

20

Integer Linear Programs

Many linear programming problems require certain variables to have whole number, or integer, values. Such a requirement arises naturally when the variables represent entities like packages or people that can not be fractionally divided — at least, not in a meaningful way for the situation being modeled. Integer variables also play a role in formulating equation systems that model logical conditions, as we will show later in this chapter.

In some situations, the optimization techniques described in previous chapters are sufficient to find an integer solution. An integer optimal solution is guaranteed for certain network linear programs, as explained in Section 15.5. Even where there is no guarantee, a linear programming solver may happen to find an integer optimal solution for the particular instances of a model in which you are interested. This happened in the solution of the multicommodity transportation model (Figure 4-1) for the particular data that we specified (Figure 4-2).

Even if you do not obtain an integer solution from the solver, chances are good that you'll get a solution in which most of the variables lie at integer values. Specifically, many solvers are able to return an “extreme” solution in which the number of variables not lying at their bounds is at most the number of constraints. If the bounds are integral, all of the variables at their bounds will have integer values; and if the rest of the data is integral, many of the remaining variables may turn out to be integers, too. You may then be able to adjust the relatively few non-integer variables to produce a completely integer solution that is close enough to feasible and optimal for practical purposes.

An example is provided by the scheduling linear program of Figures 16-4 and 16-5. Since the number of variables greatly exceeds the number of constraints, most of the variables end up at their bound of 0 in the optimal solution, and some other variables come out at integer values as well. The remaining variables can be rounded up to a nearby integer solution that is a little more costly but still satisfies the constraints.

Despite all these possibilities, there remain many circumstances in which the restriction to integrality must be enforced explicitly by the solver. Integer programming solvers face a much more difficult problem than their linear programming counterparts, however; they generally require more computer time and memory, and often demand more help

from the user in formulation and in choice of options. As a result, the size of problem that you can solve will be more limited for integer programs than for linear ones.

This chapter first describes AMPL declarations of ordinary integer variables, and then introduces the use of zero-one (or binary) variables for modeling logical conditions. A concluding section offers some advice on formulating and solving integer programs effectively.

20.1 Integer variables

By adding the keyword `integer` to the qualifying phrases of a `var` declaration, you restrict the declared variables to integer values.

As an example, in analyzing the diet model in Section 2.3, we arrived at the following optimal solution:

```

ampl: model diet.mod;
ampl: data diet2a.dat;

ampl: solve;
MINOS 5.5: optimal solution found.
13 iterations, objective 118.0594032

ampl: display Buy;
Buy [*] :=
BEEF  5.36061
CHK   2
FISH  2
HAM   10
MCH   10
MTL   10
SPG   9.30605
TUR   2
;
```

If we want the foods to be purchased in integral amounts, we add `integer` to the model's `var` declaration (Figure 2-1) as follows:

```
var Buy {j in FOOD} integer >= f_min[j], <= f_max[j];
```

We can then try to re-solve:

```

ampl: model dieti.mod; data diet2a.dat;
ampl: solve;
MINOS 5.5: ignoring integrality of 8 variables
MINOS 5.5: optimal solution found.
13 iterations, objective 118.0594032
```

As you can see, the MINOS solver does not handle integrality constraints. It has ignored them and returned the same optimal value as before.

To get the integer optimum, we switch to a solver that does accommodate integrality:

```

ampl: option solver cplex;
ampl: solve;
CPLEX 8.0.0: optimal integer solution; objective 119.3
11 MIP simplex iterations
1 branch-and-bound nodes

ampl: display Buy;
Buy [*] :=
BEEF  9
  CHK  2
FISH  2
  HAM  8
  MCH 10
  MTL 10
  SPG  7
  TUR  2
;

```

Comparing this solution to the previous one, we see a few features typical of integer programming. The minimum cost has increased from \$118.06 to \$119.30; because integrality is an additional constraint on the values of the variables, it can only make the objective less favorable. The amounts of the different foods in the diet have also changed, but in unpredictable ways. The two foods that had fractional amounts in the original optimum, BEEF and SPG, have increased from 5.36061 to 9 and decreased from 9.30605 to 7, respectively; also, HAM has dropped from the upper limit of 10 to 8. Clearly, you cannot always deduce the integer optimum by rounding the non-integer optimum to the closest integer values.

20.2 Zero-one variables and logical conditions

Variables that can take only the values zero and one are a special case of integer variables. By cleverly incorporating these “zero-one” or “binary” variables into objectives and constraints, integer linear programs can specify a variety of logical conditions that cannot be described in any practical way by linear constraints alone.

To introduce the use of zero-one variables in AMPL, we return to the multicommodity transportation model of Figure 4-1. The decision variables $\text{Trans}[i, j, p]$ in this model represent the tons of product p in set PROD to be shipped from originating city i in ORIG to destination city j in DEST . In the small example of data given by Figure 4-2, the products are `bands`, `coils` and `plate`; the origins are `GARY`, `CLEV` and `PITT`, and there are seven destinations.

The cost that this model associates with shipment of product p from i to j is $\text{cost}[i, j, p] * \text{Trans}[i, j, p]$, regardless of the amount shipped. This “variable” cost is typical of purely linear programs, and in this case allows small shipments between many origin-destination pairs. In the following examples we describe ways to use zero-

one variables to discourage shipments of small amounts; the same techniques can be adapted to many other logical conditions as well.

To provide a convenient basis for comparison, we focus on the tons shipped from each origin to each destination, summed over all products. The optimal values of these total shipments are determined by a linear programming solver as follows:

```

ampl: model multi.mod;
ampl: data multi.dat;
ampl: solve;
MINOS 5.5: optimal solution found.
41 iterations, objective 199500

ampl: option display_eps .000001;
ampl: option display_transpose -10;

ampl: display {i in ORIG, j in DEST}
ampl? sum {p in PROD} Trans[i,j,p];
sum{p in PROD} Trans[i,j,p] [*,*]
:      DET   FRA   FRE   LAF   LAN   STL   WIN   :=
CLEV   625   275   325   225   400   550   200
GARY    0     0    625   150    0    625    0
PITT   525   625   225   625   100   625   175
;
```

The quantity 625 appears often in this solution, as a consequence of the multicommodity constraints:

```

subject to Multi {i in ORIG, j in DEST}:
sum {p in PROD} Trans[i,j,p] <= limit[i,j];
```

In the data for our example, $\text{limit}[i, j]$ is 625 for all i and j ; its six appearances in the solution correspond to the six routes on which the multicommodity limit constraint is tight. Other routes have positive shipments as low as 100; the four instances of 0 indicate routes that are not used.

Even though all of the shipment amounts happen to be integers in this solution, we would be willing to ship fractional amounts. Thus we will not declare the `Trans` variables to be integer, but will instead extend the model by using zero-one integer variables.

Fixed costs

One way to discourage small shipments is to add a “fixed” cost for each origin-destination route that is actually used. For this purpose we rename the `cost` parameter `vcost`, and declare another parameter `fcost` to represent the fixed assessment for using the route from i to j :

```

param vcost {ORIG,DEST,PROD} >= 0; # variable cost on routes
param fcost {ORIG,DEST} > 0;      # fixed cost on routes
```

We want $\text{fcost}[i, j]$ to be added to the objective function if the total shipment of products from i to j — that is, $\text{sum}\{p \text{ in PROD}\} \text{Trans}[i, j, p]$ — is positive; we

want nothing to be added if the total shipment is zero. Using AMPL expressions, we could write the objective function most directly as follows:

```
minimize Total_Cost:    # NOT PRACTICAL
    sum {i in ORIG, j in DEST, p in PROD}
        vcost[i,j,p] * Trans[i,j,p]
    + sum {i in ORIG, j in DEST}
        if sum {p in PROD} Trans[i,j,p] > 0 then fcost[i,j];
```

AMPL accepts this objective, but treats it as merely “not linear” in the sense of Chapter 18, so that you are unlikely to get acceptable results trying to minimize it.

As a more practical alternative, we may associate a new variable $Use[i, j]$ with each route from i to j , as follows: $Use[i, j]$ takes the value 1 if

```
sum {p in PROD} Trans[i,j,p]
```

is positive, and is 0 otherwise. Then the fixed cost associated with the route from i to j is $fcost[i, j] * Use[i, j]$, a linear term. To declare these new variables in AMPL, we can say that they are integer with bounds ≥ 0 and ≤ 1 ; equivalently we can use the keyword `binary`:

```
var Use {ORIG,DEST} binary;
```

The objective function can then be written as a linear expression:

```
minimize Total_Cost:
    sum {i in ORIG, j in DEST, p in PROD}
        vcost[i,j,p] * Trans[i,j,p]
    + sum {i in ORIG, j in DEST} fcost[i,j] * Use[i,j];
```

Since the model has a combination of continuous (non-integer) and integer variables, it yields what is known as a “mixed-integer” program.

To complete the model, we need to add constraints to assure that $Trans$ and Use are related in the intended way. This is the “clever” part of the formulation; we simply modify the `Multi` constraints cited above so that they incorporate the Use variables:

```
subject to Multi {i in ORIG, j in DEST}:
    sum {p in PROD} Trans[i,j,p] <= limit[i,j] * Use[i,j];
```

If $Use[i, j]$ is 0, this constraint says that

```
sum {p in PROD} Trans[i,j,p] <= 0
```

Since this total of shipments from i to j is a sum of nonnegative variables, it must equal 0. On the other hand, when $Use[i, j]$ is 1, the constraint reduces to

```
sum {p in PROD} Trans[i,j,p] <= limit[i,j]
```

which is the multicommodity limit as before. Although there is nothing in the constraint to directly prevent $\sum \{p \text{ in PROD}\} Trans[i, j, p]$ from being 0 when $Use[i, j]$ is 1, so long as $fcost[i, j]$ is positive this combination can never occur in an optimal solution. Thus $Use[i, j]$ will be 1 if and only if $\sum \{p \text{ in PROD}\} Trans[i, j, p]$ is positive, which is what we want. The complete model is shown in Figure 20-1a.

```

set ORIG;    # origins
set DEST;   # destinations
set PROD;   # products

param supply {ORIG,PROD} >= 0; # amounts available at origins
param demand {DEST,PROD} >= 0; # amounts required at destinations

    check {p in PROD}:
        sum {i in ORIG} supply[i,p] = sum {j in DEST} demand[j,p];

param limit {ORIG,DEST} >= 0; # maximum shipments on routes

param vcost {ORIG,DEST,PROD} >= 0; # variable shipment cost on routes
var Trans {ORIG,DEST,PROD} >= 0; # units to be shipped

param fcost {ORIG,DEST} >= 0; # fixed cost for using a route
var Use {ORIG,DEST} binary; # = 1 only for routes used

minimize Total_Cost:
    sum {i in ORIG, j in DEST, p in PROD} vcost[i,j,p] * Trans[i,j,p]
  + sum {i in ORIG, j in DEST} fcost[i,j] * Use[i,j];

subject to Supply {i in ORIG, p in PROD}:
    sum {j in DEST} Trans[i,j,p] = supply[i,p];

subject to Demand {j in DEST, p in PROD}:
    sum {i in ORIG} Trans[i,j,p] = demand[j,p];

subject to Multi {i in ORIG, j in DEST}:
    sum {p in PROD} Trans[i,j,p] <= limit[i,j] * Use[i,j];

```

Figure 20-1a: Multicommodity model with fixed costs (multmip1.mod).

To show how this model might be solved, we add a table of fixed costs to the sample data (Figure 20-1b):

```

param fcost:   FRA  DET  LAN  WIN  STL  FRE  LAF  :=
              GARY  3000 1200 1200 1200 2500 3500 2500
              CLEV  2000 1000 1500 1200 2500 3000 2200
              PITT  2000 1200 1500 1500 2500 3500 2200 ;

```

If we apply the same solver as before, the integrality restrictions on the Use variables are ignored:

```

ampl: model multmip1.mod;
ampl: data multmip1.dat;
ampl: solve;
MINOS 5.5: ignoring integrality of 21 variables
MINOS 5.5: optimal solution found.
43 iterations, objective 223504

ampl: option display_eps .000001;
ampl: option display_transpose -10;

```

```

set ORIG := GARY CLEV PITT ;
set DEST := FRA DET LAN WIN STL FRE LAF ;
set PROD := bands coils plate ;

param supply (tr):  GARY   CLEV   PITT :=
                    bands   400   700   800
                    coils   800  1600  1800
                    plate   200   300   300 ;

param demand (tr):
                    FRA  DET  LAN  WIN  STL  FRE  LAF :=
                    bands 300 300 100  75  650 225 250
                    coils 500 750 400 250 950 850 500
                    plate 100 100  0  50  200 100 250 ;

param limit default 625 ;

param vcost :=

  [*,*,bands]:  FRA  DET  LAN  WIN  STL  FRE  LAF :=
                GARY  30  10   8  10  11  71   6
                CLEV  22   7  10   7  21  82  13
                PITT  19  11  12  10  25  83  15

  [*,*,coils]:  FRA  DET  LAN  WIN  STL  FRE  LAF :=
                GARY  39  14  11  14  16  82   8
                CLEV  27   9  12   9  26  95  17
                PITT  24  14  17  13  28  99  20

  [*,*,plate]:  FRA  DET  LAN  WIN  STL  FRE  LAF :=
                GARY  41  15  12  16  17  86   8
                CLEV  29   9  13   9  28  99  18
                PITT  26  14  17  13  31 104  20 ;

param fcost:  FRA  DET  LAN  WIN  STL  FRE  LAF :=
              GARY 3000 1200 1200 1200 2500 3500 2500
              CLEV 2000 1000 1500 1200 2500 3000 2200
              PITT 2000 1200 1500 1500 2500 3500 2200 ;

```

Figure 20-1b: Data for Figure 20-1a (multmip1.dat).

```

ampl: display sum {i in ORIG, j in DEST, p in PROD}
ampl?   vcost[i,j,p] * Trans[i,j,p];
sum{i in ORIG, j in DEST, p in PROD}
      vcost[i,j,p]*Trans[i,j,p] = 199500

ampl: display Use;
Use [*,*]
:      DET      FRA      FRE      LAF      LAN      STL      WIN      :=
CLEV  1         0.44    0.52    0.36    0.64    0.88    0.32
GARY  0         0        1       0.24    0        1       0
PITT  0.84     1         0.36   1       0.16    1       0.28
;

```

As you can see, the total variable cost is the same as before, and Use assumes a variety of fractional values. This solution tells us nothing new, and there is no simple way to convert it into a good integer solution. An integer programming solver is essential to get any practical results in this situation.

Switching to an appropriate solver, we find that the true optimum with fixed costs is as follows:

```

ampl: option solver cplex; solve;
CPLEX 8.0.0: optimal integer solution; objective 229850
295 MIP simplex iterations
19 branch-and-bound nodes

ampl: display {i in ORIG, j in DEST}
ampl? sum {p in PROD} Trans[i,j,p];
sum{p in PROD} Trans[i,j,p] [*,*]
:      DET   FRA   FRE   LAF   LAN   STL   WIN      :=
CLEV   625   275    0   425   350   550   375
GARY    0    0   625    0   150   625    0
PITT   525   625   550   575    0   625    0
;

ampl: display Use;
Use [*,*]
:      DET   FRA   FRE   LAF   LAN   STL   WIN      :=
CLEV    1    1    0    1    1    1    1
GARY    0    0    1    0    1    1    0
PITT    1    1    1    1    0    1    0
;

```

Imposing the integer constraints has increased the total cost from \$223,504 to \$229,850; but the number of unused routes has increased, to seven, as we had hoped.

Zero-or-minimum restrictions

Although the fixed-cost solution uses fewer routes, there are still some on which the amounts shipped are relatively low. As a practical matter, it may be that even the variable costs are not applicable unless some minimum number of tons is shipped. Suppose, therefore, that we declare a parameter `minload` to represent the minimum number of tons that may be shipped on a route. We could add a constraint to say that the shipments on each route, summed over all products, must be at least `minload`:

```

subject to Min_Ship {i in ORIG, j in DEST}:      # WRONG
    sum {p in PROD} Trans[i,j,p] >= minload;

```

But this would force the shipments on every route to be at least `minload`, which is not what we have in mind. We want the tons shipped to be either zero, or at least `minload`. To say this directly, we might write:

```

subject to Min_Ship {i in ORIG, j in DEST}:      # NOT ALLOWED
    sum {p in PROD} Trans[i,j,p] = 0 or
    sum {p in PROD} Trans[i,j,p] >= minload;

```

But the current version of AMPL does not accept logical operators in constraints.

The desired zero-or-minimum restrictions can be imposed by employing the variables $Use[i, j]$, much as in the previous example:

```
subject to Min_Ship {i in ORIG, j in DEST}:
    sum {p in PROD} Trans[i,j,p] >= minload * Use[i,j];
```

When total shipments from i to j are positive, $Use[i, j]$ is 1, and $Min_Ship[i, j]$ becomes the desired minimum-shipment constraint. On the other hand, when there are no shipments from i to j , $Use[i, j]$ is zero; the constraint reduces to $0 \geq 0$ and has no effect.

With these new restrictions and a minload of 375, the solution is found to be as follows:

```
ampl: model multmip2.mod;
ampl: data multmip2.dat;

ampl: solve;
CPLEX 8.0.0: optimal integer solution; objective 233150
279 MIP simplex iterations
17 branch-and-bound nodes

ampl: display {i in ORIG, j in DEST}
ampl? sum {p in PROD} Trans[i,j,p];
sum{p in PROD} Trans[i,j,p] [*,*]
:      DET   FRA   FRE   LAF   LAN   STL   WIN   :=
CLEV  625   425   425    0   500   625    0
GARY   0     0   375   425    0   600    0
PITT   525   475   375   575    0   575   375
;
```

Comparing this to the previous solution, we see that although there are still seven unused routes, they are not the same ones; a substantial rearrangement of the solution has been necessary to meet the minimum-shipment requirement. The total cost has gone up by about 1.4% as a result.

Cardinality restrictions

Despite the constraints we have added so far, origin PITT still serves 6 destinations, while CLEV serves 5 and GARY serves 3. We would like to explicitly add a further restriction that each origin can ship to at most $maxserve$ destinations, where $maxserve$ is a parameter to the model. This can be viewed as a restriction on the size, or cardinality, of a certain set. Indeed, it could in principle be written in the form of an AMPL constraint as follows:

```
subject to Max_Serve {i in ORIG}:      # NOT ALLOWED
    card {j in DEST:
        sum {p in PROD} Trans[i,j,p] > 0} <= maxserve;
```

Such a declaration will be rejected, however, because AMPL currently does not allow constraints to use sets that are defined in terms of variables.

```

set ORIG;    # origins
set DEST;   # destinations
set PROD;   # products

param supply {ORIG,PROD} >= 0; # amounts available at origins
param demand {DEST,PROD} >= 0; # amounts required at destinations

    check {p in PROD}:
        sum {i in ORIG} supply[i,p] = sum {j in DEST} demand[j,p];

param limit {ORIG,DEST} >= 0; # maximum shipments on routes
param minload >= 0;          # minimum nonzero shipment
param maxserve integer > 0; # maximum destinations served

param vcost {ORIG,DEST,PROD} >= 0; # variable shipment cost on routes
var Trans {ORIG,DEST,PROD} >= 0; # units to be shipped

param fcost {ORIG,DEST} >= 0; # fixed cost for using a route
var Use {ORIG,DEST} binary; # = 1 only for routes used

minimize Total_Cost:
    sum {i in ORIG, j in DEST, p in PROD} vcost[i,j,p] * Trans[i,j,p]
    + sum {i in ORIG, j in DEST} fcost[i,j] * Use[i,j];

subject to Supply {i in ORIG, p in PROD}:
    sum {j in DEST} Trans[i,j,p] = supply[i,p];

subject to Max_Serve {i in ORIG}:
    sum {j in DEST} Use[i,j] <= maxserve;

subject to Demand {j in DEST, p in PROD}:
    sum {i in ORIG} Trans[i,j,p] = demand[j,p];

subject to Multi {i in ORIG, j in DEST}:
    sum {p in PROD} Trans[i,j,p] <= limit[i,j] * Use[i,j];

subject to Min_Ship {i in ORIG, j in DEST}:
    sum {p in PROD} Trans[i,j,p] >= minload * Use[i,j];

```

Figure 20-2a: Multicommodity model with further restrictions (multmip3.mod).

Zero-one variables again offer a convenient alternative. Since the variables $Use[i, j]$ are 1 precisely for those destinations j served by origin i , and are zero otherwise, we can write $\sum \{j \text{ in DEST}\} Use[i, j]$ for the number of destinations served by i . The desired constraint becomes:

```

subject to Max_Serve {i in ORIG}:
    sum {j in DEST} Use[i,j] <= maxserve;

```

Adding this constraint to the previous model, and setting `maxserve` to 5, we arrive at the mixed integer model shown in Figure 20-2a, with data shown in Figure 20-2b. It is optimized as follows:

```

set ORIG := GARY CLEV PITT ;
set DEST := FRA DET LAN WIN STL FRE LAF ;
set PROD := bands coils plate ;

param supply (tr):  GARY   CLEV   PITT :=
                    bands  400   700   800
                    coils  800  1600  1800
                    plate  200   300   300 ;

param demand (tr):
                    FRA  DET  LAN  WIN  STL  FRE  LAF :=
    bands  300  300  100  75  650  225  250
    coils  500  750  400  250  950  850  500
    plate  100  100   0   50  200  100  250 ;

param limit default 625 ;

param vcost :=

    [*,*,bands]:  FRA  DET  LAN  WIN  STL  FRE  LAF :=
        GARY   30   10   8   10   11   71   6
        CLEV   22   7   10   7   21   82  13
        PITT   19   11  12   10   25   83  15

    [*,*,coils]:  FRA  DET  LAN  WIN  STL  FRE  LAF :=
        GARY   39  14  11  14  16  82  8
        CLEV   27  9  12  9  26  95  17
        PITT   24  14  17  13  28  99  20

    [*,*,plate]:  FRA  DET  LAN  WIN  STL  FRE  LAF :=
        GARY   41  15  12  16  17  86  8
        CLEV   29  9  13  9  28  99  18
        PITT   26  14  17  13  31  104  20 ;

param fcost:  FRA  DET  LAN  WIN  STL  FRE  LAF :=
    GARY  3000 1200 1200 1200 2500 3500 2500
    CLEV  2000 1000 1500 1200 2500 3000 2200
    PITT  2000 1200 1500 1500 2500 3500 2200 ;

param minload := 375 ;
param maxserve := 5 ;

```

Figure 20-2b: Data for Figure 20-2a (multmip3.dat).

```

ampl: model multmip3.mod;
ampl: data multmip3.dat;

ampl: solve;
CPLEX 8.0.0: optimal integer solution; objective 235625
392 MIP simplex iterations
36 branch-and-bound nodes

```

```

amp1: display {i in ORIG, j in DEST}
amp1?   sum {p in PROD} Trans[i,j,p];
sum{p in PROD} Trans[i,j,p] [*,*]
:      DET   FRA   FRE   LAF   LAN   STL   WIN   :=
CLEV   625   375   550    0   500   550    0
GARY   0     0     0    400   0    625   375
PITT   525   525   625   600   0    625    0
;

```

At the cost of a further 1.1% increase in the objective, rearrangements have been made so that GARY can serve WIN, bringing the number of destinations served by PITT down to five.

Notice that this section's three integer solutions have served WIN from each of the three different origins — a good example of how solutions to integer programs can jump around in response to small changes in the restrictions.

20.3 Practical considerations in integer programming

As a rule, any integer program is much harder to solve than a linear program of the same size and structure. A rough idea of the difference in the previous examples is given by the number of iterations reported by the solvers; it is 41 for solving the original linear multicommodity transportation problem, but ranges from about 280 to 400 for the mixed integer versions. The computation times vary similarly; the linear programs are solved almost instantly, while the mixed integer ones are noticeably slower.

As the size of the problem increases, the difficulty of an integer program also grows more quickly than the difficulty of a comparable linear program. Thus the practical limits to the size of a solvable integer program will be much more restrictive. Indeed, AMPL can easily generate integer programs that are too difficult for your computer to solve in a reasonable amount of time or memory. Before you make a commitment to use a model with integer variables, therefore, you should consider whether an alternative continuous linear or network formulation might give adequate answers. If you must use an integer formulation, try it on collections of data that increase gradually in size, so that you can get an idea of the computer resources required.

If you do encounter an integer program that is difficult to solve, you should consider reformulations that can make it easier. An integer programming solver attempts to investigate all of the different possible combinations of values of the integer variables; although it employs a sophisticated strategy that rules out the vast majority of combinations as infeasible or suboptimal, the number of combinations remaining to be checked can still be huge. Thus you should try to use as few integer variables as possible. For those variables that are not zero-one, lower and upper bounds should be made as tight as possible to reduce the number of combinations that might be investigated.

Solvers derive valuable information about the solutions to an integer program by fixing or restricting some of the integer variables, then solving the linear programming

“relaxation” that results when the remaining integer restrictions are dropped. You may be able to assist in this strategy by reformulating the model so that the solution of a relaxation will be closer to the solution of the associated integer problem. In the multicommodity transportation model, for example, if $Use[i, j]$ is 0 then each of the individual variables $Trans[i, j, p]$ must be 0, while if $Use[i, j]$ is 1 then $Trans[i, j, p]$ cannot be larger than either $supply[i, p]$ or $demand[j, p]$. This suggests that we add the following constraints:

```
subject to Avail {i in ORIG, j in DEST, p in PROD}:  
    Trans[i, j, p] <= min(supply[i, p], demand[j, p]) * Use[i, j];
```

Although these constraints do not rule out any previously admissible integer solutions, they tend to force $Use[i, j]$ to be closer to 1 for any solution of the relaxation that uses the route from i to j . As a result, the relaxation is more accurate, and may help the solver to find the optimum integer solution more quickly; this advantage may outweigh the extra cost of handling more constraints. Tradeoffs of this kind are most often observed for problems substantially larger than the examples in this section, however.

As this example suggests, the choice of a formulation is much more critical in integer than in linear programming. For large problems, solution times can change dramatically in response to simple reformulations. The effect of a reformulation can be hard to predict, however; it depends on the structure of your model and data, and on the details of the strategy used by your solver. Generally you will need to do some experimentation to see what works best.

You can also try to help a solver by changing some of the default settings that determine how it initially processes a problem and how it searches for integer solutions. Many of these settings can be manipulated from within AMPL, as explained in the separate instructions for using particular solvers.

Finally, a solver may provide options for stopping prematurely, returning an integer solution that has been determined to bring the objective value to within a small percentage of optimality. In some cases, such a solution is found relatively early in the solution process; you may save a great deal of computation time by not insisting that the solver go on to find a provably optimal integer solution.

In summary, experimentation is usually necessary to solve integer or mixed-integer linear programs. Your chances of success are greatest if you approach the use of integer variables with caution, and if you are willing to keep trying alternative formulations, settings and strategies until you get an acceptable result.

Bibliography

Ellis L. Johnson, Michael M. Kostreva and Uwe H. Suhl, “Solving 0-1 Integer Programming Problems Arising from Large-Scale Planning Models.” *Operations Research* **33** (1985) pp. 803–819. A case study in which preprocessing, reformulation, and algorithmic strategies were brought to bear on the solution of a difficult class of integer linear programs.

George L. Nemhauser and Laurence A. Wolsey, *Integer and Combinatorial Optimization*. John Wiley & Sons (New York, NY, 1988). A survey of integer programming problems, theory and algorithms.

Alexander Schrijver, *Theory of Linear and Integer Programming*. John Wiley & Sons (New York, NY, 1986). A guide to the fundamentals of the subject, with a particularly thorough collection of references.

Laurence A. Wolsey, *Integer Programming*. Wiley-Interscience (New York, NY, 1998). A practical, intermediate-level guide for readers familiar with linear programming.

Exercises

20-1. Exercise 1-1 optimizes an advertising campaign that permits arbitrary combinations of various media. Suppose that instead you must allocate a \$1 million advertising campaign among several media, but for each, your only choice is whether to use that medium or not. The following table shows, for each medium, the cost and audience in thousands if you use that medium, and the creative time in three categories that will have to be committed if you use that medium. The final column shows your limits on cost and on hours of creative time.

	TV	Magazine	Radio	Newspaper	Mail	Phone	Limits
Audience	300	200	100	150	100	50	
Cost	600	250	100	120	200	250	1000
Writers	120	50	20	40	30	5	200
Artists	120	80	0	60	40	0	300
Others	20	20	20	20	20	200	200

Your goal is to maximize the audience, subject to the limits.

- Formulate an AMPL model for this situation, using a zero-one integer variable for each medium.
- Use an integer programming solver to find the optimal solution.

Also solve the problem with the integrality restrictions relaxed and compare the resulting solution. Could you have guessed the integer optimum from looking at the non-integer one?

20-2. Return to the multicommodity transportation problem of Figures 4-1 and 4-2. Use appropriate integer variables to impose each of the restrictions described below. (Treat each part separately; don't try to put all the restrictions in one model.) Solve the resulting integer program, and comment in general terms on how the solution changes as a result of the restrictions. Also solve the corresponding linear program with the integrality restrictions relaxed, and compare the LP solution to the integer one.

- Require the amount shipped on each origin-destination link to be a multiple of 100 tons. To accommodate this restriction, allow demand to be satisfied only to the nearest 100 — for example, since demand for bands at FRE is 225 tons, allow either 200 or 300 tons to be shipped to FRE.
- Require each destination except STL to be served by at most two origins.
- Require the number of origin-destination links used to be as small as possible, regardless of cost.
- Require each origin that supplies product p to destination j to ship either nothing, or at least the lesser of $\text{demand}[j, p]$ and 150 tons.

20-3. Employee scheduling problems are a common source of integer programs, because it may not make sense to schedule a fraction of a person.

(a) Solve the problem of Exercise 4-4(b) with the requirement that the variables Y_{st} , representing the numbers of crews employed on shift s in period t , must all be integer. Confirm that the optimal integer solution just “rounds up” the linear programming solution to the next highest integer.

(b) Similarly attempt to find an integer solution for the problem of Exercise 4-4(c), where inventories have been added to the formulation. Compare the difficulty of this problem to the one in (a). Is the optimal integer solution the same as the rounded-up linear programming solution in this case?

(c) Solve the scheduling problem of Figures 16-4 and 16-5 with the requirement that an integer number of employees be assigned to each shift. Show that this solution is better than the one obtained by rounding up.

(d) Suppose that the number of supervisors required is 1/10 the number of employees, rounded to the nearest whole number. Solve the scheduling problem again for supervisors. Does the integer program appear to be harder or easier to solve in this case? Is the improvement over the rounded-up solution more or less pronounced?

20-4. The so-called knapsack problem arises in many contexts. In its simplest form, you start with a set of objects, each having a known weight and a known value. The problem is to decide which items to put into your knapsack. You want these items to have as great a total value as possible, but their weight cannot exceed a certain preset limit.

(a) The data for the simple knapsack problem could be written in AMPL as follows:

```
set OBJECTS;
param weight {OBJECTS} > 0;
param value {OBJECTS} > 0;
```

Using these declarations, formulate an AMPL model for the knapsack problem.

Use your model to solve the knapsack problem for objects of the following weights and values, subject to a weight limit of 100:

object	a	b	c	d	e	f	g	h	i	j
weight	55	50	40	35	30	30	15	15	10	5
value	1000	800	750	700	600	550	250	200	200	150

(b) Suppose instead that you want to fill several identical knapsacks, as specified by this extra parameter:

```
param knapsacks > 0 integer; # number of knapsacks
```

Formulate an AMPL model for this situation. Don’t forget to add a constraint that each object can only go into one knapsack!

Using the data from (a), solve for 2 knapsacks of weight limit 50. How does the solution differ from your previous one?

(c) Superficially, the preceding knapsack problem resembles an assignment problem; we have a collection of objects and a collection of knapsacks, and we want to make an optimal assignment from members of the former to members of the latter. What is the essential difference between the kinds of assignment problems described in Section 15.2, and the knapsack problem described in (b)?

(d) Modify the formulation from (a) so that it accommodates a volume limit for the knapsack as well as a weight limit. Solve again using the following volumes:

object	a	b	c	d	e	f	g	h	i	j
volume	3	3	3	2	2	2	2	1	1	1

How do the total weight, volume and value of this solution compare to those of the solution you found in (a)?

(e) How can the media selection problem of Exercise 20-1 be viewed as a knapsack problem like the one in (d)?

(f) Suppose that you can put up to 3 of each object in the knapsack, instead of just 1. Revise the model of (a) to accommodate this possibility, and re-solve with the same data. How does the optimal solution change?

(g) Review the roll-cutting problem described in Exercise 2-6. Given a supply of wide rolls, orders for narrower rolls, and a collection of cutting patterns, we seek a combination of patterns that fills all orders using the least material.

When this problem is solved, the algorithm returns a “dual value” corresponding to each ordered roll width. It is possible to interpret the dual value as the saving in wide rolls that you might achieve for each extra narrow roll that you obtain; for example, a value of 0.5 for a 50" roll would indicate that you might save 5 wide rolls if you could obtain 10 extra 50" rolls.

It is natural to ask: Of all the patterns of narrow rolls that fit within one wide roll, which has the greatest total (dual) value? Explain how this can be regarded as a knapsack problem.

For the problem of Exercise 2-6(a), the wide rolls are 110"; in the solution using the six patterns given, the dual values for the ordered widths are:

20"	0.0
40"	0.5
50"	1.0
55"	0.0
75"	1.0

What is the maximum-value pattern? Show that it is not one of the ones given, and that adding it allows you to get a better solution.

20-5. Recall the multiperiod model of production that was introduced in Section 4.2. Add zero-one variables and appropriate constraints to the formulation from Figure 4-4 to impose each of the restrictions described below. (Treat each part separately.) Solve with the data of Figure 4-5, and confirm that the solution properly obeys the restrictions.

(a) Require for each product p and week τ , that either none of the product is made that week, or at least 2500 tons are made.

(b) Require that only one product be made in any one week.

(c) Require that each product be made in at most two weeks out of any three-week period. Assume that only bands were made in “week -1 ” and that both bands and coils were made in “week 0”.