OR Assignment # 3

Date: 23-5-22

Q1

K20-1052.
S. M. Hassan Ali
BSE-4B

Q.1 $f_i(t) = \max \begin{cases} r(t) - c(t) + f_{i+1}(t+1), & \text{if Keep} \\ r(0) + s(t) - I - c(0) + f_{i+1}(1) & \text{if replace.} \end{cases}$

Stage 4

	K	R	Optimum Sol	
t	$r(t)+s(t+1)-c(t)$	$r(0)+s(t)+s(1)-c(0)-I$	$f_4(t)$	Decision
1	$19+60-0.6=78.4$	$20+80+80-0.2-200=-20.2$	78.4	K
2	$18.5+50-1.2=67.3$	$20+60+80-0.2-200=-40.2$	67.3	K
3	$17.2+30-1.5=45.7$	$20+50+80-0.2-200=-50.2$	45.7	K
6	Must replace.	$20+5+80-0.2-200=-95.2$	-95.2	R

Stage 3.

	K	R	Optimum Sol	
t	$r(t)-c(t)+f_4(t+1)$	$r(0)+s(t)-c(0)-I+f_4(1)$	$f_3(t)$	Decision
1	$19.0-0.6+67.3=85.7$	$20+80-0.2-200+78.4=-21.8$	85.7	K
2	$18.5-1.2+45.7=63$	$20+60-0.2-200+78.4=-41.8$	63	K
5	$14-1.8-95.2=-83$	$20+10-0.2-200+78.4=-91.8$	-83	R

Stage 2.

	K	R	Optimum Sol	
t	$r(t)-c(t)+f_3(t+1)$	$r(0)+s(t)-c(0)-I+f_3(1)$	$f_2(t)$	Decision
1	$19-0.6+63=81.4$	$20+80-0.2-200+85.7=85.5$	85.5	K
4	$15.5-1.7-83=-69.2$	$20+30-0.2-200+85.7=-64.5$	-64.5	R

Stage 1

t	$r(t)-c(t)+f_2(t+1)$	$r(0)+s(t)-c(0)-I+f_2(1)$	$f_1(t)$	Decision
3	$17.2-1.5-64.5=-48.8$	$20+50-0.2-200+85.5=55.3$	55.3	R

Total cost = $55,300.

ALBA

Q2.

Integer

11.1 PROTOTYPE EXAMPLE

The CALIFORNIA MANUFACTURING COMPANY is considering expansion by building a new factory in either Los Angeles or San Francisco, or perhaps even in both cities. It also is considering building at most one new warehouse, but the choice of location is restricted to a city where a new factory is being built. The *net present value* (total profitability considering the time value of money) of each of these alternatives is shown in the fourth column of Table 11.1. The rightmost column gives the capital required (already included in the net present value) for the respective investments, where the total capital available is $10 million. The objective is to find the feasible combination of alternatives that maximizes the total net present value.

Dynamic

Dynamic Programming

Dynamic programming is a useful mathematical technique for making a sequence of interrelated decisions. It provides a systematic procedure for determining the optimal combination of decisions.

In contrast to linear programming, there does not exist a standard mathematical formulation of "the" dynamic programming problem. Rather, dynamic programming is a general type of approach to problem solving, and the particular equations used must be developed to fit each situation. Therefore, a certain degree of ingenuity and insight into the general structure of dynamic programming problems is required to recognize when and how a problem can be solved by dynamic programming procedures. These abilities can best be developed by an exposure to a wide variety of dynamic programming applications and a study of the characteristics that are common to all these situations. A large number of illustrative examples are presented for this purpose.

10.1 A PROTOTYPE EXAMPLE FOR DYNAMIC PROGRAMMING

EXAMPLE 1 The Stagecoach Problem

The STAGECOACH PROBLEM is a problem specially constructed[1] to illustrate the features and to introduce the terminology of dynamic programming. It concerns a mythical fortune seeker in Missouri who decided to go west to join the gold rush in California during the mid-19th century. The journey would require traveling by stagecoach through unsettled country where there was serious danger of attack by marauders. Although his starting point and destination were fixed, he had considerable choice as to which states (or territories that subsequently became states) to travel through en route. The possible routes are shown in Fig. 10.1, where each state is represented by a circled letter and the direction of travel is always from left to right in the diagram. Thus, four stages (stagecoach runs) were required to travel from his point of embarkation in state *A* (Missouri) to his destination in state *J* (California).

This fortune seeker was a prudent man who was quite concerned about his safety. After some thought, he came up with a rather clever way of determining the safest route. Life insurance policies were offered to stagecoach passengers. Because the cost of the policy

[1]This problem was developed by Professor Harvey M. Wagner while he was at Stanford University.

10.3 DETERMINISTIC DYNAMIC PROGRAMMING

This section further elaborates upon the dynamic programming approach to *deterministic* problems, where the *state* at the *next stage* is *completely determined* by the *state* and *policy decision* at the *current stage*. The *probabilistic* case, where there is a probability distribution for what the next state will be, is discussed in the next section.

[4]Actually, for this problem the solution procedure can move *either* backward or forward. However, for many problems (especially when the stages correspond to *time periods*), the solution procedure *must* move backward.

Binary

come in whole units, x_1 and x_2 would have to be restricted to integer values.

There have been numerous applications of integer programming that involve a direct extension of linear programming where the divisibility assumption must be dropped. However, another area of application may be of even greater importance, namely, problems involving a number of interrelated "yes-or-no decisions." In such decisions, the only two possible choices are *yes* and *no*. For example, should we undertake a particular fixed project? Should we make a particular fixed investment? Should we locate a facility in a particular site?

With just two choices, we can represent such decisions by decision variables that are restricted to just two values, say 0 and 1. Thus, the jth yes-or-no decision would be represented by, say, x_j such that

$$x_j = \begin{cases} 1 & \text{if decision } j \text{ is yes} \\ 0 & \text{if decision } j \text{ is no.} \end{cases}$$

Such variables are called **binary variables** (or 0–1 variables). Consequently, IP problems that contain only binary variables sometimes are called **binary integer programming (BIP)** problems (or 0–1 integer programming problems).

Section 11.1 presents a miniature version of a typical BIP problem and Sec. 11.2 surveys a variety of other BIP applications. Additional formulation possibilities with binary variables are discussed in Sec. 11.3, and Sec. 11.4 presents a series of formulation examples. Sections 11.5–11.8 then deal with ways to solve IP problems, including both BIP and MIP problems. The chapter concludes in Sec. 11.9 by introducing an exciting recent development *(constraint programming)* that promises to greatly expand our ability to formulate and solve integer programming models.

mulate and solve integer programming models.

11.1 PROTOTYPE EXAMPLE

The CALIFORNIA MANUFACTURING COMPANY is considering expansion by building a new factory in either Los Angeles or San Francisco, or perhaps even in both cities. It also is considering building at most one new warehouse, but the choice of location is restricted to a city where a new factory is being built. The *net present value* (total profitability considering the time value of money) of each of these alternatives is shown in the fourth column of Table 11.1. The rightmost column gives the capital required (already included in the net present value) for the respective investments, where the total capital available is \$10 million. The objective is to find the feasible combination of alternatives that maximizes the total net present value.

The BIP Model

Although this problem is small enough that it can be solved very quickly by inspection (build factories in both cities but no warehouse), let us formulate the IP model for illustrative purposes. All the decision variables have the *binary* form

$$x_j = \begin{cases} 1 & \text{if decision } j \text{ is yes,} \\ 0 & \text{if decision } j \text{ is no,} \end{cases} \qquad (j = 1, 2, 3, 4).$$

Let

Z = total net present value of these decisions.

Perspective approach on solving integer programming

PROGRAMMING PROBLEMS

It may seem that IP problems should be relatively easy to solve. After all, *linear programming* problems can be solved extremely efficiently, and the only difference is that IP problems have far fewer solutions to be considered. In fact, *pure* IP problems with a bounded feasible region are guaranteed to have just a *finite* number of feasible solutions.

Unfortunately, there are two fallacies in this line of reasoning. One is that having a finite number of feasible solutions ensures that the problem is readily solvable. Finite numbers can be astronomically large. For example, consider the simple case of BIP problems. With n variables, there are 2^n solutions to be considered (where some of these solutions can subsequently be discarded because they violate the functional constraints). Thus, each time n is increased by 1, the number of solutions is *doubled*. This pattern is referred to as the **exponential growth** of the difficulty of the problem. With $n = 10$, there are more than 1,000 solutions (1,024); with $n = 20$, there are more than 1,000,000; with $n = 30$, there are more than 1 billion; and so forth. Therefore, even the fastest computers are incapable of performing exhaustive enumeration (checking each solution for feasibility and, if it is feasible, calculating the value of the objective value) for BIP problems with more than a few dozen variables, let alone for *general* IP problems with the same number of integer variables. Fortunately, by starting with the ideas described in subsequent sections, today's best IP algorithms are vastly superior to exhaustive enumeration. The improvement over just the past two decades has been dramatic. BIP problems that would have required years of computing time to solve 20 years ago now can be solved in seconds with today's best commercial software (such as CPLEX). This huge speedup is due to great progress in three areas—dramatic improvements in BIP algorithms (as well as other IP algorithms), striking improvements in linear programming algorithms that are heavily used within the integer programming algorithms, and the great speedup in computers (including desktop computers). As a result, vastly larger BIP problems now are sometimes being solved than would have been possible in past decades. The best algorithms today are capable of solving *some* pure BIP problems with over a hundred thousand variables. Nevertheless, because of *exponential growth,* even the best algorithms cannot be guaranteed to solve every relatively small problem (less than a few hundred binary variables). Depending on their characteristics, certain relatively small problems can be much more difficult to solve than some much larger ones.

When dealing with general integer variables instead of binary variables, the size of the problems that can be solved tend to be substantially smaller. However, there are exceptions. For example, several years ago, the professional version of CPLEX 8.0 successfully solved an IP problem with 215,000 general integer variables, 75,000 functional constraints, and 6,000,000 nonzero constraint coefficients, and current versions of CPLEX

Taco Bell Corporation has over 6,500 quick-service restaurants in the United States and a growing international market. It serves approximately 2 billion meals per year, generating about $5.4 billion in annual sales income.

At each Taco Bell restaurant, the amount of business is highly variable throughout the day (and from day to day), with a heavy concentration during the normal meal times. Therefore, determining how many employees should be scheduled to perform what functions in the restaurant at any given time is a complex and vexing problem.

To attack this problem, Taco Bell management instructed an OR team (including several consultants) to develop a new labor-management system. The team concluded that the system needed three major components: (1) *a forecasting model* for predicting customer transactions at any time, (2) *a simulation model* (such as those described in Chap. 20) to translate customer transactions to labor requirements, and (3) *an integer programming model* to schedule employees to satisfy labor requirements and minimize payroll.

The integer decision variables for this integer programming model for any restaurant are the number of employees assigned to each of the shifts that begin at various specified times. The lengths of these shifts also are decision variables (constrained to be between minimum and maximum permissible shift lengths), but *continuous* decision variables in this case, so the model is a mixed IP model. The main constraints specify that the number of employees working during each 15-minute time interval must be greater than or equal to the minimum number required during that interval (according to the forecasting model).

This MIP model is similar to the *linear programming* model for assigning employees to shifts at United Airlines facilities that is described in an application vignette in Sec. 3.4. However, the key difference is that the number of employees working shifts at Taco Bell restaurants is much smaller than the number at United Airlines facilities, so it is necessary to restrict these decision variables to integer values for the Taco Bell model (whereas noninteger values in a solution for the United Airlines can readily be rounded to integer values with little loss of accuracy).

The implementation of this MIP model along with the other components of the labor-management system has provided Taco Bell with *documented savings of* **$13 million** *per year* in labor costs.

Source: J. Hueter and W. Swart: "An Integrated Labor-Management System for Taco Bell," *Interfaces*, **28**(1): 75–91, Jan.–Feb. 1998. (A link to this article is provided on our website, www.mhhe.com/hillier.)

Branch and cut

in computation time as the problem size was increased. Many important problems arising in practice could not be solved.

Then came the next breakthrough in the mid-1980s, with the introduction of the *branch-and-cut approach* to solving BIP problems. There were early reports of very large problems with as many as a couple thousand variables being solved using this approach. This

created great excitement and led to intensive research and development activities to refine the approach that have continued ever since. At first, the approach was limited to *pure* BIP, but soon was extended to *mixed* BIP, and then to MIP problems with some general integer variables as well. We will limit our description of the approach to the *pure* BIP case.

It is fairly common now for the branch-and-cut approach to solve some problems with many thousand variables, and occasionally even hundreds of thousands of variables. As mentioned in Sec. 11.4, this tremendous speedup is due to huge progress in three areas—dramatic improvements in BIP algorithms by incorporating and further developing the branch-and-cut approach, striking improvements in linear programming algorithms that are heavily used within the BIP algorithms, and the great speedup in computers (including desktop computers).

We do need to add one note of caution. This algorithmic approach cannot consistently solve *all* pure BIP problems with a few thousand variables, or even a few hundred variables. The very large pure BIP problems solved have *sparse* **A** matrices; i.e., the percentage of coefficients in the functional constraints that are *nonzeros* is quite small (perhaps less than 5 percent, or even less than 1 percent). In fact, the approach depends heavily upon this sparsity. (Fortunately, this kind of sparsity is typical in large practical problems.) Furthermore, there are other important factors besides sparsity and size that affect just how difficult a given IP problem will be to solve. IP formulations of fairly substantial size should still be approached with considerable caution.

Although it would be beyond the scope and level of this book to fully describe the al-

Automatic Problem Preprocessing for Pure BIP

Automatic problem preprocessing involves a "computer inspection" of the user-supplied formulation of the IP problem in order to spot reformulations that make the problem quicker to solve without eliminating any feasible solutions. These reformulations fall into three categories:

1. *Fixing variables:* Identify variables that can be fixed at one of their possible values (either 0 or 1) because the other value cannot possibly be part of a solution that is both feasible and optimal.
2. *Eliminating redundant constraints:* Identify and eliminate *redundant constraints* (constraints that automatically are satisfied by solutions that satisfy all the other constraints).
3. *Tightening constraints:* Tighten some constraints in a way that reduces the feasible region for the LP relaxation without eliminating any feasible solutions for the BIP problem.

These categories are described in turn.

Fixing Variables. One general principle for fixing variables is the following.

> If one value of a variable cannot satisfy a certain constraint, even when the other variables equal their best values for trying to satisfy the constraint, then that variable should be fixed at its other value.

[9]As discussed briefly in Sec. 11.4, still another technique that has played a significant role in the recent progress has been the use of *heuristics* for quickly finding good feasible solutions.

Constraint programming

in related areas of mathematical programming, especially combinatorial optimization, but we will limit our discussion to their central use in integer programming.)

The Nature of Constraint Programming

In the mid-1980s, researchers in the computer science community began to develop constraint programming by combining ideas in artificial intelligence with the development of computer programming languages. The goal was to have a flexible computer programming system that would include both *variables* and *constraints* on their values, while also allowing the description of search procedures that would generate feasible values of the variables. Each variable has a *domain* of possible values, e.g., {2, 4, 6, 8, 10}. Rather than being limited to the types of mathematical constraints used in mathematical programming, there is great flexibility in how to state the constraints. In particular, the constraints can be any of the following types.

1. Mathematical constraints, e.g., $x + y < z$.
2. Disjunctive constraints, e.g., the times of certain tasks in the problem being modeled cannot overlap.
3. Relational constraints, e.g., at least three tasks should be assigned to a certain machine.
4. Explicit constraints, e.g., although both x and y have domains {1, 2, 3, 4, 5}, (x, y) must be (1, 1), (2, 3), or (4, 5).
5. Unary constraints, e.g., z is an integer between 5 and 10.
6. Logical constraints, e.g., if x is 5, then y is between 6 and 8.

value. It is therefore necessary to write a search procedure that will try different assignments of values to the variables. As these assignments are tried, the constraint propagation algorithm is triggered and further domain reduction occurs. The process creates a *search tree,* which is similar to the branching tree when applying the branch-and-bound technique to integer programming.

The overall process of applying constraint programming to complicated IP problems (or related problems) involves the following three steps.

1. Formulate a compact model for the problem by using a variety of constraint types (most of which do not fit the format of integer programming).
2. Efficiently find feasible solutions that satisfy all these constraints.
3. Search among these feasible solutions for an optimal solution.

The power of constraint programming lies in its great ability to perform the first two steps rather than the third, whereas the main strength of integer programming and its algorithms lie in performing the third step. Thus, constraint programming is ideally suited for a highly constrained problem that has no objective function, so the only goal is to find a feasible solution. However, it also can be extended to the third step. One method of doing so is to *enumerate* the feasible solutions and calculate the value of the objective function for each one. However, this would be extremely inefficient for problems where there are numerous feasible solutions. To circumvent this drawback, the common

The Potential of Constraint Programming

In the 1990s, constraint programming features, including powerful constraint-solving algorithms, were successfully incorporated into a number of general-purpose programming languages, as well as several special-purpose programming languages. This brought computer science closer and closer to the Holy Grail of computer programming, namely, allowing the user to simply state the problem and then the computer will solve it.

As word of this exciting development began to spread beyond the computer science community, researchers in operations research began to realize the great potential of integrating constraint programming with the traditional techniques of integer programming (and other areas of mathematical programming as well). The much greater flexibility in

expressing the constraints of the problem should greatly increase the ability to formulate valid models for complex problems. It also should lead to much more compact and straightforward formulations. In addition, by reducing the size of the feasible region that needs to be considered while efficiently finding solutions within this region, the constraint-solving algorithms of constraint programming might help accelerate the progress of integer programming algorithms in finding an optimal solution.

Because of their substantial differences, integrating constraint programming with integer programming is a very difficult task. Since integer programming does not recognize most of the constraints of constraint programming, this requires developing computer-implemented procedures for translating from the language of constraint programming to the language of integer programming and vice versa. Good progress is being made, but this undoubtedly will continue to be one of the most active areas of OR research for some years to come.

but these same types of constraints also can readily be used for some much more complicated problems.

The All-Different Constraint

The *all-different* global constraint simply specifies that all the variables in a given set must have different values. If $x_1, x_2, \ldots, x_n$ are the variables involved, the constraint can be written succinctly as

$$\text{all-different } (x_1, x_2, \ldots, x_n)$$

while also specifying the domains of the individual variables in the model. (These domains collectively need to include at least n different values in order to enforce the all-different constraint.)

To illustrate this constraint, consider the classical *assignment problem* presented in Sec. 8.3. Recall that this problem involves assigning n assignees to n tasks on a one-to-one basis so as to minimize the total cost of these assignments. Although the assignment problem is a particularly easy one to solve (as described in Sec. 8.4), it nicely illustrates how the all-different constraint can greatly simplify the formulation of the model.

With the traditional formulation presented in Sec. 8.3, the decision variables are the binary variables.

Therefore, rather than n^2 variables and $2n$ functional constraints, this complete constraint programming model (excluding the objective function) has only n variables and a *single* constraint (plus one domain for all the variables).

Now let us see how the next global constraint enables incorporating the objective function into this tiny model as well.

The Element Constraint

The *element* global constraint is most commonly used to look up a cost or profit associated with an integer variable. In particular, suppose that a variable y has domain $\{1, 2, \ldots, n\}$ and that the cost associated with each of these values is $c_1, c_2, \ldots, c_n$, respectively. Then the constraint

$$\text{element } (y, [c_1, c_2, \ldots, c_n], z)$$

constrains the variable z to equal the yth constant in the list $[c_1, c_2, \ldots, c_n]$. In other words, $z = c_y$. This variable z can now be included in the objective function to provide the cost associated with y.

To illustrate the use of the element constraint, consider the assignment problem again and let

$$c_{ij} = \text{cost of assigning assignee } i \text{ to task } j$$

for $i, j, = 1, 2, \ldots, n$. The complete constraint programming model (including the objective function for this problem is

$$\text{Minimize } Z = \sum_{i=1}^{n} z_i,$$

subject to