Chapter 3
Integer Programming

Robert Bosch and Michael Trick

3.1 Introduction

Over the last 25 years, the combination of faster computers, more reliable data and improved algorithms has resulted in the near-routine solution of many integer programs of practical interest. Integer programming models are used in a wide variety of applications, including scheduling, resource assignment, planning, supply chain design, auction design, and many, many others. In this tutorial, we outline some of the major themes involved in creating and solving integer programming models.

The foundation of much of analytical decision making is linear programming. In a linear program, there are *variables*, *constraints*, and an *objective function*. The variables, or decisions, take on numerical values. Constraints are used to limit the values to a feasible region. These constraints must be linear in the decision variables. The objective function then defines which particular assignment of feasible values to the variables is optimal: it is the one that maximizes (or minimizes, depending on the type of the objective) the objective function. The objective function must also be linear in the variables.

Linear programs can model many problems of practical interest, and modern linear programming optimization codes can find optimal solutions to problems with hundreds of thousands of constraints and variables. It is this combination of modeling strength and solvability that makes linear programming so important.

Integer programming adds additional constraints to linear programming. An integer program begins with a linear program, and adds the requirement that some or all of the variables take on integer values. This seemingly innocuous change greatly

R. Bosch
Oberlin College, Oberlin, OH, USA

M. Trick (✉)
Carnegie Mellon University, Pittsburgh, PA, USA
e-mail: trick@cmu.edu

E.K. Burke and G. Kendall (eds.), *Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques*, DOI 10.1007/978-1-4614-6940-7_3,

increases the number of problems that can be modeled, but also makes the models more difficult to solve. In fact, one frustrating aspect of integer programming is that two seemingly similar formulations for the same problem can lead to radically different computational experience: one formulation may quickly lead to optimal solutions, while the other may take an excessively long time to solve.

There are many keys to successfully developing and solving integer programming models. We consider the following aspects:

- Be creative in formulations
- Find integer programming formulations with a strong relaxation
- Avoid symmetry
- Consider formulations with many constraints
- Consider formulations with many variables
- Modify branch-and-bound search parameters.

To fix ideas, we will introduce a particular integer programming model, and show how the main integer programming algorithm, branch-and-bound, operates on that model. We will then use this model to illustrate the key ideas to successful integer programming.

3.1.1 Facility Location

We consider a facility location problem. A chemical company owns four factories that manufacture a certain chemical in raw form. The company would like to get into the business of refining the chemical. It is interested in building refining facilities, and it has identified three possible sites. The table below contains variable costs, fixed costs, and weekly capacities for the three possible refining facility sites, and weekly production amounts for each factory. The variable costs are in dollars per week and include transportation costs. The fixed costs are in dollars per year. The production amounts and capacities are in tonnes per week.

	Site			
Var. cost	1	2	3	Prod.
Factory 1	25	20	15	1,000
Factory 2	15	25	20	1,000
Factory 3	20	15	25	500
Factory 4	25	15	15	500
Fixed cost	500,000	500,000	500,000	
Capacity	1,500	1,500	1,500	

The decision maker who faces this problem must answer two very different types of questions: questions that require numerical answers (e.g. how many tonnes of chemical should factory i send to the site-j refining facility each week?) and questions that require yes–no answers (e.g. should the site-j facility be constructed?). While we can easily model the first type of question by using continuous decision variables (by letting x_{ij} equal the number of tonnes of chemical sent from factory

i to site j each week), we *cannot* do this with the second. We need to use integer variables. If we let y_j equal 1 if the site-j refining facility is constructed and 0 if it isn't, we quickly arrive at an integer programming formulation of the problem:

$$
\begin{aligned}
\text{minimize} \quad & 52 \cdot 25x_{11} + 52 \cdot 20x_{12} + 52 \cdot 15x_{13} \\
+ \; & 52 \cdot 15x_{21} + 52 \cdot 25x_{22} + 52 \cdot 20x_{23} \\
+ \; & 52 \cdot 20x_{31} + 52 \cdot 15x_{32} + 52 \cdot 25x_{33} \\
+ \; & 52 \cdot 25x_{41} + 52 \cdot 15x_{42} + 52 \cdot 15x_{43} \\
+ \; & 500{,}000y_1 + 500{,}000y_2 + 500{,}000y_3 \\
\text{subject to} \quad & x_{11} + x_{12} + x_{13} = 1{,}000 \\
& x_{21} + x_{22} + x_{23} = 1{,}000 \\
& x_{31} + x_{32} + x_{33} = 500 \\
& x_{41} + x_{42} + x_{43} = 500 \\
& x_{11} + x_{21} + x_{31} + x_{41} \leq 1{,}500y_1 \\
& x_{12} + x_{22} + x_{32} + x_{42} \leq 1{,}500y_2 \\
& x_{13} + x_{23} + x_{33} + x_{43} \leq 1{,}500y_3 \\
& x_{ij} \geq 0 \quad \text{for all } i \text{ and } j \\
& y_j \in \{0,1\} \quad \text{for all } j.
\end{aligned}
$$

The objective is to minimize the yearly cost, the sum of the variable costs (which are measured in dollars per week) and the fixed costs (which are measured in dollars per year). The first set of constraints ensures that each factory's weekly chemical production is sent somewhere for refining. Since factory 1 produces 1,000 tonnes of chemical per week, factory 1 must ship a total of 1,000 tonnes of chemical to the various refining facilities each week. The second set of constraints guarantees two things: (1) if a facility it open, it will operate at or below its capacity, and (2) if a facility is not open, it will not operate at all. If the site-1 facility is open ($y_1 = 1$) then the factories can send it up to $1{,}500y_1 = 1{,}500 \times 1 = 1{,}500$ tonnes of chemical per week. If it is not open ($y_1 = 0$), then the factories can send it up to $1{,}500y_1 = 1{,}500 \times 0 = 0$ tonnes per week.

This introductory example demonstrates the need for integer variables. It also shows that with integer variables one can model simple logical requirements (if a facility is open, it can refine up to a certain amount of chemical; if not, it can't do any refining at all). It turns out that with integer variables one can model a whole host of logical requirements. One can also model fixed costs, sequencing and scheduling requirements, and many other problem aspects.

3.1.2 Solving the Facility Location IP

Given an integer program (IP), there is an associated linear program (LR) called the *linear relaxation*. It is formed by dropping (relaxing) the integrality restrictions. Since (LR) is less constrained than (IP), the following are immediate:

- If (IP) is a minimization problem, the optimal objective value of (LR) is less than or equal to the optimal objective value of (IP).

- If (IP) is a maximization problem, the optimal objective value of (LR) is greater than or equal to the optimal objective value of (IP).
- If (LR) is infeasible, then so is (IP).
- If all the variables in an optimal solution of (LR) are integer valued, then that solution is optimal for (IP) too.
- If the objective function coefficients are integer valued, then for minimization problems the optimal objective value of (IP) is greater than or equal to the ceiling of the optimal objective value of (LR). For maximization problems, the optimal objective value of (IP) is less than or equal to the floor of the optimal objective value of (LR).

In summary, solving (LR) can be quite useful: it provides a bound on the optimal value of (IP), and may (if we are lucky) give an optimal solution to (IP).

For the remainder of this section, we will let (IP) stand for the facility location integer program and (LR) for its linear programming relaxation. When we solve (LR), we get

Objective		
x_{11}	x_{12}	x_{13}
x_{21}	x_{22}	x_{23}
x_{31}	x_{32}	x_{33}
x_{41}	x_{42}	x_{43}
y_1	y_2	y_3

$=$

3,340,000		
·	·	1,000
1,000	·	·
·	500	·
·	500	·
$\frac{2}{3}$	$\frac{2}{3}$	$\frac{2}{3}$

This solution has factory 1 send all 1,000 tonnes of its chemical to site 3, factory 2 send all 1,000 tonnes of its chemical to site 1, factory 3 send all 500 tonnes to site 2, and factory 4 send all 500 tonnes to site 2. It constructs two-thirds of a refining facility at each site. Although it costs only 3,340,000 dollars per year, it cannot be implemented; all three of its integer variables take on fractional values.

It is tempting to try to produce a feasible solution by rounding. Here, if we round y_1, y_2 and y_3 from $2/3$ to 1, we get lucky (this is certainly not always the case!) and get an integer feasible solution. Although we can state that this is a good solution—its objective value of 3,840,000 is within 15% of the objective value of (LR) and hence within 15% of optimal—we can't be sure that it is optimal.

So how can we find an optimal solution to (IP)? Examining the optimal solution to (LR), we see that y_1, y_2 and y_3 are fractional. We want to force y_1, y_2 and y_3 to be integer valued. We start by *branching* on y_1, creating two new integer programming problems. In one, we add the constraint $y_1 = 0$. In the other, we will add the constraint $y_1 = 1$. Note that any optimal solution to (IP) must be feasible for one of the two subproblems.

After we solve the linear programming relaxations of the two subproblems, we can display what we know in a tree, as shown in Fig. 3.1.

Note that the optimal solution to the left subproblem's LP relaxation is integer valued. It is therefore an optimal solution to the left subproblem. Since there is no point in doing anything more with the left subproblem, we mark it with an "$\times$" and focus our attention on the right subproblem.

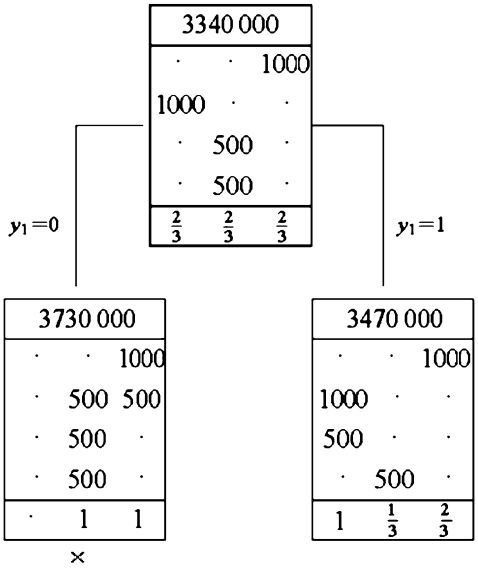

Fig. 3.1 Intermediate branch-and-bound tree

Both y_2 and y_3 are fractional in the optimal solution to the right subproblem's LP relaxation. We want to force both variables to be integer valued. Although we could branch on either variable, we will branch on y_2. That is, we will create two more subproblems, one with $y_2 = 0$ and the other with $y_2 = 1$. After we solve the LP relaxations, we can update our tree, as in Fig. 3.2.

Note that we can immediately "$\times$ out" the left subproblem; the optimal solution to its LP relaxation is integer valued. In addition, by employing a *bounding* argument, we can also "$\times$ out" the right subproblem. The argument goes like this: Since the objective value of its LP relaxation ($3{,}636{,}666\frac{2}{3}$) is greater than the objective value of our newly found integer feasible solution (3,470,000), the optimal value of the right subproblem must be higher than (worse than) the objective value of our newly found integer feasible solution. So there is no point in expending any more effort on the right subproblem.

Since there are no active subproblems (subproblems that require branching), we are done. We have found an optimal solution to (IP). The optimal solution has factories 2 and 3 use the site-1 refining facility and factories 1 and 4 use the site-3 facility. The site-1 and site-3 facilities are constructed. The site-2 facility is not. The optimal solution costs 3,470,000 dollars per year, 370,000 dollars per year less than the solution obtained by rounding the solution to (LR).

This method is called *branch and bound*, and is the most common method for finding solutions to integer programming formulations.

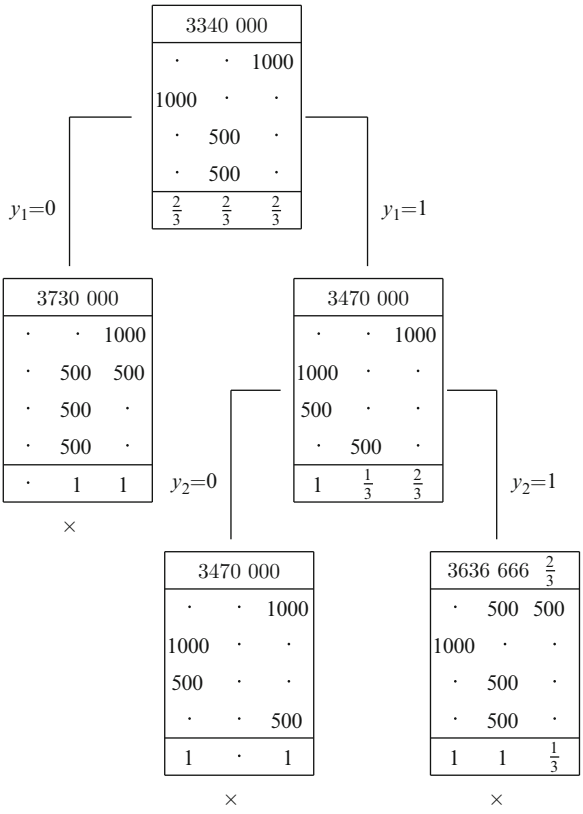

Fig. 3.2 Final branch-and-bound tree

3.1.3 Difficulties with Integer Programs

While we were able to get the optimal solution to the example integer program relatively quickly, it is not always the case that branch and bound quickly solves integer programs. In particular, it is possible that the *bounding* aspects of branch and bound are not invoked, and the branch and bound algorithm can then generate a huge number of subproblems. In the worst case, a problem with n binary variables (variables that have to take on the value 0 or 1) can have 2^n subproblems. This exponential growth is inherent in any algorithm for integer programming (unless P = NP) due to the range of problems that can be formulated within integer programming.

Despite the possibility of extreme computation time, there are a number of techniques that have been developed to increase the likelihood of finding optimal solutions quickly. After we discuss creativity in formulations, we will discuss some of these techniques.

3.2 Be Creative in Formulations

At first, it may seem that integer programming does not offer much over linear programming: both require linear objectives and constraints, and both have numerical variables. Can requiring some of the variables to take on integer values significantly expand the capability of the models? Absolutely . . . integer programming models go far beyond the power of linear programming models. The key is the creative use of integrality to model a wide range of common structures in models. Here we outline some of the major uses of integer variables.

3.2.1 Integer Quantities

The most obvious use of integer variables is when an integer quantity of a good is required. For instance, in a production model involving television sets, an integral number of television sets might be required. Or, in a personnel assignment problem, an integer number of workers might be assigned to a shift.

This use of integer variables is the most obvious, and the most over-used. For many applications, the added "accuracy" in requiring integer variables is far outweighed by the greater difficulty in finding the optimal solution. For instance, in the production example, if the number of televisions produced is in the hundreds (say the fractional optimal solution is 202.7) then having a plan with the rounded-off value (203 in this example) is likely appropriate in practice. The uncertainty of the data almost certainly means that no production plan is accurate to four figures! Similarly, if the personnel assignment problem is for a large enterprise over a year, and the linear programming model suggests 154.5 people are required, it is probably not worthwhile to invoke an integer programming model in order to handle the fractional parts.

However, there are times when integer quantities are required. A production system that can produce either two or three aircraft carriers and a personnel assignment problem for small teams of five or six people are examples. In these cases, the addition of the integrality constraint can mean the difference between useful models and irrelevant models.

3.2.2 Binary Decisions

Perhaps the most used type of integer variable is the *binary variable*: an integer variable restricted to take on the values 0 or 1. We will see a number of uses of these variables. Our first example is in modeling binary decisions.

Many practical decisions can be seen as "yes" or "no" decisions: should we construct a chemical refining facility in site j (as in the introduction), should we invest in project B, should we start producing new product Y? For many of these decisions,

a binary integer programming model is appropriate. In such a model, each decision is modeled with a binary variable: setting the variable equal to 1 corresponds to making the "yes" decision, while setting it to 0 corresponds to going with the "no" decision. Constraints are then formed to correspond to the effects of the decision.

As an example, suppose we need to choose among projects A, B, C and D. Each project has a capital requirement ($1, $2.5, $4 and $5 million respectively) and an expected return (say, $3, $6, $13 and $16 million). If we have $7 million to invest, which projects should we take on in order to maximize our expected return?

We can formulate this problem with binary variables $x_{\mathrm{A}}, x_{\mathrm{B}}, x_{\mathrm{C}}$ and x_{D} representing the decision to take on the corresponding project. The effect of taking on a project is to use up some of the funds we have available to invest. Therefore, we have a constraint

$$x_A + 2.5x_B + 4x_C + 5x_D \leq 7.$$

Our objective is to maximize the expected profit:

$$\text{Maximize } 3x_A + 6x_B + 13x_C + 16x_D.$$

In this case, binary variables let us make the "yes–no" decision on whether to invest in each fund, with a constraint ensuring that our overall decisions are consistent with our budget. Without integer variables, the solution to our model would have fractional parts of projects, which may not be in keeping with the needs of the model.

3.2.3 Fixed-Charge Requirements

In many production applications, the cost of producing x of an item is roughly linear except for the special case of producing no items. In that case, there are additional savings since no equipment or other items need be procured for the production. This leads to a *fixed-charge* structure. The cost for producing x of an item is

- 0, if $x = 0$
- $c_1 + c_2 x$, if $x > 0$ for constants c_1, c_2.

This type of cost structure is impossible to embed in a linear program. With integer programming, however, we can introduce a new binary variable y. The value $y = 1$ is interpreted as having non-zero production, while $y = 0$ means no production. The objective function for these variables then becomes

$$c_1 y + c_2 x,$$

which is appropriately linear in the variables. It is necessary, however, to add constraints that link the x and y variables. Otherwise, the solution might be $y = 0$ and $x = 10$, which we do not want. If there is an upper bound M on how large x can be (perhaps derived from other constraints), then the constraint

$$x \leq My$$

correctly links the two variables. If $y = 0$ then x must equal 0; if $y = 1$ then x can take on any value. Technically, it is possible to have the values $x = 0$ and $y = 1$ with this formulation, but as long as this is modeling a fixed cost (rather than a fixed profit), this will not be an optimal (cost minimizing) solution.

This use of "M" values is common in integer programming, and the result is called a Big-M model. Big-M models are often difficult to solve, for reasons we will see.

We saw this fixed-charge modeling approach in our initial facility location example. There, the y variables corresponded to opening a refining facility (incurring a fixed cost). The x variables correspond to assigning a factory to the refining facility, and there was an upper bound on the volume of raw material a refinery could handle.

3.2.4 Logical Constraints

Binary variables can also be used to model complicated logical constraints, a capability not available in linear programming. In a facility location problem with binary variables y_1, y_2, y_3, y_4 and y_5 corresponding to the decisions to open warehouses at locations 1, 2, 3, 4 and 5 respectively, complicated relationships between the warehouses can be modeled with linear functions of the y variables. Here are a few examples:

- At most one of locations 1 and 2 can be opened: $y_1 + y_2 \leq 1$
- Location 3 can only be opened if location 1 is: $y_3 \leq y_1$
- Location 4 cannot be opened if locations 2 or 3 are: $y_4 + y_2 \leq 1, y_4 + y_3 \leq 1$
- If location 1 is open, either locations 2 or 5 must be: $y_2 + y_5 \geq y_1$.

Much more complicated logical constraints can be formulated with the addition of new binary variables. Consider a constraint of the form $3x_1 + 4x_2 \leq 10$ OR $4x_1 + 2x_2 \geq 12$. As written, this is not a linear constraint. However, if we let M be the largest either $|3x_1 + 4x_2|$ or $|4x_1 + 2x_2|$ can be, then we can define a new binary variable z which is 1 only if the first constraint is satisfied and 0 only if the second constraint is satisfied. Then we get the constraints

$$\begin{aligned} 3x_1 + 4x_2 &\leq 10 + (M - 10)(1 - z) \\ 4x_1 + 2x_2 &\geq 12 - (M + 12)z. \end{aligned}$$

When $z = 1$, we get

$$\begin{aligned} 3x_1 + 4x_2 &\leq 10 \\ 4x_1 + 2x_2 &\geq -M. \end{aligned}$$

When $z = 0$, we get

$$3x_1 + 4x_2 \leq M$$
$$4x_1 + 2x_2 \geq 12.$$

This correctly models the original nonlinear constraint.

As we can see, logical requirements often lead to Big-M type formulations.

3.2.5 Sequencing Problems

Many problems in sequencing and scheduling require the modeling of the order in which items appear in the sequence. For instance, suppose we have a model in which there are items, where each item i has a processing time on a machine p_i. If the machine can only handle one item at a time and we let t_i be a (continuous) variable representing the start time of item i on the machine, then we can ensure that items i and j are not on the machine at the same time with the constraints

$$t_j \geq t_i + p_i \text{ IF } t_j \geq t_i$$
$$t_i \geq t_j + p_j \text{ IF } t_j < t_i.$$

This can be handled with a new binary variable y_{ij} which is 1 if $t_i \leq t_j$ and 0 otherwise. This gives the constraints

$$t_j \geq t_i + p_i - M(1-y)$$
$$t_i \geq t_j + p_j - My$$

for sufficiently large M. If y is 1 then the second constraint is automatically satisfied (so only the first is relevant) while the reverse happens for $y = 0$.

3.3 Find Formulations with Strong Relaxations

As the previous section made clear, integer programming formulations can be used for many problems of practical interest. In fact, for many problems, there are many alternative integer programming formulations. Finding a *good* formulation is key to the successful use of integer programming. The definition of a good formulation is primarily computational: a good formulation is one for which branch and bound (or another integer programming algorithm) will find and prove the optimal solution quickly. Despite this empirical aspect of the definition, there are some guidelines to help in the search for good formulations. The key to success is to find formulations whose linear relaxation is not too different from the underlying integer program.

We saw in our first example that solving linear relaxations was key to the basic integer programming algorithm. If the solution to the initial linear relaxation is integer, then no branching need be done and integer programming is no harder

than linear programming. Unfortunately, finding formulations with this property is very hard to do. But some formulations can be better than other formulations in this regard.

Let us modify our facility location problem by requiring that every factory be assigned to exactly one refinery (incidentally, the optimal solution to our original formulation happened to meet this requirement). Now, instead of having x_{ij} be the tonnes sent from factory i to refinery j, we define x_{ij} to be 1 if factory i is serviced by refinery j. Our formulation becomes

$$
\begin{aligned}
\text{minimize} \quad & 1{,}000 \cdot 52 \cdot 25x_{11} + 1{,}000 \cdot 52 \cdot 20x_{12} + 1{,}000 \cdot 52 \cdot 15x_{13} \\
+ \; & 1{,}000 \cdot 52 \cdot 15x_{21} + 1{,}000 \cdot 52 \cdot 25x_{22} + 1{,}000 \cdot 52 \cdot 20x_{23} \\
+ \; & 500 \cdot 52 \cdot 20x_{31} + 500 \cdot 52 \cdot 15x_{32} + 500 \cdot 52 \cdot 25x_{33} \\
+ \; & 500 \cdot 52 \cdot 25x_{41} + 500 \cdot 52 \cdot 15x_{42} + 500 \cdot 52 \cdot 15x_{43} \\
+ \; & 500{,}000y_1 + 500{,}000y_2 + 500{,}000y_3 \\
\text{subject to} \quad & x_{11} + x_{12} + x_{13} = 1 \\
& x_{21} + x_{22} + x_{23} = 1 \\
& x_{31} + x_{32} + x_{33} = 1 \\
& x_{41} + x_{42} + x_{43} = 1 \\
& 1{,}000x_{11} + 1{,}000x_{21} + 500x_{31} + 500x_{41} \leq 1{,}500y_1 \\
& 1{,}000x_{12} + 1{,}000x_{22} + 500x_{32} + 500x_{42} \leq 1{,}500y_2 \\
& 1{,}000x_{13} + 1{,}000x_{23} + 500x_{33} + 500x_{43} \leq 1{,}500y_3 \\
& x_{ij} \in \{0,1\} \quad \text{for all } i \text{ and } j \\
& y_j \in \{0,1\} \quad \text{for all } j.
\end{aligned}
$$

Let us call this formulation the *base formulation*. This is a correct formulation to our problem. There are alternative formulations, however. Suppose we add to the base formulation the set of constraints

$$x_{ij} \leq y_j \quad \text{for all } i \text{ and } j.$$

Call the resulting formulation the *expanded formulation*. Note that it too is an appropriate formulation for our problem. At the simplest level, it appears that we have simply made the formulation larger: there are more constraints so the linear programs solved within branch-and-bound will likely take longer to solve. Is there any advantage to the expanded formulation?

The key is to look at non-integer solutions to linear relaxations of two formulations: we know the two formulations have the same integer solutions (since they are formulations of the same problem), but they can differ in non-integer solutions. Consider the solution $x_{13} = 1, x_{21} = 1, x_{32} = 1, x_{42} = 1, y_1 = 2/3, y_2 = 2/3, y_3 = 2/3$. This solution is feasible to the linear relaxation of the base formulation but is not feasible to the linear relaxation of the expanded formulation. If the branch-and-bound algorithm works on the base formulation, it may have to consider this solution; with the expanded formulation, this solution can never be examined. If there are fewer fractional solutions to explore (technically, fractional extreme point solutions), branch and bound will typically terminate quicker.

Since we have added constraints to get the expanded formulation, there is no non-integer solution to the linear relaxation of the expanded formulation that is not also feasible for the linear relaxation of the base formulation. We say that the expanded formulation is *tighter* than the base formulation.

In general, tighter formulations are to be preferred for integer programming formulations even if the resulting formulations are larger. Of course, there are exceptions: if the size of the formulation is much larger, the gain from the tighter formulation may not be sufficient to offset the increased linear programming times. Such cases are definitely the exception, however: almost invariably, tighter formulations are better formulations. For this particular instance, the expanded formulation happens to provide an integer solution without branching.

There has been a tremendous amount of work done on finding tighter formulations for different integer programming models. For many types of problems, classes of constraints (or *cuts*) to be added are known. These constraints can be added in one of two ways: they can be included in the original formulation or they can be added as needed to remove fractional values. The latter case leads to a *branch and cut* approach, which is the subject of Sect. 3.6.

A cut relative to a formulation has to satisfy two properties: first, every feasible integer solution must also satisfy the cut; second, some fractional solution that is feasible to the linear relaxation of the formulation must not satisfy the cut. For instance, consider the single constraint

$$3x_1 + 5x_2 + 8x_3 + 10x_4 \leq 16,$$

where the x_i are binary variables. Then the constraint $x_3 + x_4 \leq 1$ is a cut (every integer solution satisfies it and, for instance $x = (0,0,0.75,1)$ does not) but $x_1 + x_2 + x_3 + x_4 \leq 4$ is not a cut (no fractional solutions removed) nor is $x_1 + x_2 + x_3 \leq 2$ (which incorrectly removes $x = (1,1,1,0)$).

Given a formulation, finding cuts to add to it to strengthen the formulation is not a routine task. It can take deep understanding, and a bit of luck, to find improving constraints.

One generally useful approach is called the Chvátal (or Gomory–Chvátal) procedure. Here is how the procedure works for "$\leq$" constraints where all the variables are non-negative integers:

1. Take one or more constraints, multiple each by a non-negative constant (the constant can be different for different constraints). Add the resulting constraints into a single "$\leq$" constraint.
2. Round down each coefficient on the left-hand side of the constraint.
3. Round down the right-hand side of the constraint.

The result is a constraint that does not cut off any feasible integer solutions. It may be a cut if the effect of rounding down the right-hand side of the constraint is more than the effect of rounding down the coefficients.

This is best seen through an example. Taking the constraint above, let us take the two constraints

$$\begin{aligned} 3x_1 + 5x_2 + 8x_3 + 10x_4 &\leq 16 \\ x_3 &\leq 1. \end{aligned}$$

If we multiply each constraint by 1/9 and add them we get

$$3/9x_1 + 5/9x_2 + 9/9x_3 + 10/9x_4 \leq 17/9.$$

Now, round down the left-hand coefficients (this is valid since the x variables are non-negative and it is a "$\leq$" constraint):

$$x_3 + x_4 \leq 17/9.$$

Finally, round down the right-hand side (this is valid since the x variables are integer) to get

$$x_3 + x_4 \leq 1$$

which turns out to be a cut. Notice that the three steps have differing effects on feasibility. The first step, since it is just taking a linear combination of constraints neither adds nor removes feasible values; the second step weakens the constraint, and may add additional fractional values; the third step strengthens the constraint, ideally removing fractional values.

This approach is particularly useful when the constants are chosen so that no rounding-down is done in the second step. For instance, consider the following set of constraints (where the x_i are binary variables):

$$\begin{aligned} x_1 + x_2 &\leq 1 \\ x_2 + x_3 &\leq 1 \\ x_1 + x_3 &\leq 1. \end{aligned}$$

These types of constraints often appear in formulations where there are lists of mutually exclusive variables. Here, we can multiply each constraint by 1/2 and add them to get

$$x_1 + x_2 + x_3 \leq 3/2.$$

There is no rounding down on the left-hand side, so we can move on to rounding down the right-hand side to get

$$x_1 + x_2 + x_3 \leq 1,$$

which, for instance, cuts off the solution $x = (1/2, 1/2, 1/2)$.

In cases where no rounding-down is needed on the left-hand side but there is rounding-down on the right-hand side, the result has to be cut (relative to the included constraints). Conversely, if no rounding down is done on the right-hand side, the result cannot be a cut.

In the formulation section, we mentioned that Big-M formulations often lead to poor formulations. This is because the linear relaxation of such a formulation often allows for many fractional values. For instance, consider the constraint (all variables are binary)

$$x_1 + x_2 + x_3 \leq 1{,}000y.$$

Such constraints often occur in facility location and related problems. This constraint correctly models a requirement that the x variables can be 1 only if y is also 1, but does so in a very weak way. Even if the x values of the linear relaxation are integer, y can take on a very small value (instead of the required 1). Here, even for $x = (1, 1, 1)$, y need only be 3/1,000 to make the constraint feasible. This typically leads to very bad branch-and-bound trees: the linear relaxation gives little guidance as to the *true* values of the variables.

Better would be the constraint

$$x_1 + x_2 + x_3 \leq 3y,$$

which forces y to take on larger values. This is the concept of making the M in Big-M as small as possible. Better still would be the three constraints

$$x_1 \leq y$$
$$x_2 \leq y$$
$$x_3 \leq y,$$

which forces y to be integer as soon as the x values are.

Finding improved formulations is a key concept to the successful use of integer programming. Such formulations typically revolve around the strength of the linear relaxation: does the relaxation well-represent the underlying integer program? Finding classes of cuts can improve formulations. Finding such classes can be difficult, but without good formulations, integer programming models are unlikely to be successful except for very small instances.

3.4 Avoid Symmetry

Symmetry often causes integer programming models to fail. Branch and bound can become an extremely inefficient algorithm when the model being solved displays many symmetries.

Consider again our facility location model. Suppose instead of having just one refinery at a site, we were permitted to have up to three refineries at a site. We could modify our model by having variables y_j, z_j and w_j for each site (representing the three refineries). In this formulation, the cost and other coefficients for y_j are the same as for z_j and w_j. The formulation is straightforward, but branch and bound does very poorly on the result.

The reason for this is symmetry: for every solution in the branch-and-bound tree with a given y, z and w, there is an equivalent solution with z taking on y's values, w taking on z's and y taking on w. This greatly increases the number of solutions the branch-and-bound algorithm must consider in order to find and prove the optimality of a solution.

It is important to remove as many symmetries in a formulation as possible. Depending on the problem and the symmetry, this removal can be done by adding constraints, fixing variables or modifying the formulation.

For our facility location problem, the easiest thing to do is to add the constraints

$$y_j \geq z_j \geq w_j \text{ for all } j.$$

Now, at a refinery site, z_j can be non-zero only if y_j is non-zero, and w_j non-zero only if both y_j and z_j are. This partially breaks the symmetry of this formulation, though other symmetries (particularly in the x variables) remain.

This formulation can be modified in another way by redefining the variables. Instead of using binary variables, let y_j be the number of refineries put in location j. This removes all of the symmetries at the cost of a weaker linear relaxation (since some of the strengthenings we have explored require binary variables).

Finally, to illustrate the use of variable fixing, consider the problem of coloring a graph with K colors: we are given a graph with node set V and edge set E and wish to determine if we can assign a value $v(i)$ to each node i such that $v(i) \in \{1,\ldots,K\}$ and $v(i) \neq v(j)$ for all $(i,j) \in E$.

We can formulate this problem as an integer programming by defining a binary variable x_{ik} to be 1 if i is given color k and 0 otherwise. This leads to the constraints

$$\begin{aligned} \textstyle\sum_k x_{ik} = 1 \qquad & \text{for all } i \text{ (every node gets a color)} \\ x_{ik} + x_{jk} = 1 \qquad & \text{for all } k, (i,j) \in E \text{ (no adjacent get same)} \\ x_{ik} \in \{0,1\} \quad & \text{for all } i,k. \end{aligned}$$

The graph-coloring problem is equivalent to determining if the above set of constraints is feasible. This can be done by using branch-and-bound with an arbitrary objective value.

Unfortunately, this formulation is highly symmetric. For any coloring of graph, there is an equivalent coloring that arises by permuting the coloring (that is, permuting the set $\{1,\ldots,k\}$ in this formulation). This makes branch and bound very ineffective for this formulation. Note also that the formulation is very weak, since setting $x_{ik} = 1/k$ for all i,k is a feasible solution to the linear relaxation no matter what E is.

We can strengthen this formulation by breaking the symmetry through variable fixing. Consider a clique (set of mutually adjacent vertices) of the graph. Each member of the clique has to get a different color. We can break the symmetry by finding a large (ideally maximum sized) clique in the graph and setting the colors of the clique arbitrarily, but fixed. So if the clique has size k_c, we would assign the colors $1,\ldots,k_c$ to members of the clique (adding in constraints forcing the corresponding x values to be 1). This greatly reduces the symmetry, since now only permutations among the colors $k_c+1,\ldots,K$ are valid. This also removes the $x_{ik} = 1/k$ solution from consideration.

3.5 Consider Formulations with Many Constraints

Given the importance of the strength of the linear relaxation, the search for improved formulations often leads to sets of constraints that are too large to include in the formulation. For example, consider a single constraint with non-negative coefficients

$$a_1x_1 + a_2x_2 + a_3x_3 + \cdots + a_nx_n \leq b,$$

where the x_i are binary variables. Consider a subset S of the variables such that $\sum_{i \in S} a_i > b$. The constraint

$$\sum_{i \in S} x_i \leq |S| - 1$$

is valid (it isn't violated by any feasible integer solution) and cuts off fractional solutions as long as S is minimal. These constraints are called *cover constraints*. We would then like to include this set of constraints in our formulation.

Unfortunately, the number of such constraints can be very large. In general, it is exponential in n, making it impractical to include the constraints in the formulation. But the relaxation is much tighter with the constraints.

To handle this problem, we can choose to generate only those constraints that are needed. In our search for an optimal integer solution, many of the constraints aren't needed. If we can generate the constraints as we need them, we can get the strength of the improved relaxation without the huge number of constraints.

Suppose our instance is

Maximize $9x_1 + 14x_2 + 20x_3 + 32x_4$
Subject to
$3x_1 + 5x_2 + 8x_3 + 10x_4 \leq 16$
$x_i \in \{0,1\}$.

The optimal solution to the linear relaxation is $x^* = (1, 0.6, 0, 1)$ with objective 49.4. Now consider the set $S = (x_1, x_2, x_4)$. The constraint

$$x_1 + x_2 + x_4 \leq 2$$

is a cut that x^* violates. If we add that constraint to our problem, we get a tighter formulation. Solving this model gives solution $x = (1, 0, 0.375, 1)$ and objective 48.5. The constraint

$$x_3 + x_4 \leq 1$$

is a valid cover constraint that cuts off this solution. Adding this constraint and solving gives solution $x = (0, 1, 0, 1)$ with objective 46. This is the optimal solution to the original integer program, which we have found only by generating cover inequalities.

In this case, the cover inequalities were easy to see, but this process can be formalized. A reasonable heuristic for identifying violated cover inequalities would be

to order the variables by decreasing $a_i x_i^*$ then add the variables to the cover S until $\sum_{i \in S} a_i > b$. This heuristic is not guaranteed to find violated cover inequalities (for that, a knapsack optimization problem can be formulated and solved) but even this simple heuristic can create much stronger formulations without adding too many constraints.

This idea is formalized in the *branch-and-cut* approach to integer programming. In this approach, a formulation has two parts: the *explicit constraints* (denoted $Ax \leq b$) and the *implicit constraints* ($A'x \leq b'$). Denote the objective function as Maximize cx. Here we will assume that all x are integral variables, but this can be easily generalized:

Step 1 Solve the linear program "Maximize cx" subject to $Ax \leq b$ to get optimal relaxation solution x^*.
Step 2 If x^* integer, then stop. x^* is optimal.
Step 3 Try to find a constraint $a'x \leq b'$ from the implicit constraints such that $a'x^* > b$. If found, add $a'x \leq b$ to the $Ax \leq b$ constraint set and go to step 1. Otherwise, do branch-and-bound on the current formulation.

In order to create a branch-and-cut model, there are two aspects: the definition of the implicit constraints, and the definition of the approach in Step 3 to find violated inequalities. The problem in Step 3 is referred to as the *separation problem* and is at the heart of the approach. For many sets of constraints, no good separation algorithm is known. Note, however, that the separation problem might be solved heuristically: it may miss opportunities for separation and therefore invoke branch-and-bound too often. Even in this case, it is often the case that the improved formulations are sufficiently tight to greatly decrease the time needed for branch-and-bound.

This basic algorithm can be improved by doing cut generation within the branch-and-bound tree. It may be that by fixing variables, different constraints become violated and those can be added to the subproblems.

3.6 Consider Formulations with Many Variables

Just as improved formulations can result from adding many constraints, adding many variables can lead to very good formulations. Let us begin with our graph coloring example. Recall that we are given a graph with vertices V and edges E and want to assign a value $v(i)$ to each node i such that $v(i) \neq v(j)$ for all $(i, j) \in E$. Our objective is to use the minimum number of different values (before, we had a fixed number of colors to use: in this section we will use the optimization version rather than the feasibility version of this problem).

Previously, we described a model using binary variables x_{ik} denoting whether node i gets color k or not. As an alternative model, let us concentrate on the set of nodes that gets the same color. Such a set must be an *independent set* (a set of mutually non-adjacent nodes) of the graph. Suppose we listed all independent sets of the graph: $S_1, S_2, \ldots, S_m$. Then we can define binary variables $y_1, y_2, \ldots, y_m$ with the

interpretation that $y_j = 1$ means that independent set S_j is part of the coloring, and $y_j = 0$ means that independent set S_j is not part of the coloring. Now our formulation becomes

Minimize $\sum_j y_j$
Subject to
$\sum_{j:i \in S_j} y_j = 1$ for all $i \in V$
$y_j \in \{0,1\}$ for all $j \in \{1 \ldots m\}$.

The constraint states that every node must be in some independent set of the coloring.

This formulation is a much better formulation that our x_{ik} formulation. It does not have the symmetry problems of the previous formulation and results in a much tighter linear relaxation. Unfortunately, the formulation is impractical for most graphs because the number of independent sets is exponential in the number of nodes, leading to an impossibly large formulation.

Just as we could handle an exponential number of constraints by generating them as needed, we can also handle an exponential number of variables by *variable generation*: the creation of variables only as they are needed. In order to understand how to do this, we will have to understand some key concepts from linear programming.

Consider a linear program, where the variables are indexed by j and the constraints indexed by i:

Maximize $\sum_j c_i x_j$
Subject to
$\sum_i a_{ij} x_{ij} \leq b_i$ for all i
$x_j \geq 0$ for all j.

When this linear program is solved, the result is the optimal solution x^*. In addition, however, there is a value called the *dual value*, denoted π_i, associated with each constraint. This value gives the marginal change in the objective value as the right-hand side for the corresponding constraint is changed. So if the right-hand side of constraint i changes to $b_i + \Delta$, then the objective will change by $\pi_i \Delta$ (there are some technical details ignored here involving how large Δ can be for this to be a valid calculation: since we are only concerned with marginal calculations, we can ignore these details).

Now, suppose there is a new variable x_{n+1}, not included in the original formulation. Suppose it could be added to the formulation with corresponding objective coefficient c_{n+1} and coefficients $a_{i,n+1}$. Would adding the variable to the formulation result in an improved formulation? The answer is certainly "no" in the case when

$$c_{n+1} < \sum_i a_{i,n+1} \pi_i.$$

In this case, the value gained from the objective is insufficient to offset the cost charged marginally by the effect on the constraints. We need $c_{n+1} - \sum_i a_{i,n+1}\pi_i > 0$ in order to possibly improve on our solution.

This leads to the idea of variable generation. Suppose you have a formulation with a huge number of variables. Rather than solve this huge formulation, begin with a smaller number of variables. Solve the linear relaxation and get dual values π. Using π, determine if there is one (or more) variables whose inclusion might improve the solution. If not, then the linear relaxation is solved. Otherwise, add one or more such variables to the formulation and repeat.

Once the linear relaxation is solved, if the solution is integer, then it is optimal. Otherwise, branch and bound is invoked, with the variable generation continuing in the subproblems.

Key to this approach is the algorithm for generating the variables. For a huge number of variables it is not enough to check all of them: that would be too time consuming. Instead, some sort of optimization problem must be defined whose solution is an improving variable. We'll illustrate this for our graph coloring problem.

Suppose we begin with a limited set of independent sets and solve our relaxation over them. This leads to a dual value π_i for each node. For any other independent set S, if $\sum_{i \in S} \pi_i > 1$, then S corresponds to an improving variable. We can write this problem using binary variables z_i corresponding to whether i is in S or not:

Maximize $\sum_i \pi_i z_i$
Subject to
$z_i + z_j \leq 1$ for all $(i, j) \in E$
$z_i \in \{0, 1\}$ for all i.

This problem is called the *maximum weighted independent set* (MWIS) problem, and, while the problem is formally hard, effective methods have been found for solving it for problems of reasonable size.

This gives a variable generation approach to graph coloring: begin with a small number of independent sets, then solve the MWIS problem, adding in independent sets until no independent set improves the current solution. If the variables are integer, then we have the optimal coloring. Otherwise we need to branch.

Branching in this approach needs special care. We need to branch in such a way that our subproblem is not affected by our branching. Here, if we simply branch on the y_j variables (so have one branch with $y_j = 1$ and another with $y_j = 0$), we end up not being able to use the MWIS model as a subproblem. In the case where $y_j = 0$ we need to find an improving set, except S_j does not count as improving. This means we need to find the second most improving set. As more branching goes on, we may need to find the third most improving, the fourth most improving, and so on. To handle this, specialized branching routines are needed (involving identifying nodes that, on one side of the branch, must be the same color and, on the other side of the branch, cannot be the same color).

Variable generation together with appropriate branching rules and variable generation at the subproblems is a method known as *branch and price*. This approach has been very successful in attacking a variety of very difficult problems over the last few years.

To summarize, models with a huge number of variables can provide very tight formulations. To handle such models, it is necessary to have a variable generation routine to find improving variables, and it may be necessary to modify the branching method in order to keep the subproblems consistent with that routine. Unlike constraint generation approaches, heuristic variable generation routines are not enough to ensure optimality: at some point it is necessary to prove conclusively that the right variables are included. Furthermore, these variable generation routines must be applied at each node in the branch-and-bound tree if that node is to be crossed out from further analysis.

3.7 Modify Branch-and-Bound Parameters

Integer programs are solved with computer programs. There are a number of computer programs available to solve integer programs. These range from basic spreadsheet-oriented systems to open-source research codes to sophisticated commercial applications. To a greater or lesser extent, each of these codes offers parameters and choices that can have a significant affect on the solvability of integer programming models. For most of these parameters, the only way to determine the best choice for a particular model is experimentation: any choice that is uniformly dominated by another choice would not be included in the software.

Here are some common key choices and parameters, along with some comments on each.

3.7.1 Description of Problem

The first issue to be handled is to determine how to describe the integer program to the optimization routine(s). Integer programs can be described as spreadsheets, computer programs, matrix descriptors and higher-level languages. Each has advantages and disadvantages with regards to such issues as ease of use, solution power, flexibility and so on. For instance, implementing a branch-and-price approach is difficult if the underlying solver is a spreadsheet program. Using "callable libraries" that give access to the underlying optimization routines can be very powerful, but can be time consuming to develop.

Overall, the interface to the software will be defined by the software. It is generally useful to be able to access the software in multiple ways (callable libraries, high-level languages, command line interfaces) in order to have full flexibility in solving.

3.7.2 Linear Programming Solver

Integer programming relies heavily on the underlying linear programming solver. Thousands or tens of thousands of linear programs might be solved in the course of branch-and-bound. Clearly a faster linear programming code can result in faster integer programming solutions. Some possibilities that might be offered are primal simplex, dual simplex, or various interior point methods. The choice of solver depends on the problem size and structure (for instance, interior point methods are often best for very large, block-structured models) and can differ for the initial linear relaxation (when the solution must be found "from scratch") and subproblem linear relaxations (when the algorithm can use previous solutions as a starting basis). The choice of algorithm can also be affected by whether constraint and/or variable generation are being used.

3.7.3 Choice of Branching Variable

In our description of branch-and-bound, we allowed branching on any fractional variable. When there are multiple fractional variables, the choice of variable can have a big effect on the computation time. As a general guideline, more "important" variables should be branched on first. In a facility location problem, the decisions on opening a facility are generally more important than the assignment of a customer to that facility, so those would be better choices for branching when a choice must be made.

3.7.4 Choice of Subproblem to Solve

Once multiple subproblems have been generated, it is necessary to choose which subproblem to solve next. Typical choices are depth-first search, breadth-first search, or best-bound search. Depth-first search continues fixing variables for a single problem until integrality or infeasibility results. This can lead quickly to an integer solution, but the solution might not be very good. Best-bound search works with subproblems whose linear relaxation is as large (for maximization) as possible, with the idea that subproblems with good linear relaxations may have good integer solutions.

3.7.5 Direction of Branching

When a subproblem and a branching variable have been chosen, there are multiple subproblems created corresponding to the values the variable can take on. The ordering of the values can affect how quickly good solutions can be found. Some choices

here are a fixed ordering or the use of estimates of the resulting linear relaxation value. With fixed ordering, it is generally good to first try the more restrictive of the choices (if there is a difference).

3.7.6 Tolerances

It is important to note that while integer programming problems are primarily combinatorial, the branch-and-bound approach uses numerical linear programming algorithms. These methods require a number of parameters giving allowable tolerances. For instance, if $x_j = 0.998$ should x_j be treated as the value 1 or should the algorithm branch on x_j? While it is tempting to give overly big values (to allow for faster convergence) or small values (to be "more accurate"), either extreme can lead to problems. While for many problems the default values from a quality code are sufficient, these values can be the source of difficulties for some problems.

3.8 Tricks of the Trade

Faced with the contents of this chapter, all of which is about "tricks of the trade", it is easy to throw one's hands up and give up on integer programming! There are so many choices, so many pitfalls, and so much chance that the combinatorial explosion will make solving problems impossible. Despite this complexity, integer programming is used routinely to solve problems of practical interest. There are a few key steps to make your integer programming implementation go well.

- Use state-of-the-art software. It is tempting to use software because it is easy, or available, or cheap. For integer programming, however, not having the most current software embedding the latest techniques can doom your project to failure. Not all such software is commercial. The COIN-OR project is an open-source effort to create high-quality optimization codes.
- Use a modeling language. A modeling language, such as OPL, Mosel, AMPL, or other language can greatly reduce development time, and allows for easy experimentation of alternatives. Callable libraries can give more power to the user, but should be reserved for "final implementations", once the model and solution approached are known.
- If an integer programming model does not solve in a reasonable amount of time, look at the formulation first, not the solution parameters. The default settings of current software are generally pretty good. The problem with most integer programming formulations is the formulation, not the choice of branching rule, for example.
- Solve some small instances and look at the solutions to the linear relaxations. Often constraints to add to improve a formulation are quite obvious from a few small examples.

- Decide whether you need "optimal" solutions. If you are consistently getting within 0.1 % of optimal, without proving optimality, perhaps you should declare success and go with the solutions you have, rather than trying to hunt down that final gap.
- Try radically different formulations. Often, there is another formulation with completely different variables, objective and constraints which will have a very different computational experience.

3.9 Conclusion

Integer programming models represent a powerful approach to solving hard problems. The bounds generated from linear relaxations are often sufficient to greatly cut down on the search tree for these problems. Key to successful integer programming is the creation of good formulations. A good formulation is one where the linear relaxation closely resembles the underlying integer program. Improved formulations can be developed in a number of ways, including finding formulations with tight relaxations, avoiding symmetry, and creating and solving formulations that have an exponential number of variables or constraints. It is through the judicious combination of these approaches, combined with fast integer programming computer codes, that the practical use of integer programming has greatly expanded in the last 20 years.

Sources of Additional Information

Integer programming has existed for more than 50 years and has developed a huge literature. This bibliography therefore makes no effort to be comprehensive, but rather provides initial pointers for further investigation.

General Integer Programming

There have been a number of excellent monographs on integer programming recently. The classic is Nemhauser and Wolsey (1998). A more recent book updating much of the material is Wolsey (1998). Schrijver (1998) is an outstanding reference book, covering the theoretical underpinnings of integer programming. An unusual resource on integer programming is a volume by Jünger et al. (2010) celebrating 50 years of integer programming. This volume contains a mix of classic papers dating back to Dantzig et al.'s paper on solving the traveling salesman problem, commentaries on those papers, and a selection of current surveys, including topics such as reformulations, symmetry, nonlinear integer programming and much more. The volume includes a DVD of presentations made at a 2008 conference held in Aussois, France.

Integer Programming Formulation

There are relatively few books on formulating problems. An exception is Williams (1999). In addition, most operations research textbooks offer examples and exercises on formulations, though many of the examples are not of realistic size. Some choices are Winston (2003), Taha (2010), and Hillier and Lieberman (2009).

Branch and Bound

Branch and bound can be traced back to the 1960s and the work of Land and Doig (1960). Most basic textbooks (see above) give an outline of the method (at the level given here).

Branch and Cut

Cutting plane approaches date back to the late 1950s and the work of Gomory (1958), whose cutting planes are applicable to any integer program. Jünger et al. (1995) provides a survey of the use of cutting plane algorithms for specialized problem classes.

As a computational technique, the work of Crowder et al. (1983) showed how cuts could greatly improve basic branch and bound.

For an example of the success of such approaches for solving extremely large optimization problems, see Applegate et al. (2011).

Branch and Price

Barnhart et al. (1998) give an excellent survey of this approach.

Implementations

There are a number of very good implementations that allow the optimization of realistic integer programs. Some of these are commercial, like IBM's ILOG CPLEX implementation, currently up to version 12.4.[1] Bixby et al. (1999) give a detailed description of the advances that this software has made.

[1] http://www-01.ibm.com/software/integration/optimization/cplex-optimizer/

Another commercial product is FICO's Xpress-MP (http://www.fico.com/en/~Products/DMTools/Pages/FICO-Xpress-Optimization-Suite.aspx), with the textbook by Gueret et al. (2002) providing a very nice set of examples and applications.

A final major commercial product is the code by Gurobi (http://www.gurobi.com). While the initial version of this code was released in 2008, this product has quickly become competitive with its older rivals.

COIN-OR (http://www.coin-or.org) provides an open-source initiative for optimization, including integer programming. Other approaches are described by Ralphs and Ladanyi (1999) and Cordier et al. (1999).

References

Applegate DL, Bixby RE, Chvatal V, Cook WJ (2011) The traveling salesman problem: a computational study. Princeton University Press

Barnhart C, Johnson EL, Nemhauser GL, Savelsbergh MWP, Vance PH (1998) Branch-and-price: column generation for huge integer programs. Oper Res 46:316

Bixby RE, Fenelon M, Gu Z, Rothberg E, Wunderling R (1999) MIP: theory and practice—closing the gap. In: Proceedings of the 19th IFIP TC7 conference on system modelling, Cambridge. Kluwer, Dordrecht, pp 19–50

Common Optimization INterface for Operations Research (COIN-OR) (2004) http://www.coin-or.org

Cordier C, Marchand H, Laundy R, Wolsey LA (1999) bc-opt: a branch-and-cut code for mixed integer programs. Math Program 86:335

Crowder H, Johnson EL, Padberg MW (1983) Solving large scale zero-one linear programming problems. Oper Res 31:803–834

Gomory RE (1958) Outline of an algorithm for integer solutions to linear programs. Bull Am Math Soc 64(5):275–278

Gueret C, Prins C, Sevaux M (2002) Applications of optimization with Xpress-MP (trans: Heipcke S). Dash Optimization, Blisworth

Hillier FS, Lieberman GJ (2009) Introduction to operations research, 9th edn. McGraw-Hill, New York

Jünger M, Reinelt G, Thienel S (1995) Practical problem solving with cutting plane algorithms in combinatorial optimization. DIMACS series in discrete mathematics and theoretical computer science. AMS, Providence, p 111

Jünger M, Liebling T, Naddef D, Nemhauser G, Pulleyblank W, Reinelt G, Rinaldi G, Wolsey L (2010) 40 years of integer programming 1958–2008. Springer, Heidelberg

Land AH, Doig AG (1960) An automatic method for solving discrete programming problems. Econometrica 28:83–97

Nemhauser GL, Wolsey LA (1998) Integer and combinatorial optimization. Wiley, New York

Ralphs TK, Ladanyi L (1999) SYMPHONY: a parallel framework for branch and cut. White paper, Rice University

Schrijver A (1998) Theory of linear and integer programming. Wiley, New York

Taha HA (2010) Operations research: an introduction, 9th edn. Prentice-Hall, New York

Williams HP (1999) Model building in mathematical programming. Wiley, New York

Winston W (2003) Operations research: applications and algorithms, 4th edn. Thomson, New York

Wolsey LA (1998) Integer programming. Wiley, New York