IV. Integer Linear Programs

Robert Fourer

Department of Industrial Engineering and Management Sciences
Northwestern University
Evanston, Illinois 60208-3119, U.S.A.

(847) 491-3151
4er@iems.northwestern.edu
http://www.iems.northwestern.edu/~4er/

Copyright © 1989–2005 Robert Fourer
Introduction

Many linear programming problems require whole number, or *integer*, values of the variables in order to be properly solved. Such a requirement arises naturally when the variables represent entities like packages or people that cannot 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 simplex method for linear programs is able to find integer solutions. An integer optimal solution is guaranteed for certain network linear programs, as explained in Chapter 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. In Chapter 6, this happened in the solution of the multicommodity transportation model (multi1.mod) for the particular data that we specified (multi.dat).

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. In particular, solvers based on the so-called simplex method — described in of Part I of *Optimization Methods* — are able to return an “extreme” solution in which the number of variables not lying at their bounds is at most the total 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, then many of the remaining variables will 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 has been provided in Chapter 3 by the scheduling linear program of sched.mod and sched.dat. 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.

All of these possibilities notwithstanding, there remain many circumstances in which the restriction to integrality must be enforced explicitly by the solver. Even though the number of feasible integer solutions is finite, compared to an uncountably infinite number of feasible solutions to an LP, integer programming solvers face a much more difficult problem than their linear programming counterparts. Indeed, integer programming lies in the category of "hard" or NP-complete problems, introduced in Chapter 15 of *Optimization Methods*. In comparison to linear programming, integer programming generally requires more computer time and memory, and often demands more help from the user in formulation and in choice of options. As a result, the sizes of problems that can be solved are more limited for integer programs than for linear ones. The relationship between the size and the difficulty of solution is also much harder to predict.

This part first describes AMPL declarations of ordinary integer variables, and then introduces the use of zero-one (or binary) integer variables for model-
ing logical conditions. Part V of *Optimization Methods* introduces the so-called branch-and-bound approach underlying most integer programming solvers, and will offer advice on formulating integer programs in ways that make them easier to solve.
9. Integer Variables

Any optimization problem can be turned into an integer program by simply stipulating that the variables must take integer values. In practice, the integer programs of interest are almost always derived from linear programs, and that is the case we will consider in this chapter.

As an example, we have seen that when the simplex method is applied to one of our small diet models, it returns the following solution:

```ampl
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;
```

Because three of the variables are at their lower limit of 2 and three are at their upper limit of 10, only two are fractional in the optimal solution. It's not unusual for a linear program to have a near-integer solution like this.

If we want all of the foods to be purchased in integral amounts, however, then we must do some more work. One simple approach is to round all fractional values to nearby integers. If we round each fractional amount in our solution to the nearest integer, however — so that we buy 5 packages of BEEF and 9 of SPG — then the total cost falls below the minimum found by the solver, and the rounded solution must fail to be feasible. (You can check that the amount of B2 in the diet falls 10 below the requirement.) We could instead round up the fractional variables to the next highest values. Then the total cost must go up, but again the solution is infeasible, this time exceeding the NA limit:

```ampl
ampl: let Buy['BEEF'] := 6;
ampl: let Buy['SPG'] := 10;
ampl: display diet.lb, diet.body, diet.ub;
: diet.lb diet.body diet.ub :=
  A 700  2012 20000
  B1 700  1060 20000
  B2 700  720  20000
  C  700  1730 20000
  CAL 16000 20240 24000
  NA  0  51522 50000
;```
Further experimentation shows that NA continues to exceed its limit, though by a lesser amount, when one of the fractions is rounded up and the other down.

This behavior is typical of integer programs. In general, there is no obvious way to go from a feasible fractional solution to a feasible integer one. Going from an optimal fractional solution to an optimal integer one is much harder still. Rounding an optimal fractional solution, in particular, usually does no better than to get you a good solution that nearly satisfies the constraints. Even if the rounded solution is feasible, moreover, there is in general no simple way to tell how close to the best integer solution it might be.

Rather than trying to fix up a fractional solution, we can add the requirement of integrality to the model. In AMPL this is simply a matter of adding integer to the model's var declaration:

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

Saving the revised model in dieti.mod, we can then try to re-solve using the same data:

```ampl
ampl: reset;
ampl: model dieti.mod;
ampl: 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 solver that we are using here does not accommodate integrality constraints. It has ignored them and has returned the same optimal value as before. To get the integer optimum, we switch to a solver that does accommodate integrality:

```ampl
ampl: option solver cplex;
ampl: solve;
CPLEX 5.0: optimal integer solution; objective 119.3
80 simplex iterations
67 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 that 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 changed in less
predictable ways. The two foods that had fractional amounts in the original optimal solution, BEEF and SPG, have increased from 5.36061 to 9 and decreased from 9.30605 to 7, respectively, while HAM has dropped from the upper limit of 10 to 8.

There is also a change in the work involved in finding the optimal solution. Only 13 simplex iterations were needed to find the fractional solution, but 80 were needed to find the integer one. The disparity only grows, moreover, as the problem size and complexity increases. For our large diet problem, the number of iterations grows by a factor of over a thousand:

```ampl
ampl: model diet2.mod;
ampl: data diet2.dat;
ampl: solve;
63 variables, all integer
17 constraints, all linear; 639 nonzeros
1 linear objective; 52 nonzeros.
MINOS 5.5: ignoring integrality of 63 variables
MINOS 5.5: optimal solution found.
40 iterations, objective 7.798353835
ampl: option solver cplex;
ampl: solve;
CPLEX 5.0: optimal integer solution within mipgap or absmipgap;
  objective 9.06
44960 simplex iterations
21362 branch-and-bound nodes
```

The method being used for integer programming in this example solves one linear program for each of the 21362 “branch-and-bound nodes” mentioned in its output. By saving the optimal bases from some of these LPs to serve as starting points for others, and by using an alternate form of the simplex method tailored to this situation, the solver manages to average only about 2.1 iterations per node. Yet with the number of nodes being so large, solving all of the LPs adds up to a substantial amount of work. The time to find the fractional solution is measured in hundredths of a second on current computers, while the time to find the integer solution is typically over 10 seconds. Part V of *Optimization Methods* will explain this disparity in its introduction to the branch-and-bound approach.
10. 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, we turn to the multicommodity transportation model of Figure 10–1. The decision variables $\text{Trans}[i, j, p]$ in this model represent the tons of product $p$ in set $\text{PROD}$ to be shipped from originating city $i$ in $\text{ORIG}$ to destination city $j$ in $\text{DEST}$. In the small example of data given by Figure 10–2, 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 $\text{cost}[i, j, p] \times \text{Trans}[i, j, p]$, 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. We then suggest how the same techniques can be adapted to other logical conditions.

To provide a convenient basis for comparison, we focus on the tons shipped

```
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;
param cost {ORIG,DEST,PROD} >= 0; # shipment costs per unit
var Trans {ORIG,DEST,PROD} >= 0; # units to be shipped
minimize Total_Cost:
  sum {i in ORIG, j in DEST, p in PROD} cost[i,j,p] \times Trans[i,j,p];
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];
```

Figure 10–1. A simple multicommodity flow model in AMPL (multi.mod).
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 cost :=
   [*,*,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 ;

Figure 10–2. Sample data for the multicommodity flow model (multi.dat).

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
ampl: model multi.mod;
ampl: data multi.dat;
ampl: solve;
MINOS 5.5: optimal solution found.
41 iterations, objective 199500
ampl: option display_transpose -10;
ampl: display {i in ORIG, j in DEST} Trans[i,j,p];
```

\[
\sum_{p \in \text{PROD}} \text{Trans}[i,j,p] =\]

\[
\text{CLEV} \quad 625 \quad 275 \quad 325 \quad 225 \quad 400 \quad 550 \quad 200
\]

\[
\text{GARY} \quad 0 \quad 0 \quad 625 \quad 150 \quad 0 \quad 625 \quad 0
\]

\[
\text{PITT} \quad 525 \quad 625 \quad 225 \quad 625 \quad 100 \quad 625 \quad 175
\]

; \]

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

\[
\text{subject to Multi } \{i \in \text{ORIG}, j \in \text{DEST}\}:
\sum_{p \in \text{PROD}} \text{Trans}[i,j,p] \leq \text{limit}[i,j];
\]

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 \( \text{Trans} \) variables to be integer, but will instead extend the model by using other integer variables.

### 10.1 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 \( \text{vcost} \), and declare another parameter \( \text{fcost} \) to represent the fixed assessment for using the route from \( i \) to \( j \):

\[
\text{param vcost } \{\text{ORIG,DEST,PROD}\} \geq 0; \quad \# \text{ variable cost on routes}
\]

\[
\text{param fcost } \{\text{ORIG,DEST}\} > 0; \quad \# \text{ fixed cost on routes}
\]

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, \( \sum_{p \in \text{PROD}} \text{Trans}[i,j,p] \) — 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:

\[
\text{minimize All\_Cost:} \quad \# \text{ NOT PRACTICAL}
\]

\[
\sum_{i \in \text{ORIG}, j \in \text{DEST}, p \in \text{PROD}} \text{vcost}[i,j,p] \cdot \text{Trans}[i,j,p]
\]

\[
+ \sum_{i \in \text{ORIG}, j \in \text{DEST}} \text{if } \sum_{p \in \text{PROD}} \text{Trans}[i,j,p] > 0 \text{ then } \text{fcost}[i,j];
\]

AMPL lets us add this objective to the model, and display its value after solving:

\[
\text{ampl: option solver minos;}
\]

\[
\text{ampl: objective Total\_Cost;}
\]

\[
\text{ampl: solve;}
\]

\[
\text{MINOS 5.5: optimal solution found.}
\]

\[
41 \text{ iterations, objective 199500}
\]

\[
\text{ampl: display All\_Cost;}
\]

\[
\text{All\_Cost = 235800}
\]
If we try to minimize the objective directly in this form, however, the most widely used kinds of solvers fail to give acceptable results. Linear and integer programming codes reject such an objective as nonlinear:

```
ampl: option solver cplex;
ampl: objective All_Cost;
ampl: solve;
/tmp/atl7745.nl contains a nonlinear objective.
```

Classical nonlinear solvers are fooled by the discontinuity at zero:

```
ampl: option solver minos;
ampl: objective All_Cost;
ampl: solve;
MINOS 5.5: the current point cannot be improved.
81 iterations, objective 235225
```

The message in this case indicates that the solver is “stuck” at a solution where it is unable to confirm the conditions for optimality.

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

$$
\sum \{p \in \text{PROD}\} \text{Trans}[i,j,p]
$$

is positive, and is 0 otherwise. Then the fixed cost associated with the route from $i$ to $j$ is just $f\text{cost}[i,j] \times \text{Use}[i,j]$, a linear term. To declare these new variables in AMPL, we can say that they are integer with bounds $\geq 0$ and $\leq 1$; 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 \text{ORIG}, j \in \text{DEST}, p \in \text{PROD}\}
  \text{vcost}[i,j,p] \times \text{Trans}[i,j,p]
  + \sum \{i \in \text{ORIG}, j \in \text{DEST}\} \text{fcost}[i,j] \times \text{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*, often abbreviated (and pronounced) *MIP*.

To complete the model, we need to add constraints to assure that $\text{Trans}$ and $\text{Use}$ 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 $\text{Use}$ variables:

```
subject to Multi \{i \in \text{ORIG}, j \in \text{DEST}\}:
  \sum \{p \in \text{PROD}\} \text{Trans}[i,j,p] \leq \text{limit}[i,j] \times \text{Use}[i,j];
```

If $\text{Use}[i,j]$ is 0, this constraint says that

$$
\sum \{p \in \text{PROD}\} \text{Trans}[i,j,p] \leq 0
$$
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;
param vcost {ORIG,DEST,PROD} >= 0; # shipment costs per unit
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 10–3. A multicommodity flow model with fixed costs (multmip1.mod).

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

\[ \sum_{p} Trans[i,j,p] \leq limit[i,j] \]

which is the multicommodity limit as before. Although there is nothing in the constraint to directly prevent \( \sum_{p} Trans[i,j,p] \) 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 \( \sum_{p} Trans[i,j,p] \) is positive, which is what we want. The complete model is shown in Figure 10–3.

To show how this model might be solved, we add a table of fixed costs to the sample data from Figure 10–2:

\[
\begin{array}{cccccccc}
\text{fcost} & \text{FRA} & \text{DET} & \text{LAN} & \text{WIN} & \text{STL} & \text{FRE} & \text{LAF} \\
\hline
\text{GARY} & 3000 & 1200 & 1200 & 1200 & 2500 & 3500 & 2500 \\
\text{CLEV} & 2000 & 1000 & 1500 & 1200 & 2500 & 3000 & 2200 \\
\text{PITT} & 2000 & 1200 & 1500 & 1500 & 2500 & 3500 & 2200 \\
\end{array}
\]

If we apply the same solver as before, the integrality restrictions on the Use variables are ignored:
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 such a solver, we find the true optimum with fixed costs as follows:

```ampl
ampl: option solver cplex;
ampl: solve;
CPLEX 5.0: optimal integer solution; objective 229850
470 simplex iterations
112 branch-and-bound nodes
```

```ampl
display {i in ORIG, j in DEST} 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
;
```

The reported minimum cost is $229,850 — higher than the $199,500 with variable costs only, but lower than the $235,800 in variable and fixed costs that we found by inappropriate use of a nonlinear solver. The number of unused routes
in our new solution has increased, to seven, as we expected.

10.2 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 LINEAR
    sum {p in PROD} Trans[i,j,p] = 0 or
    sum {p in PROD} Trans[i,j,p] >= minload;

But then we have something more complicated than the individual linear equalities and inequalities that standard integer programming solvers are designed to accept. Solvers and modeling languages that recognize or as an operator on linear constraints are still in an early stage of development.

The desired zero-or-minimum restrictions can be imposed in the context of integer programming by a clever application of 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 this becomes the desired minimum-shipment constraint. On the other hand, when there are no shipments from i to j, Use[i,j] is 0; the constraint reduces to 0 >= 0 and has no effect.

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

ampl: model multip2.mod;
ampl: data multip2.dat;
ampl: solve;
CPLEX 5.0: optimal integer solution; objective 233150
798 simplex iterations
146 branch-and-bound nodes
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.

10.3 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 could explicitly add a further restriction that each origin can ship to at most a certain number of destinations maxserve specified as 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 LINEAR
    card {j in DEST:
        sum {p in PROD} Trans[i,j,p] > 0} <= maxserve;
```

Integer programming solvers make no provision for constraining the size of a set that is defined in terms of variables, however. In fact, the current version of the AMPL language would reject this declaration as an error.

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 in DEST} Use[i,j] for the number of destinations served by i. The desired constraint then becomes:

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

Adding this constraint, we arrive at the mixed integer model shown in Figure 10–4. With maxserve set at 5 in the data, an optimum is determined as follows:

```ampl
ampl: model multmip3.mod;
ampl: data multmip3.dat;
ampl: solve;
CPLEX 5.0: optimal integer solution; objective 235625
585 simplex iterations
91 branch-and-bound nodes
ampl: display {i in ORIG, j in DEST}
ampl? sum {p in PROD} Trans[i,j,p];
```
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 5. 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 model.