof. (ii) Consider any problem $R \in \mathcal{NP}$. As $P \in \mathcal{NPC}$, R is polynomially reducible to P. However P is polynomially reducible to Q by hypothesis, and thus R is polynomially reducible to Q. As this holds for all $R \in \mathcal{NP}$, $Q \in \mathcal{NPC}$. ∎

This has an important corollary.

Corollary 6.1 *If $\mathcal{P} \cap \mathcal{NPC} \neq \emptyset$, then $\mathcal{P} = \mathcal{NP}$.*

Proof. Suppose $Q \in \mathcal{P} \cap \mathcal{NPC}$ and take $R \in \mathcal{NP}$. By (ii), as $R \in \mathcal{NP}$ and $Q \in \mathcal{NPC}$, R is polynomially reducible to Q. By (i), as $Q \in \mathcal{NP}$ and R is polynomially reducible to Q, $R \in \mathcal{P}$. So $\mathcal{NP} \subseteq \mathcal{P}$ and thus $\mathcal{P} = \mathcal{NP}$. ∎

The list of problems known to be in $\mathcal{NPC}$ is now enormous. Some of the most basic problems in $\mathcal{NPC}$ are the problems in our second list. Below we prove that the *Capacitated Lot-Sizing Problem* (*CLS*) can be added to the list.

Example 6.2 Consider the lot-sizing problem introduced in Chapter 1, with a production capacity constraint in each period. *CLS* has a formulation:

$$\begin{aligned} \min \sum_{t=1}^{n} p_t x_t \quad &+ \quad \sum_{t=1}^{n} h_t s_t + \sum_{t=1}^{n} f_t y_t \\ s_{t-1} + x_t \quad &= \quad d_t + s_t \text{ for } t = 1, \ldots, n \\ x_t \quad &\leq \quad C_t y_t \text{ for } t = 1, \ldots, n \\ s_0 = s_n = 0, s \quad &\in \quad R_+^{n+1}, x \in R_+^n, y \in B^n. \end{aligned}$$

First, is the decision version of *CLS* in $\mathcal{NP}$? One answer lies in the observation that there is always an optimal solution of the form: $y \in \{0,1\}^n$ with x a basic feasible solution of the network flow problem in which y is fixed. So there exists an optimal solution whose length is polynomial in the length of the input, and can be used to verify a YES answer in polynomial time. One just needs to check that it satisfes the constraints and its value is sufficiently small.

We now show that 0–1 *KNAPSACK* is polynomially reducible to *CLS*. To do this, we show how an instance of the 0–1 knapsack problem can be solved as a capacitated lot-sizing problem. Given an instance:

$$\min\{\sum_{j=1}^{n} c_j y_j : \sum_{j=1}^{n} a_j y_j \geq b, y \in B^n\},$$

we solve a lot-sizing instance with n periods, $p_t = h_t = 0, f_t = c_t, C_t = a_t$ for all t, d_t=0 for $t = 1, \ldots, n-1$ and $d_n = b$.

An equivalent formulation of the lot-sizing problem, obtained by eliminating the stock variables as described in Section 1.4, is:

$$\begin{aligned} \min \sum_{t=1}^{n} p'_t x_t \quad &+ \quad \sum_{t=1}^{n} f_t y_t \\ \sum_{i=1}^{t} x_i \quad &\geq \quad \sum_{i=1}^{t} d_i \text{ for } t = 1, \ldots, n-1 \\ \sum_{i=1}^{n} x_i \quad &= \quad \sum_{i=1}^{n} d_i \\ x_t \quad &\leq \quad C_t y_t \text{ for } t = 1, \ldots, n \\ x \in R_+^n, y \quad &\in \quad B^n. \end{aligned}$$

Rewriting this with the chosen values for the data, we obtain:

$$\begin{aligned} \min \sum_{t=1}^{n} c_t y_t \\ \sum_{i=1}^{t} x_i \quad &\geq \quad 0 \text{ for } t = 1, \ldots, n-1 \end{aligned}$$

$$\begin{aligned} \sum_{i=1}^{n} x_i &= b \\ x_t &\leq a_t y_t \text{ for } t = 1, \ldots, n \\ x \in R_+^n, y &\in B^n. \end{aligned}$$

Dropping the $n-1$ redundant demand constraints leaves

$$\begin{aligned} \min \sum_{t=1}^{n} c_t y_t & \\ \sum_{t=1}^{n} x_t &= b \\ x_t &\leq a_t y_t \text{ for } t = 1, \ldots, n \\ x &\in R_+^n, y \in B^n. \end{aligned}$$

Now let (x^*, y^*) be an optimal solution of this lot-sizing instance. Combining the constraint $\sum_{t=1}^{n} x_t = b$ with the constraints $x_t \leq a_t y_t$ for $t = 1, \ldots, n$, we see that $\sum_t a_t y_t^* \geq b$ and so y^* is feasible in the knapsack instance. It is also optimal, because a better knapsack solution $\overline{y}$ with $c\overline{y} < cy^*$ would also provide a better lot-sizing solution $((\overline{x})_t = a_t \overline{y}_t, \overline{y})$. So an optimal y vector for the lot-sizing instance solves the knapsack instance. So 0–1 *KNAPSACK* is polynomially reducible to *CLS*, and as 0–1 *KNAPSACK* $\in \mathcal{NPC}$, *CLS* $\in \mathcal{NPC}$. ∎

6.4 CONSEQUENCES OF $\mathcal{P} = \mathcal{NP}$ OR $\mathcal{P} \neq \mathcal{NP}$

Most problems of interest have either been shown to be in $\mathcal{P}$ or in $\mathcal{NPC}$. What is more, nobody has succeeded either in proving that $\mathcal{P} = \mathcal{NP}$ or in showing that $\mathcal{P} \neq \mathcal{NP}$. However, given the huge number of problems in $\mathcal{NPC}$ for which no polynomial algorithm has been found, it is a practical working hypothesis that $\mathcal{P} \neq \mathcal{NP}$.

So how should we interpret the above results and observations?

A first important remark concerns the class $\mathcal{NP}$. Typically, problems in this class have a very large (exponentially large) set of feasible solutions, and these problems can in theory be solved by enumerating the feasible solutions. As we saw in Table 1.1, this is impractical for instances of any reasonable size.

A *pessimist* might say that as most problems appear to be hard (i.e., their decision version lies in $\mathcal{NPC}$), we have no hope of solving instances of large size (because in the worst case we cannot hope to do better than enumeration), and so we should give up.

A *mathematician (optimist)* might set out to become famous by proving that $\mathcal{P} = \mathcal{NP}$

A *mathematician (pessimist)* might set out to become famous by proving that $\mathcal{P} \neq \mathcal{NP}$

A *mathematician (thoughtful)* might decide to ask a different question: Can I find an algorithm that is guaranteed to find a solution "close to optimal" in polynomial time in all cases?

A *probabilist (thoughtful)* might also ask a different question: Can I find an algorithm that runs in polynomial time with high probability and that is guaranteed to find an optimal or "close to optimal" solution with high probability?

An *engineer* would start looking for a heuristic algorithm that produces practically usable solutions.

Your *boss* might say: I don't care a damn about integer programming theory. You just worry about our scheduling problem. Give me a feasible production schedule for tomorrow in which William Brown and Daughters' order is out of the door by 4 P.M.

A *struggling professor* might say: Great. Previously I was trying to develop one algorithm to solve all integer programs, and publishing one paper every two years explaining why I was not succeeding. Now I know that I might as well study each $\mathcal{NP}$ problem individually. As there are thousands of them, I should be able to write twenty papers a year.

Needless to say they are nearly all right. There is no easy and rapid solution, but the problems will not go away, and more and more fascinating and important practical problems are being formulated as integer programs. So in spite of the $\mathcal{NP}$-completness theory, using an appropriate combination of theory, algorithms, experience, and intensive calculation, verifiably good solutions for large instances can and must be found.

Definition 6.7 An optimization problem for which the decision problem lies in $\mathcal{NPC}$ is called *$\mathcal{NP}$-hard.*

The following chapters are devoted to ways to tackle such *$\mathcal{NP}$-hard* problems. First, however, we return briefly to the *Separation Problem* introduced in Chapter 2.

6.5 OPTIMIZATION AND SEPARATION

Here we consider the question of whether there are ties between problems in $\mathcal{P}$. How can we show that a problem is in $\mathcal{P}$? The most obvious way is by finding a polynomial algorithm. We have also seen that another indirect way is by reduction.

There is, however, one general and important result tying together pairs of problems. Put imprecisely, it says:

Given a family of polyhedra associated with a class of problems (such as the convex hulls of the incidence vectors of feasible points $S \subseteq B^n$), the family of

optimization problems:

$$\max\{cx : x \in \text{conv}(S)\}$$

is polynomially solvable if and only if the family of separation problems:

Is $x \in \text{conv}(S)$? If not, find an inequality satisfied by all points of S, but cutting off x.

is polynomially solvable.

In other words, the Efficient Optimization and Efficient Separation Properties introduced in Chapter 3 are really equivalent. The other two properties are not exactly equivalent. As we indicated earlier, if a problem has the Efficient Separation Property, it suggests that it may have the Explicit Convex Hull Property. Also if a problem has the Explicit Convex Hull Property, then its linear programming dual may lead to the Strong Dual Property.

6.6 NOTES

6.2 An important step in the development of the distinction between easy and difficult problems is the concept of a certificate of optimality [Edm65a], [Edm65b].

6.3 Cook [Coo71] formally introduced the class $\mathcal{NP}$ and showed the existence of an $\mathcal{NP}$-complete problem. The reduction of many decision versions of integer and combinatorial optimization problems to a $\mathcal{NP}$-complete problem was shown in [Karp72], [Karp75]. The book [GarJoh79] lists an enormous number of $\mathcal{NP}$-complete problems and their reductions. A recent update is [CreKan95].

6.5 The equivalence of optimization and separation is shown in [GroLovSch81], [GroLovSch84]. A more thorough exploration of the equivalence appears in [GroLovSch88]. Other results of importance for integer programming concern the difficulty of finding a short description of all facets for $\mathcal{NP}$-hard problems [PapYan84], and the polynomiality of integer programming with a fixed number of variables [Len83]. Some separation problems are examined in Chapter 9.

For a general book on computational complexity, see [Pap94].

6.7 EXERCISES

1. The 2-*PARTITION* problem is specified by n positive integers $(a_1, \ldots, a_n)$. The problem is to find a subset $S \subset N = \{1, \ldots, n\}$ such that $\sum_{j \in S} a_j =$

$\sum_{j \in N \setminus S} a_j$, or prove that it is impossible. Show that 2-*PARTITION* is polynomially reducible to 0–1 *KNAPSACK*. Does this imply that 2-*PARTITION* is $\mathcal{NP}$-complete?

2. Show that *SATISFIABILITY* is polynomially reducible to *STABLE SET* (Node Packing), and thus that *STABLE SET* is $\mathcal{NP}$-complete, where *STABLE SET* is the problem of finding a maximum weight set of nonadjacent nodes in a graph.

3. Show that *STABLE SET* is polynomially reducible to *SET PACKING*, where *SET PACKING* is the problem of finding a maximum weight set of disjoint columns in a 0–1 matrix.

4. Show that *SET COVERING* is polynomially reducible to *UFL*.

5. Show that *SET COVERING* is polynomially reducible to *DIRECTED STEINER TREE*.

6. Given $D = (V, A)$, c_e for $e \in A$, a subset $F \subseteq A$, and a node $r \in V$, *ARC ROUTING* is the problem of finding a minimum length directed subtour that contains the arcs in F and starts and ends at node r. Show that *TSP* is polynomially reducible to *ARC ROUTING*.

7.* Show that the decision problem associated to *IP* is an integer programming feasibility problem, and is in $\mathcal{NP}$.

8. Consider a 0–1 knapsack set $X = \{x \in B^n : \sum_{j \in N} a_j x_j \leq b\}$ with $0 \leq a_j \leq b$ for $j \in N$ and let $\{x^t\}_{t=1}^T$ be the points of X. With it, associate the bounded polyhedron $\Pi^1 = \{\pi \in R_+^n : x^t \pi \leq 1 \text{ for } t = 1, \ldots, T\}$ with extreme points $\{\pi^s\}_{s=1}^S$. Consider a point x^* with $0 \leq x_j^* \leq 1$ for $j \in N$.

(i) Show that $x^* \in \text{conv}(X)$ if and only if $\min\{\sum_{t=1}^T \lambda_t : x^* \leq \sum_{t=1}^T x^t \lambda_t, \lambda \in R_+^T\} = \max\{x^* \pi : \pi \in \Pi^1\} \leq 1$.
(ii) Deduce that if $x^* \notin \text{conv}(X)$, then for some $s = 1, \ldots, S$, $\pi^s x^* > 1$.

7

Branch and Bound

7.1 DIVIDE AND CONQUER

Consider the problem:

$$z = \max\{cx : x \in S\}.$$

How can we break the problem into a series of smaller problems that are easier, solve the smaller problems, and then put the information together again to solve the original problem?

Proposition 7.1 *Let $S = S_1 \cup \ldots \cup S_K$ be a decomposition of S into smaller sets, and let $z^k = \max\{cx : x \in S_k\}$ for $k = 1, \ldots, K$. Then $z = \max_k z^k$.*

A typical way to represent such a divide and conquer approach is via an enumeration tree. For instance, if $S \subseteq \{0,1\}^3$, we might construct the enumeration tree shown in Figure 7.1.

Here we first divide S into $S_0 = \{x \in S : x_1 = 0\}$ and $S_1 = \{x \in S : x_1 = 1\}$, then $S_{00} = \{x \in S_0 : x_2 = 0\} = \{x \in S : x_1 = x_2 = 0\}$, $S_{01} = \{x \in S_0 : x_2 = 1\}$, and so on. Note that a leaf of the tree $S_{i_1 i_2 i_3}$ is nonempty if and only if $x = (i_1, i_2, i_3)$ is in S. Thus the leaves of the tree correspond precisely to the points of B^3 that one would examine if one carried out complete enumeration. Note that by convention the tree is drawn upside down with its root at the top.

Another example is the enumeration of all the tours of the traveling salesman problem. First we divide S the set of all tours on 4 cities into $S_{(12)}, S_{(13)}, S_{(14)}$ where $S_{(ij)}$ is the set of all tours containing arc (ij). Then $S_{(12)}$ is divided again into $S_{(12)(23)}$ and $S_{(12)(24)}$, and so on. Note that at the first level we have arbitrarily chosen to branch on the arcs leaving node 1, and at the

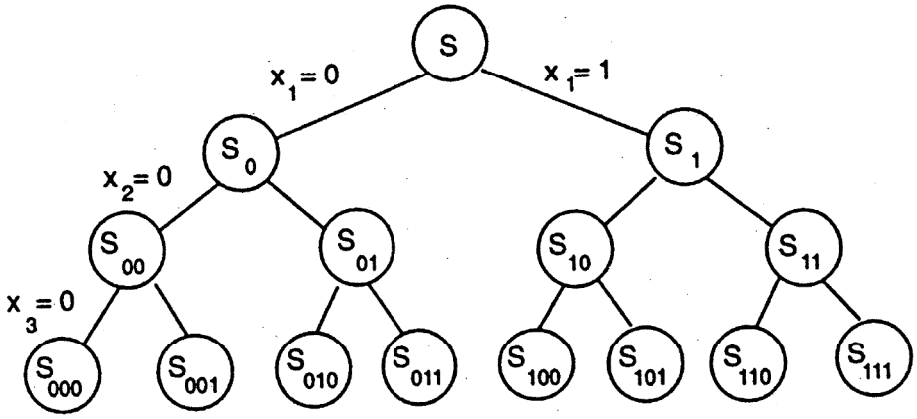

Fig. 7.1 Binary enumeration tree

second level on the arcs leaving node 2 that do not immediately create a subtour with the previous branching arc. The resulting tree is shown in Figure 7.2. Here the six leaves of the tree correspond to the $(n-1)!$ tours shown, where $i_1 i_2 i_3 i_4$ means that the cities are visited in the order i_1, i_2, i_3, i_4, i_1 respectively. Note that this is an example of multiway as opposed to binary branching, where a set can be divided into more than two parts.

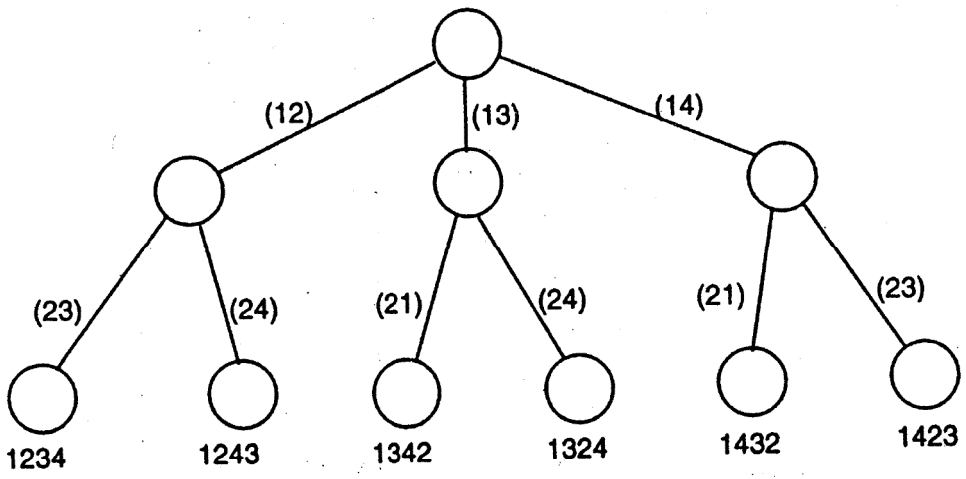

Fig. 7.2 *TSP* enumeration tree

7.2 IMPLICIT ENUMERATION

We saw in Chapter 1 that complete enumeration is totally impossible for most problems as soon as the number of variables in an integer program, or nodes in a graph exceeds 20 or 30. So we need to do more than just divide indefinitely. How can we use some bounds on the values of $\{z^k\}$ intelligently? First, how can we put together bound information?

Proposition 7.2 *Let $S = S_1 \cup \ldots \cup S_K$ be a decomposition of S into smaller sets, and let $z^k = \max\{cx : x \in S_k\}$ for $k = 1, \ldots, K$, $\overline{z}^k$ be an upper bound on z^k and $\underline{z}^k$ be a lower bound on z^k. Then $\overline{z} = \max_k \overline{z}^k$ is an upper bound on z and $\underline{z} = \max_k \underline{z}^k$ is a lower bound on z.*

Now we examine three hypothetical examples to see how bound information, or partial information about a subproblem can be put to use. What can be deduced about lower and upper bounds on the optimal value z and which sets need further examination in order to find the optimal value?

Example 7.1 In Figure 7.3 we show a decomposition of S into two sets S_1 and S_2 as well as upper and lower bounds on the corresponding problems.

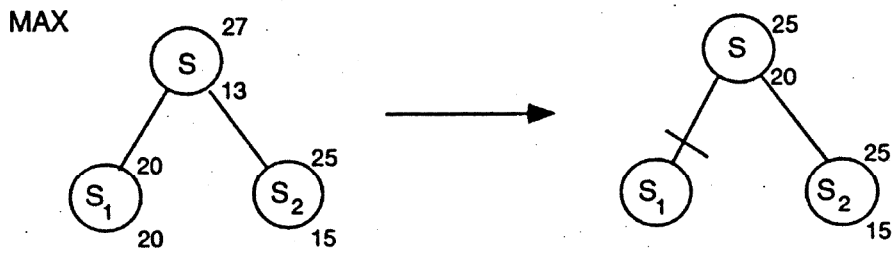

Fig. 7.3 Pruned by optimality

We note first that $\overline{z} = \max_k \overline{z}^k = \max\{20, 25\} = 25$ and $\underline{z} = \max_k \underline{z}^k = \max\{20, 15\} = 20$.

Second, we observe that as the lower and upper bounds on z_1 are equal, $z_1 = 20$, and there is no further reason to examine the set S_1. Therefore the branch S_1 of the enumeration tree can be pruned *by optimality.* ∎

Example 7.2 In Figure 7.4 we again decompose S into two sets S_1 and S_2 and show upper and lower bounds on the corresponding problems.

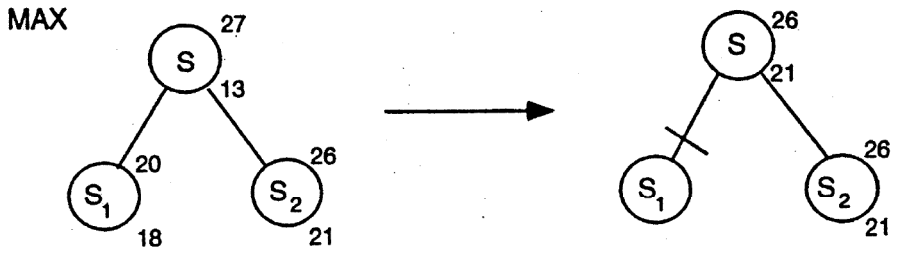

Fig. 7.4 Pruned by bound

We note first that $\overline{z} = \max_k \overline{z}^k = \max\{20, 26\} = 26$ and $\underline{z} = \max_k \underline{z}^k = \max\{18, 21\} = 21$.

Second, we observe that as the optimal value has value at least 21, and the upper bound $\overline{z}^1 = 20$, no optimal solution can lie in the set S_1. Therefore the branch S_1 of the enumeration tree can be pruned *by bound.* ∎

Example 7.3 In Figure 7.5 we again decompose S into two sets S_1 and S_2 with different upper and lower bounds.

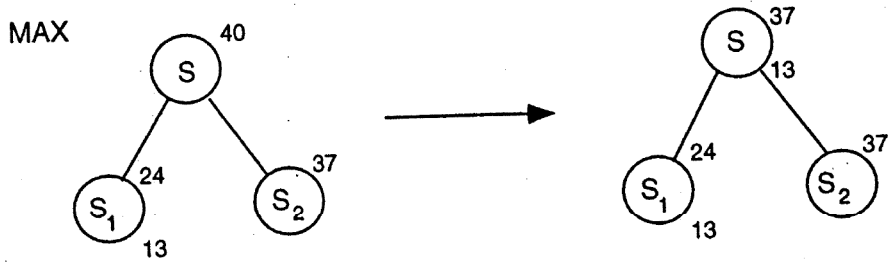

Fig. 7.5 No pruning possible

We note first that $\overline{z} = \max_k \overline{z}^k = \max\{24, 37\} = 37$ and $\underline{z} = \max_k \underline{z}^k = \max\{13, -\} = 13$. Here no other conclusion can be drawn and we need to explore both sets S_1 and S_2 further. ∎

Based on these examples, we can list at least three reasons that allow us to prune the tree and thus enumerate a large number of solutions implicitly.

(i) Pruning by optimality: $z^t = \{\max cx : x \in S_t\}$ has been solved.

(ii) Pruning by bound: $\overline{z}^t \leq \underline{z}$.

(iii) Pruning by infeasiblity: $S_t = \phi$.

If we now ask how the bounds are to be obtained, the reply is no different from in Chapter 2. The primal (lower) bounds are provided by feasible solutions and the dual (upper) bounds by relaxation or duality.

Building an implicit enumeration algorithm based on the above ideas is now in principle a fairly straightforward task. There are, however, many questions that must be addressed before such an algorithm is well-defined. Some of the most important questions are:

What relaxation or dual problem should be used to provide upper bounds? How should one choose between a fairly weak bound that can be calculated very rapidly and a stronger bound whose calculation takes a considerable time?

How should the feasible region be separated into smaller regions $S = S_1 \cup \ldots \cup S_K$? Should one separate into two or more parts? Should one use a fixed a priori rule for dividing up the set, or should the divisions evolve as a function of the bounds and solutions obtained en route?

In what order should the subproblems be examined? Typically there is a list of active problems that have not yet been pruned. Should the next one be chosen on a the basis of last-in first-out, of best/largest upper bound first, or of some totally different criterion?

These and other questions will be discussed further once we have seen an example.

7.3 BRANCH AND BOUND: AN EXAMPLE

The most common way to solve integer programs is to use implicit enumeration, or *branch and bound*, in which linear programming relaxations provide the bounds. We first demonstrate the approach by an example:

$$z = \max 4x_1 - x_2, \tag{7.1}$$
$$7x_1 - 2x_2 \leq 14 \tag{7.2}$$
$$x_2 \leq 3 \tag{7.3}$$
$$2x_1 - 2x_2 \leq 3 \tag{7.4}$$
$$x \in Z_+^2. \tag{7.5}$$

Bounding. To obtain a first upper bound, we add slack variables x_3, x_4, x_5 and solve the linear programming relaxation in which the integrality constraints are dropped. The resulting optimal basis representation is:

$$\begin{array}{llllll} \overline{z} = \max \frac{59}{7} & & -\frac{4}{7}x_3 & -\frac{1}{7}x_4 & & \\ x_1 & & +\frac{1}{7}x_3 & +\frac{2}{7}x_4 & & = \frac{20}{7} \\ & x_2 & & +x_4 & & = 3 \\ & & -\frac{2}{7}x_3 & +\frac{10}{7}x_4 & +x_5 & = \frac{23}{7} \\ x_1, & x_2, & x_3, & x_4, & x_5 & \geq 0. \end{array}$$

Thus we obtain an upper bound $\overline{z} = \frac{59}{7}$, and a nonintegral solution $(\overline{x}_1, \overline{x}_2) = (\frac{20}{7}, 3)$. Is there any straightforward way to find a feasible solution? Apparently not. By convention, as no feasible solution is yet available, we take as lower bound $\underline{z} = -\infty$.

Branching. Now because $\underline{z} < \overline{z}$, we need to divide or branch. How should we split up the feasible region? One simple idea is to choose an integer variable that is basic and fractional in the linear programming solution, and split the problem into two about this fractional value. If $x_j = \overline{x}_j \notin Z^1$, one can take:

$$S_1 = S \cap \{x : x_j \leq \lfloor \overline{x}_j \rfloor\}$$
$$S_2 = S \cap \{x : x_j \geq \lceil \overline{x}_j \rceil\}.$$

It is clear that $S = S_1 \cup S_2$ and $S_1 \cap S_2 = \phi$. Another reason for this choice is that the solution $\bar{x}$ of $LP(S)$ is not feasible in either $LP(S_1)$ or $LP(S_2)$. This implies that if there is no degeneracy (i.e., multiple optimal LP solutions), then $\max\{\overline{z}_1, \overline{z}_2\} < \overline{z}$, so the upper bound will strictly decrease.

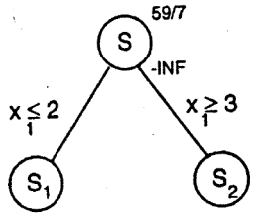

Fig. 7.6 Partial branch-and-bound tree 1

Following this idea, as $\bar{x}_1 = 20/7 \notin Z^1$, we take $S_1 = S \cap \{x : x_1 \leq 2\}$ and $S_2 = S \cap \{x : x_1 \geq 3\}$. We now have the tree shown in Figure 7.6. The subproblems (nodes) that must still be examined are called *active*.

Choosing a Node. The list of active problems (nodes) to be examined now contains S_1, S_2. We arbitrarily choose S_1.

Reoptimizing. How should we solve the new modified linear programs $LP(S_i)$ for $i = 1, 2$ without starting again from scratch?

As we have just added one single upper or lower bound constraint to the linear program, our previous optimal basis remains dual feasible, and it is therefore natural to reoptimize from this basis using the dual simplex algorithm. Typically, only a few pivots will be needed to find the new optimal linear programming solution.

Applying this to the linear program $LP(S_1)$, we can write the new constraint $x_1 \leq 2$ as $x_1 + s = 2, s \geq 0$, which can be rewritten in terms of the nonbasic variables as

$$-\tfrac{1}{7}x_3 - \tfrac{2}{7}x_4 + s = -\tfrac{6}{7}.$$

Thus we have the dual feasible representation:

$$\begin{array}{lccccccc}
\overline{z}_1 = \max \frac{59}{7} & & -\frac{4}{7}x_3 & -\frac{1}{7}x_4 & & & & \\
x_1 & & +\frac{1}{7}x_3 & +\frac{2}{7}x_4 & & & = & \frac{20}{7} \\
& x_2 & & +x_4 & & & = & 3 \\
& & -\frac{2}{7}x_3 & +\frac{10}{7}x_4 & +x_5 & & = & \frac{23}{7} \\
& & -\frac{1}{7}x_3 & -\frac{2}{7}x_4 & & +s & = & -\frac{6}{7} \\
x_1, & x_2, & x_3, & x_4, & x_5, & s & \geq & 0.
\end{array}$$

After two simplex pivots, the linear program is reoptimized, giving:

$$\begin{array}{lccccccc}
\overline{z}_1 = \max \frac{15}{2} & & & & -\frac{1}{2}x_5 & -3s & & \\
x_1 & & & & & +s & = & 2 \\
& x_2 & & & -\frac{1}{2}x_5 & +s & = & \frac{1}{2} \\
& & x_3 & & -x_5 & -5s & = & 1 \\
& & & x_4 & +\frac{1}{2}x_5 & +6s & = & \frac{5}{2} \\
x_1, & x_2, & x_3, & x_4, & x_5, & s & \geq & 0
\end{array}$$

with $\overline{z}_1 = \frac{15}{2}$, and $(\overline{x}_1^1, \overline{x}_2^1) = (2, \frac{1}{2})$.

Branching. S_1 cannot be pruned, so using the same branching rule as before, we create two new nodes $S_{11} = S_1 \cap \{x : x_2 \leq 0\}$ and $S_{12} = S_1 \cap \{x : x_2 \geq 1\}$, and add them to the node list. The tree is now as shown in Figure 7.7.

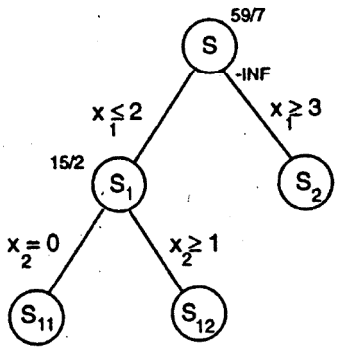

Fig. 7.7 Partial branch-and-bound tree 2

Choosing a Node. The active node list now contains S_2, S_{11}, S_{12}. Arbitrarily choosing S_2, we remove it from the node list and examine it in more detail.

Reoptimizing. To solve $LP(S_2)$, we use the dual simplex algorithm in the same way as above. The constraint $x_1 \geq 3$ is first written as $x_1 - t = 3, t \geq 0$, which expressed in terms of the nonbasic variables becomes:

$$\tfrac{1}{7}x_3 + \tfrac{2}{7}x_4 + t = -\tfrac{1}{7}.$$

From inspection of this constraint, we see that the resulting linear program

$$\begin{array}{lccccccc}
\overline{z}_2 = \max \frac{59}{7} & & -\frac{4}{7}x_3 & -\frac{1}{7}x_4 & & & & \\
x_1 & & +\frac{1}{7}x_3 & +\frac{2}{7}x_4 & & & = & \frac{20}{7} \\
& x_2 & & +x_4 & & & = & 3 \\
& & -\frac{2}{7}x_3 & +\frac{10}{7}x_4 & +x_5 & & = & \frac{23}{7} \\
& & \frac{1}{7}x_3 & +\frac{2}{7}x_4 & & +t & = & -\frac{1}{7} \\
x_1, & x_2, & x_3, & x_4, & x_5, & t & \geq & 0
\end{array}$$

is infeasible, $\overline{z}_2 = -\infty$, and hence node S_2 is *pruned by infeasibility.*

Choosing a Node. The node list now contains S_{11}, S_{12}. Arbitrarily choosing S_{12}, we remove it from the list.

Reoptimizing. $S_{12} = S \cap \{x : x_1 \leq 2, x_2 \geq 1\}$. The resulting linear program has optimal solution $\overline{x}^{12} = (2, 1)$ with value 7. As $\overline{x}^{12}$ is integer, $z^{12} = 7$.

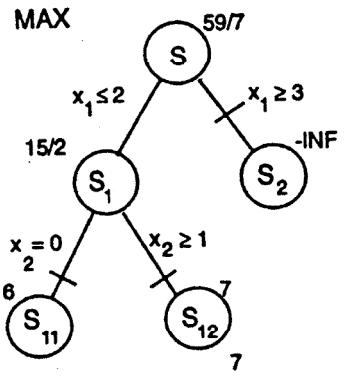

Fig. 7.8 Complete branch and bound tree

Updating the Incumbent. As the solution of $LP(S_{12})$ is integer, we update the value of the best feasible solution found $\underline{z} \leftarrow \max\{\underline{z}, 7\}$, and store the corresponding solution $(2,1)$. S_{12} is now *pruned by optimality.*

Choosing a Node. The node list now contains only S_{11}.

Reoptimizing. $S_{11} = S \cap \{x : x_1 \leq 2, x_2 \leq 0\}$. The resulting linear program has optimal solution $\overline{x}^{11} = (\frac{3}{2}, 0)$ with value 6. As $\underline{z} = 7 > \overline{z}_{11} = 6$, the node is *pruned by bound.*

Choosing a Node. As the node list is empty, the algorithm terminates. The incumbent solution $x = (2,1)$ with value $z = 7$ is optimal.

The complete branch-and-bound tree is shown in Figure 7.8. In Figure 7.9 we show graphically the feasible node sets S_i, the branching, the relaxations $LP(S_i)$, and the solutions encountered in the example.

7.4 LP-BASED BRANCH AND BOUND

In Figure 7.10 we present a flowchart of a simple branch and bound algorithm, and then discuss in more detail some of the practical aspects of developing and using such an algorithm.

Storing the Tree. In practice one does not store a tree, but just the list of *active* nodes or subproblems that have not been pruned and that still need to be explored further. Here the question arises of how much information one should keep. Should one keep a minimum of information and be prepared to repeat certain calculations, or should one keep all the information available? At a minimum, the best known dual bound and the variable lower and upper

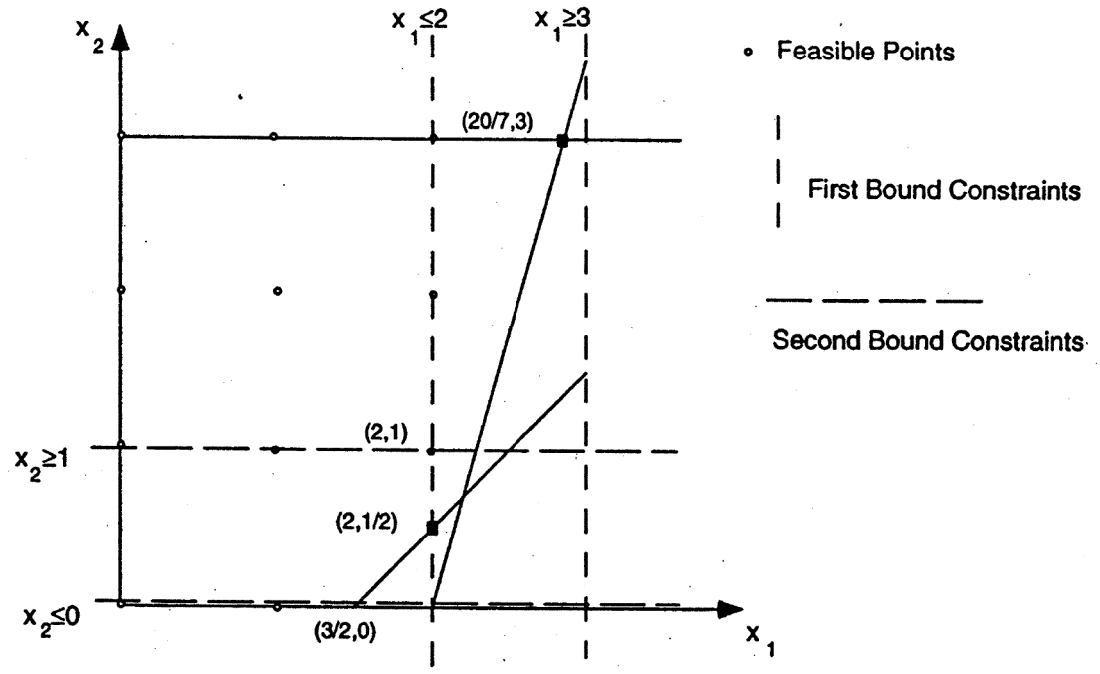

Fig. 7.9 Division of the feasible region

bounds needed to restore the subproblem are stored. Usually one also keeps an optimal or near-optimal basis, so that the linear programming relaxation can be reoptimized rapidly.

Returning to the questions raised earlier, there is no single answer that is best for all instances. One needs to use rules based on a combination of theory, common sense, and practical experimentation. In our example, the question of **how to bound** was solved by using an LP relaxation; **how to branch** was solved by choosing an integer variable that is fractional in the LP solution. However, as there is typically a choice of a set C of several candidates, we need a rule to choose between them. One common choice is *the most fractional variable*:

$$\arg \max_{j \in C} \min[f_j, 1 - f_j]$$

where $f_j = x_j^* - \lfloor x_j^* \rfloor$, so that a variable with fractional value $f_j = \frac{1}{2}$ is best. Other rules are based on the idea of *estimating* the cost of forcing the variable x_j to become integer.

How to choose a node was avoided by making an arbitrary choice. In practice there are several contradictory arguments that can be invoked:

(i) It is only possible to prune the tree significantly with a (primal) feasible solution, giving a hopefully good lower bound. Therefore one should descend as quickly as possible in the enumeration tree to find a first feasible solution. This suggests the use of a *Depth-First Search* strategy. Another argument for

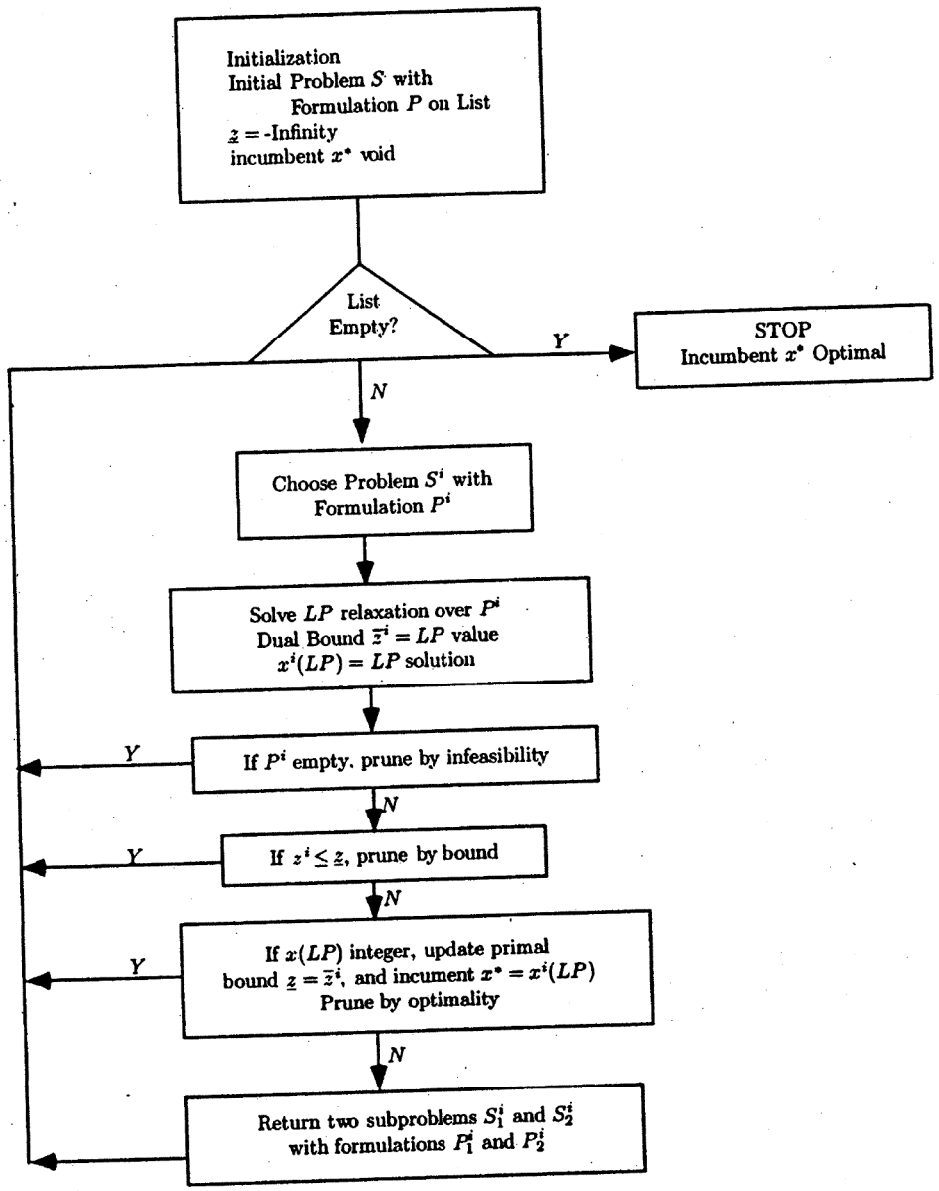

Fig. 7.10 Branch-and-bound flow chart

such a strategy is the observation that it is always easy to resolve the linear programming relaxation when a simple constraint is added, and the optimal basis is available. Therefore passing from a node to one of its immediate descendants is to be encouraged. In the example this would imply that after treating node S_1, the next node treated would be S_{11} or S_{12} rather than S_2.

(ii) To minimize the total number of nodes evaluated in the tree, the optimal strategy is to always choose the active node with the best (largest upper) bound (i.e., choose node s where $\overline{z}_s = \max_t \overline{z}_t$). With such a rule, one will never divide any node whose upper bound $\overline{z}_t$ is less than the optimal value z. This leads to a *Best-Node First* strategy. In the example of the previous section, this would imply that after treating node S_1, the next node chosen

would be S_2 with bound $\frac{59}{7}$ from its predecessor, rather than S_{11} or S_{12} with bound $\frac{15}{2}$.

In practice a compromise between these ideas is often adopted, involving an initial depth-first strategy until at least one feasible solution has been found, followed by a strategy mixing best node and depth first so as to try to prove optimality and also find better feasible solutions.

7.5 USING A BRANCH-AND-BOUND SYSTEM

Commercial branch-and-bound systems for integer and mixed integer programming are essentially as described in the previous section, and the default strategies have been chosen by tuning over hundreds of different problem instances. The basic philosophy is to solve and resolve the linear programming relaxations as rapidly as possible, and if possible to branch intelligently. Given this philosophy, all recent systems contain, or offer,

1. A powerful (automatic) preprocessor, which simplifies the model by reducing the number of constraints and variables, so that the linear programs are easier
2. The simplex algorithm with a choice of pivoting strategies, and an interior point option for solving the linear programs
3. Limited choice of branching and node selection options
4. Use of priorities

and some offer

5. GUB/SOS branching
6. Strong branching
7. Reduced cost fixing
8. Primal heuristics

In this section we briefly discuss those topics requiring user intervention. Preprocessing, which is very important, but automatic, is presented in the (optional) next section. Reduced cost fixing is treated in Exercise 7.7, and primal heuristics are discussed in Chapter 12.

Priorities. Priorities allow the user to tell the system the relative importance of the integer variables. The user provides a file specifying a value (importance) of each integer variable. When it has to decide on a branching variable, the system will choose the highest priority integer variable whose current linear programming value is fractional. At the same time the user can specify a preferred branching direction telling the system which of the two branches to

explore first.

GUB Branching. Many models contain generalized upper bound (GUB) or special ordered sets (SOS) of the form

$$\sum_{j=1}^{k} x_j = 1$$

with $x_j \in \{0,1\}$ for $j = 1,\ldots,k$. If the linear programming solution x^* has some of the variables $x_1^*,\ldots,x_k^*$ fractional, then the standard branching rule is to impose $S_1 = S \cap \{x : x_j = 0\}$ and $S_2 = S \cap \{x : x_j = 1\}$ for some $j \in \{1,\ldots,k\}$. However, because of the GUB constraint, $\{x : x_j = 0\}$ leaves $k-1$ possibilities $\{x : x_i = 1\}_{i \neq j}$ whereas $\{x : x_j = 1\}$ leaves only one possibility. So S_1 is typically a much larger set than S_2, and the tree is *unbalanced.*

GUB branching is designed to provide a more balanced division of S into S_1 and S_2. Specifically the user specifies an ordering of the variables in the GUB set $j_1,\ldots,j_k$, and the branching scheme is then to set

$$S_1 = S \cap \{x : x_{j_i} = 0 \; i = 1,\ldots r\} \text{ and}$$
$$S_2 = S \cap \{x : x_{j_i} = 0 \; i = r+1,\ldots k\},$$

where $r = \min\{t : \sum_{i=1}^{t} x_{j_i}^* \geq \frac{1}{2}\}$. In many cases such a branching scheme is much more effective than the standard scheme, and the number of nodes in the tree is significantly reduced.

User Options (a) **Cutoffs.** If the user knows or can construct a good feasible solution to his or her problem, it is very important that its value is passed to the system as the incumbent value to serve as a *cutoff* in the branch and bound.

(b) **Simplex Strategies.** Though the linear programming algorithms are finely tuned, the default strategy will not be best for all classes of problems. Different *simplex pricing* strategies may make a huge difference in running times for a given class of models, so if similar models are resolved repeatedly or the linear programs seem very slow, some experimentation by the user with pricing strategies is permitted. In addition, on very large models, *interior point methods* may be best for the solution of the first linear program. Unfortunately, up to now such methods are still not good for reoptimizing quickly at each node of the tree.

(c) **Strong Branching.** The idea behind strong branching is that on difficult problems it should be worthwhile to do more work to try to choose a better branching variable. The system chooses a set C of basic integer variables that are fractional in the LP solution, branches up and down on each of them in

turn, and reoptimizes on each branch either to optimality, or for a specified number of dual simplex pivots. Now for each variable $j \in C$, it has upper bounds z_j^D for the down branch and z_j^U for the up branch. The variable having the largest effect (decrease of the dual bound)

$$j^* = \arg\min_{j \in C} \max[z_j^D, z_j^U]$$

is then chosen, and branching really takes place on this variable. Obviously, solving two LPs for each variable in C is costly, so such branching should only be used when the other criteria have been found to be ineffective.

7.5.1 If All Else Fails

What can one do if a particular problem instance turns out to be difficult, meaning that after a certain time

(i) no feasible solution has been found, or

(ii) the gap between the value of the best feasible solution and the value of the dual upper bound is unsatisfactorily large, or

(iii) the system runs out of space because there are too many active nodes in the node list?

Finding Feasible Solutions. This is in general $\mathcal{NP}$-hard. Some systems have simple primal heuristics embedded in them. Also as discussed earlier, using priorities and directions for branching can help. How to find feasible solutions, starting from the LP solution or using explicit problem structure, is the topic of Chapter 12.

Finding Better Dual Bounds. Branch-and-bound algorithms fail very often because the bounds obtained from the linear programming relaxations are too weak. This means that tightening up the formulation of the problem is of crucial importance. Systematic ways to do this are the subject of Chapters 8–11. Specifically the addition of constraints or cuts to improve the formulation is treated in Chapers 8 and 9, leading to the development of a potentially more powerful branch-and-cut algorithm. The Lagrangian relaxation and column generation approaches of Chapters 10 and 11 provide alternative ways to strengthen the formulation by convexifying part of the feasible region.

7.6 PREPROCESSING*

Before solving a linear or integer program, it is natural to check that the formulation is "sensible", and as strong as possible given the information available.

All the commercial branch-and-bound systems carry out such a check, called *preprocessing*. The basic idea is to try to quickly detect and eliminate redundant constraints and variables, and tighten bounds where possible. Then if the resulting linear/integer program is smaller/tighter, it will typically be solved much more quickly. This is especially important in the case of branch-and-bound because tens or hundreds of thousands of linear programs may need to be solved.

First we demonstrate linear programming preprocessing by an example.

Example 7.4 Consider the linear program

$$\begin{array}{lrcrcrcl} \max & 2x_1 & + & x_2 & - & x_3 & & \\ & 5x_1 & - & 2x_2 & + & 8x_3 & \leq & 15 \\ & 8x_1 & + & 3x_2 & - & x_3 & \geq & 9 \\ & x_1 & + & x_2 & + & x_3 & \leq & 6 \\ & 0 & \leq & x_1 & \leq & 3 & & \\ & 0 & \leq & x_2 & \leq & 1 & & \\ & 1 & \leq & x_3. & & & & \end{array}$$

Tightening Bounds. Isolating variable x_1 in the first constraint we obtain

$$5x_1 \leq 15 + 2x_2 - 8x_3 \leq 15 + 2 \times 1 - 8 \times 1 = 9$$

where we use the bound inequalities $x_2 \leq 1$ and $-x_3 \leq -1$. Thus we obtain the tightened bound $x_1 \leq \frac{9}{5}$.

Similarly isolating variable x_3, we obtain

$$8x_3 \leq 15 + 2x_2 - 5x_1 \leq 15 + 2 \times 1 - 5 \times 0 = 17,$$

and the tightened bound $x_3 \leq \frac{17}{8}$.

Isolating variable x_2, we obtain

$$2x_2 \geq 5x_1 + 8x_3 - 15 \geq 5 \times 0 + 8 \times 1 - 15 = -7.$$

Here the existing bound $x_2 \geq 0$ is not changed.

Turning to the second constraint, isolating x_1 and using the same approach, we obtain $8x_1 \geq 9 - 3x_2 + x_3 \geq 9 - 3 + 1 = 7$, and an improved lower bound $x_1 \geq \frac{7}{8}$.

No more bounds are changed based on the second or third constraints. However, as certain bounds have been tightened, it is worth passing through the constraints again.

Constraint 1 for x_3 now gives $8x_3 \leq 15 + 2x_2 - 5x_1 \leq 15 + 2 - 5 \times \frac{7}{8} = \frac{101}{8}$. Thus we have the new bound $x_3 \leq \frac{101}{64}$.

Redundant Constraints. Using the latest upper bounds in constraint 3, we see that

$$x_1 + x_2 + x_3 \leq \frac{9}{5} + 1 + \frac{101}{64} < 6,$$

and so this constraint is redundant and can be discarded. The problem is now reduced to

$$\begin{array}{lllll} \max & 2x_1 & +x_2 & -x_3 & \\ & 5x_1 & -2x_2 & +8x_3 & \leq 15 \\ & 8x_1 & +3x_2 & -x_3 & \geq 9 \\ & \frac{7}{8} \leq x_1 & \leq \frac{9}{5}, & 0 \leq x_2 \leq 1, & 1 \leq x_3 \leq \frac{101}{64}. \end{array}$$

Variable Fixing (by Duality). Considering variable x_2, observe that increasing its value makes all the constraints (other than its bound constraints) less tight. As the variable has a positive objective coefficent, it is advantageous to make the variable as large as possible, and thus set it to its upper bound of 1. (Another way to arrive at a similar conclusion is to write out the LP dual. For the dual constraint corresponding to the primal variable x_2 to be feasible, the dual variable associated with the constraint $x_2 \leq 1$ must be positive. This implies by complemementary slackness that $x_2 = 1$ in any optimal solution.)

Similarly, decreasing x_3 makes the constraints less tight. As the variable has a negative objective coefficient, it is best to make it as small as possible, and thus set it to its lower bound $x_3 = 1$. Finally the LP is reduced to the trivial problem

$$\max\{2x_1 : \tfrac{7}{8} \leq x_1 \leq \tfrac{9}{5}\}.$$ ∎

Formalizing the above ideas is straightforward.

Proposition 7.3 *Consider the set* $S = \{x : a_0x_0 + \sum_{j=1}^{n} a_jx_j \leq b, l_j \leq x_j \leq u_j \text{ for } j = 0, 1, \ldots, n\}$.

(i) Bounds on Variables. If $a_0 > 0$, *then*

$$x_0 \leq (b - \sum_{j:a_j>0} a_jl_j - \sum_{j:a_j<0} a_ju_j)/a_0,$$

and if $a_0 < 0$, *then*

$$x_0 \geq (b - \sum_{j:a_j>0} a_jl_j - \sum_{j:a_j<0} a_ju_j)/a_0.$$

(ii) Redundancy. The constraint $a_0x_0 + \sum_{j=1}^{n} a_jx_j \leq b$ *is redundant if*

$$\sum_{j:a_j>0} a_ju_j + \sum_{j:a_j<0} a_jl_j \leq b.$$

(iii) Infeasibility. $S = \emptyset$ *if*

$$\sum_{j:a_j>0} a_jl_j + \sum_{j:a_j<0} a_ju_j > b.$$

(iv) Variable Fixing. For a maximization problem in the form: $\max\{cx : Ax \leq b, l \leq x \leq u\}$, *if* $a_{ij} \geq 0$ *for all* $i = 1, \ldots, m$ *and* $c_j < 0$, *then* $x_j = l_j$. *Conversely if* $a_{ij} \leq 0$ *for all* $i = 1, \ldots, m$ *and* $c_j > 0$, *then* $x_j = u_j$.

Turning now to integer programming problems, preprocessing can sometimes be taken a step further. Obviously, if $x_j \in Z^1$ and the bounds l_j or u_j are not integer, we can tighten to

$$\lceil l_j \rceil \leq x_j \leq \lfloor u_j \rfloor.$$

For mixed integer programs with variable upper and lower bound constraints $l_j y_j \leq x_j \leq u_j y_j$ with $y_j \in \{0,1\}$, it is also important to use the tightest bound information.

For *BIPs* it is common to look for simple "logical" or "boolean" constraints involving only one or two variables, and then either add them to the problem or use them to fix some variables. Again we demonstrate by example.

Example 7.5 Consider the set of constraints involving four 0–1 variables:

$$\begin{array}{rrrrcl} 7x_1 & +3x_2 & -4x_3 & -2x_4 & \leq & 1 \\ -2x_1 & +7x_2 & +3x_3 & +x_4 & \leq & 6 \\ & -2x_2 & -3x_3 & -6x_4 & \leq & -5 \\ 3x_1 & & -2x_3 & & \geq & -1 \\ & & x & \in & B^4. & \end{array}$$

Generating Logical Inequalities. Examining row 1, we see that if $x_1 = 1$, then necessarily $x_3 = 1$, and similarly $x_1 = 1$ implies $x_4 = 1$. This can be formulated with the linear inequalities $x_1 \leq x_3$ and $x_1 \leq x_4$. We see also that the constraint is infeasible if both $x_1 = x_2 = 1$ leading to the constraint $x_1 + x_2 \leq 1$.

Row 2 gives the inequalities $x_2 \leq x_1$ and $x_2 + x_3 \leq 1$.

Row 3 gives $x_2 + x_4 \geq 1$ and $x_3 + x_4 \geq 1$.

Row 4 gives $x_1 \geq x_3$.

Combining Pairs of Logical Inequalities. We consider pairs involving the same variables.

From rows 1 and 4, we have $x_1 \leq x_3$ and $x_1 \geq x_3$, which together give $x_1 = x_3$.

From rows 1 and 2, we have $x_1 + x_2 \leq 1$ and $x_2 \leq x_1$ which together give $x_2 = 0$. Now from $x_2 + x_4 \geq 1$ and $x_2 = 0$, we obtain $x_4 = 1$.

Simplifying. Making the substitutions $x_2 = 0, x_3 = x_1, x_4 = 1$, all four constraints of the feasible region are redundant, and we are left with $x_1 \in \{0,1\}$, so the only feasible solutions are $(1,0,1,1)$ and $(0,0,0,1)$. ∎

In Exercise 7.10, the reader is asked to formalize the approach taken in this example. The logical inequalities can also be viewed as providing a foretaste of the valid inequalities to be developed in the next chapter.

7.7 NOTES

7.2 The first paper presenting a branch-and-bound algorithm for integer programming is [LanDoi60]. [Litetal63] presents a computationally successful application to the *TSP* problem using an assignment relaxation. [Balas65] developed an algorithm for 0–1 problems using simple tests to obtain dual bounds and check primal feasibility.

7.4 Almost all commercial codes since the 1960s have been linear programming based branch-and-bound codes. The two–way branching scheme commonly used is from [Dak65].

7.5 A discussion of important elements of commercial codes can be found in [Beal79]. GUB/SOS branching is from [BealTom70], probing from [GuiSpi81], and strong branching from [Appetal95]. One important new idea is constraint branching, used for *TSP* problems in [CloNad93], and by [CooRutetal93] in their implementation of basis reduction for integer programming based upon the fundamental paper [Len83]. Recent experiments with various branch-and-bound strategies are reported in [LinSav97].

As solving linear programs forms such an important part of an integer programming algorithm, improvements in solving linear programs are crucial. All recent commercial codes include an interior point algorithm, as for many large linear programs, the latter algorithm is faster than the simplex method. However, because reoptimization with the simplex method is easier than with interior point codes, the simplex method is still used in branch-and-bound. Improving reoptimization with interior point codes is a major challenge for the next few years. See [RooTerVia97] and [Wri97] for recent texts on interior point algorithms. Work on solving integer programs with interior point algorithms is a wide open area [MitTod92],[Mit96].

Knapsack problems, in which the linear programming relaxations can be solved by inspection, have always been treated by specialized codes; see the book [MarTot90].

7.6 Preprocessing is crucially important for the rapid solution of linear programs. Its importance for integer programs is recognized in [BreMitWil73], and discussed more recently in [HofPad91],[Sav94].

7.8 EXERCISES

1. Consider the enumeration tree (minimization problem) in Figure 7.11:

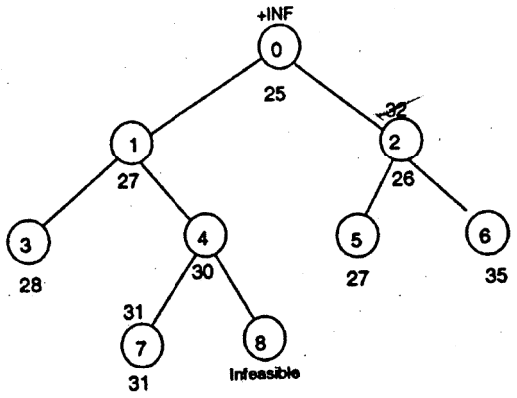

Fig. 7.11 Enumeration tree (min)

(i) Give tightest possible lower and upper bounds on the optimal value z.
(ii) Which nodes can be pruned and which must be explored further?

2. Consider the two-variable integer program:

$$\begin{array}{rrcrcl} \max & 9x_1 & + & 5x_2 & & \\ & 4x_1 & + & 9x_2 & \leq & 35 \\ & x_1 & & & \leq & 6 \\ & x_1 & - & 3x_2 & \geq & 1 \\ & 3x_1 & + & 2x_2 & \leq & 19 \\ & & x & \in & Z^2_+. & \end{array}$$

Solve by branch-and-bound graphically and algebraically.

3. Consider the 0–1 knapsack problem:

$$\max\{\sum_{j=1}^{n} c_j x_j : \sum_{j=1}^{n} a_j x_j \leq b, x \in B^n\}$$

with $a_j, c_j > 0$ for $j = 1, \ldots, n$.

(i) Show that if $\frac{c_1}{a_1} \geq \ldots \geq \frac{c_n}{a_n} > 0$, $\sum_{j=1}^{r-1} a_j \leq b$ and $\sum_{j=1}^{r} a_j > b$, the solution of the LP relaxation is $x_j = 1$ for $j = 1, \ldots, r-1$, $x_r = (b - \sum_{j=1}^{r-1} a_j)/a_r$, and $x_j = 0$ for $j > r$.
(ii) Solve the instance

$$\begin{gathered} \max 17x_1 + 10x_2 + 25x_3 + 17x_4 \\ 5x_1 + 3x_2 + 8x_3 + 7x_4 \leq 12 \\ x \in B^4 \end{gathered}$$

by branch-and-bound.

4. Solve the integer knapsack problem:

$$\begin{aligned} &\max 10x_1 + 12x_2 + 7x_3 + \tfrac{3}{2}x_4 \\ &4x_1 + 5x_2 + 3x_3 + 1x_4 \leq 10 \\ &x_1, x_2 \in Z_+^1, x_3, x_4 \in \{0,1\} \end{aligned}$$

by branch-and-bound.

5. (i) Solve the $STSP$ instance with $n = 5$ and distance matrix

$$(c_e) = \begin{pmatrix} - & 10 & 2 & 4 & 6 \\ - & - & 9 & 3 & 1 \\ - & - & - & 5 & 6 \\ - & - & - & - & 2 \end{pmatrix}$$

by branch-and-bound using a 1-tree relaxation (see Definition 2.3) to obtain bounds.
(ii) Solve the TSP instance with $n = 4$ and distance matrix

$$(c_{ij}) = \begin{pmatrix} - & 7 & 6 & 3 \\ 3 & - & 6 & 9 \\ 2 & 3 & - & 1 \\ 7 & 9 & 4 & \end{pmatrix}$$

by branch-and-bound using an assignment relaxation to obtain bounds.
(iii) Describe clearly the branching rules you use in (i) and (ii), and motivate your choice.

6. Using a branch-and-bound system, solve your favorite integer program with different choices of branching and node selection rules, and report on the differences in the running time and the number of nodes in the branch-and-bound tree.

7. *Reduced cost fixing.* Suppose that the linear programming relaxation of an integer program has been solved to optimality, and the objective function is then represented in the form

$$z = \max cx, cx = \bar{a}_{00} + \sum_{j \in NB_1} \bar{a}_{0j} x_j + \sum_{j \in NB_2} \bar{a}_{0j}(x_j - u_j)$$

where NB_1 are the nonbasic variables at zero, and NB_2 are the nonbasic variables at their upper bounds u_j, $\bar{a}_{0j} \leq 0$ for $j \in NB_1$, and $\bar{a}_{0j} \geq 0$ for $j \in NB_2$. In addition suppose that a primal feasible solution of value $\underline{z}$ is known. Prove the following: In any optimal solution,

$$x_j \leq \lfloor \tfrac{\bar{a}_{00}-z}{-\bar{a}_{0j}} \rfloor \text{ for } j \in N_1, \text{ and}$$
$$x_j \geq u_j - \lceil \tfrac{\bar{a}_{00}-z}{\bar{a}_{0j}} \rceil \text{ for } j \in N_2.$$

8. Consider a fixed charge network problem:

$$\min\{cx + fy : Nx = b, x \leq uy, x \in R^n_+, y \in Z^n_+\}$$

where N is the node-arc incidence matrix of the network, and b the demand vector. In using priorities, suggest a preferred direction for the variables.

9. Consider the 0–1 problem:

$$\begin{array}{lrrrrrrrrrrr}
\max & 5x_1 & - & 7x_2 & - & 10x_3 & + & 3x_4 & - & 5x_5 & & \\
 & x_1 & + & 3x_2 & - & 5x_3 & + & x_4 & + & 4x_5 & \leq & 0 \\
 & -2x_1 & - & 6x_2 & + & 3x_3 & - & 2x_4 & - & 2x_5 & \leq & -4 \\
 & & & 2x_2 & - & 2x_3 & - & x_4 & + & x_5 & \leq & -2 \\
 & & & & & & x & \in & B^5. & & &
\end{array}$$

Simplify using logical inequalties.

10. *Logical.* Given a set in 0–1 variables

$$X = \{x \in B^n : \sum_{j=1}^{n} a_j x_j \leq b\}$$

with $a_j \geq 0$ for $j = 1, \ldots, n$, under what conditions is

(i) the set X empty?
(ii) the constraint $\sum_{j=1}^{n} a_j x_j \leq b$ redundant?
(iii) the constraint $x_j = 0$ valid?
(iv) the constraint $x_i + x_j \leq 1$ valid?

Apply these rules to the first constraint in Exercise 9.

11. Prove Proposition 7.3 concerning preprocessing.

12. Let

$$X = \{x \in B^n : \sum_{j=1}^{n} a_j x_j \leq b\}$$

with $a_1 \geq a_2 \geq \ldots \geq a_n \geq 0$ and $b \geq 0$. The idea is to write each such set in some simple canonical form. For example, for $x \in B^3$, $12x_1 + 8x_2 + 3x_1 \leq 14$ is equivalent to $2x_1 + 1x_2 + 1x_3 \leq 2$.

(i) When $n = 2$, how many distinct knapsack sets are there? Write them out in a canonical form with integral coefficients and $1 = a_1 \geq a_2$.

(ii) Repeat for $n = 3$ with $a_1 \leq 2$.

13*. Some small integer programs are very difficult for mixed integer programming systems. Try to find a feasible solution to the integer equality knapsack: $\{x \in Z_+^n : \sum_{j=1}^n a_j x_j = b\}$ with $a = (12228, 36679, 36682, 48908, 61139, 73365)$ and $b = 89716837$.

14. Suppose that $P^i = \{x \in R^n : A^i x \leq b^i\}$ for $i = 1, \ldots, m$ and that $C_k \subseteq \{1, \ldots, m\}$ for $k = 1, \ldots, K$. A *disjunctive program* is a problem of the form

$$\max\{cx : x \in \cup_{i \in C_k} P^i \text{ for } k = 1, \ldots, K\}.$$

Show how the following can be modeled as disjunctive programs:

(i) a 0–1 integer program.
(ii) a linear complementarity problem: $w = q + Mz, w, z \in R_+^m, w_j z_j = 0$ for $j = 1, \ldots, m$.
(iii) a single machine sequencing problem with job processing times p_j, and variables t_j representing the start time of job j for $j = 1, \ldots, n$.
(iv) the nonlinear expression $z = \max\{3x_1 + 2x_2 - 3, 9x_1 - 4x_2 + 6\}$.
(v) the constraint: if $x \in R_+^1$ is positive, then x lies between 20 and 50, and is a multiple of 5.

15. Devise a branch-and-bound algorithm for a disjunctive program.