8

Linear Programs: Variables, Objectives and Constraints

The best-known kind of optimization model, which has served for all of our examples so far, is the linear program. The variables of a linear program take values from some continuous range; the objective and constraints must use only linear functions of the variables. Previous chapters have described these requirements informally or implicitly; here we will be more specific.

Linear programs are particularly important because they accurately represent many practical applications of optimization. The simplicity of linear functions makes linear models easy to formulate, interpret, and analyze. They are also easy to solve; if you can express your problem as a linear program, even in thousands of constraints and variables, then you can be confident of finding an optimal solution accurately and quickly.

This chapter describes how variables are declared, defines the expressions that AMPL recognizes as being linear in the variables, and gives the rules for declaring linear objectives and constraints. Much of the material on variables, objectives and constraints is basic to other AMPL models as well, and will be used in later chapters.

Because AMPL is fundamentally an algebraic modeling language, this chapter concentrates on features for expressing linear programs in terms of algebraic objectives and constraints. For linear programs that have certain special structures, such as networks, AMPL offers alternative notations that may make models easier to write, read and solve. Such special structures are among the topics of Chapters 15 through 17.

8.1 Variables

The variables of a linear program have much in common with its numerical parameters. Both are symbols that stand for numbers, and that may be used in arithmetic expressions. Parameter values are supplied by the modeler or computed from other values, while the values of variables are determined by an optimizing algorithm (as implemented in one of the packages that we refer to as solvers).

Syntactically, variable declarations are the same as the parameter declarations defined in Chapter 7, except that they begin with the keyword `var` rather than `param`. The meaning of qualifying phrases within the declaration may be different, however, when these phrases are applied to variables rather than to parameters.

Phrases beginning with >= or <= are by far the most common in declarations of variables for linear programs. They have appeared in all of our examples, beginning with the production model of Figure 1-4:

```
var Make {p in PROD} >= 0, <= market[p];
```

This declaration creates an indexed collection of variables `Make[p]`, one for each member `p` of the set `PROD`; the rules in this respect are exactly the same as for parameters. The effect of the two qualifying phrases is to impose a restriction, or constraint, on the permissible values of the variables. Specifically, `>= 0` implies that all of the variables `Make[p]` must be assigned nonnegative values by the optimizing algorithm, while the phrase <= `market[p]` says that, for each product `p`, the value given to `Make[p]` may not exceed the value of the parameter `market[p]`.

In general, either >= or <= may be followed by any arithmetic expression in previously defined sets and parameters and currently defined dummy indices. Most linear programs are formulated in such a way that every variable must be nonnegative; an AMPL variable declaration can specify nonnegativity either directly by `>= 0`, or indirectly as in the diet model of Figure 5-1:

```
param f_min {FOOD} >= 0;
param f_max {j in FOOD} >= f_min[j];

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

The values following >= and <= are lower and upper *bounds* on the variables. Because these bounds represent a kind of constraint, they could just as well be imposed by the constraint declarations described later in this chapter. By placing bounds in the `var` declaration instead, you may be able to make the model shorter or clearer, although you will not make the optimal solution any different or easier to find. Some solvers do treat bounds specially in order to speed up their algorithms, but with AMPL all bounds are identified automatically, no matter how they are expressed in your model.

Variable declarations may not use the comparison operators <, > or <> in qualifying phrases. For linear programming it makes no sense to constrain a variable to be, say, < 3, since it could always be chosen as 2.99999... or as close to 3 as you like.

An = phrase in a variable declaration gives rise to a definition, as in a parameter declaration. Because a variable is being declared, however, the expression to the right of the = operator may contain previously declared variables as well as sets and parameters. For example, instead of writing the complicated objective from the multi-period production model of Figure 6-3 (`steelT3.mod`) as

```
maximize Total_Profit:
   sum {p in PROD, t in 1..T}
      (sum {a in AREA[p]} revenue[p,a,t]*Sell[p,a,t] -
         prodcost[p]*Make[p,t] - invcost[p]*Inv[p,t]);
```

you could instead define variables to represent the total revenues, production costs, and inventory costs:

```
var Total_Revenue =
   sum {p in PROD, t in 1..T}
      sum {a in AREA[p]} revenue[p,a,t] * Sell[p,a,t];
var Total_Prod_Cost =
   sum {p in PROD, t in 1..T} prodcost[p] * Make[p,t];
var Total_Inv_Cost =
   sum {p in PROD, t in 1..T} invcost[p] * Inv[p,t];
```

The objective would then be the sum of these three defined variables:

```
maximize Total_Profit:
   Total_Revenue - Total_Prod_Cost - Total_Inv_Cost;
```

The structure of the objective is clearer this way. Also the defined variables are conveniently available to a `display` statement to show how the three main components of profit compare:

```
ampl: display Total_Revenue, Total_Prod_Cost, Total_Inv_Cost;
Total_Revenue = 801385
Total_Prod_Cost = 285643
Total_Inv_Cost = 1221
```

Declarations of defined variables like these do not give rise to additional constraints in the resulting problem instance. Rather, the linear expression to the right of the = is substituted for every occurrence of the defined variable in the objective and constraints. Defined variables are even more useful for nonlinear programming, where the substitution may be only implicit, so we will return to this topic in Chapter 18.

If the expression to the right of the = operator contains no variables, then you are merely defining variables to be fixed to values given by the data. In that case you should use a `param` declaration instead. On the other hand, if you only want to fix some variables temporarily while developing or analyzing a model, then you should leave the declarations unchanged and instead fix them with the `fix` command described in Section 11.4.

A `:=` or `default` phrase in a variable declaration gives *initial* values to the indicated variables. Variables not assigned an initial value by `:=` can also be assigned initial values from a data file. Initial values of variables are normally changed — ideally to optimal values — when a solver is invoked. Thus the main purpose of initial values of variables is to give the solver a good starting solution. Solvers for linear programming can seldom make good use of a starting solution, however, so we defer further discussion of this topic to Chapter 18 on nonlinear programming.

Finally, variables may be declared as `integer` so that they must take whole number values in any optimal solution, or as `binary` so that they may only take the values 0 and

1. Models that contain any such variables are integer programs, which are the topic of Chapter 20.

8.2 Linear expressions

An arithmetic expression is *linear* in a given variable if, for every unit increase or decrease in the variable, the value of the expression increases or decreases by some fixed amount. An expression that is linear in all its variables is called a linear expression. (Strictly speaking, these are *affine* expressions, and a linear expression is an affine expression with constant term zero. For simplicity, we will ignore this distinction.)

AMPL recognizes as a linear expression any sum of terms of the form

constant-expr
variable-ref
(*constant-expr*) * *variable-ref*

provided that each *constant-expr* is an arithmetic expression that contains no variables, while *var-ref* is a reference (possibly subscripted) to a variable. The parentheses around the *constant-expr* may be omitted if the result is the same according to the rules of operator precedence (Table A-1). The following examples, from the constraints in the multiperiod production model of Figure 6-3, are all linear expressions under this definition:

```
avail[t]
Make[p,t] + Inv[p,t-1]
sum {p in PROD} (1/rate[p]) * Make[p,t]
sum {a in AREA[p]} Sell[p,a,t] + Inv[p,t]
```

The model's objective,

```
sum {p in PROD, t in 1..T}
   (sum {a in AREA[p]} revenue[p,a,t] * Sell[p,a,t] -
      prodcost[p] * Make[p,t] - invcost[p] * Inv[p,t])
```

is also linear because subtraction of a term is the addition of its negative, and a sum of sums is itself a sum.

Various kinds of expressions are equivalent to a sum of terms of the forms above, and are also recognized as linear by AMPL. Division by an arithmetic expression is equivalent to multiplication by its inverse, so

```
(1/rate[p]) * Make[p,t]
```

may be written in a linear program as

```
Make[p,t] / rate[p]
```

The order of multiplications is irrelevant, so the *variable-ref* need not come at the end of a term; for instance,

```
revenue[p,a,t] * Sell[p,a,t]
```

is equivalent to

```
Sell[p,a,t] * revenue[p,a,t]
```

As an example combining these principles, imagine that `revenue[p,a,t]` is in dollars per metric ton, while `Sell` remains in tons. If we define conversion factors

```
param mt_t = 0.90718474;   # metric tons per ton
param t_mt = 1 / mt_t;     # tons per metric ton
```

then both

```
sum {a in AREA[p]} mt_t * revenue[p,a,t] * Sell[p,a,t]
```

and

```
sum {a in AREA[p]} revenue[p,a,t] * Sell[p,a,t] / t_mt
```

are linear expressions for total revenue.

To continue our example, if costs are also in dollars per metric ton, the objective could be written as

```
mt_t * sum {p in PROD, t in 1..T}
   (sum {a in AREA[p]} revenue[p,a,t] * Sell[p,a,t] -
      prodcost[p] * Make[p,t] - invcost[p] * Inv[p,t])
```

or as

```
sum {p in PROD, t in 1..T}
   (sum {a in AREA[p]} revenue[p,a,t] * Sell[p,a,t] -
      prodcost[p] * Make[p,t] - invcost[p] * Inv[p,t]) / t_mt
```

Multiplication and division distribute over any summation to yield an equivalent linear sum of terms. Notice that in the first form, `mt_t` multiplies the entire `sum {p in PROD, t in 1..T}`, while in the second `t_mt` divides only the summand that follows `sum {p in PROD, t in 1..T}`, because the `/` operator has higher precedence than the `sum` operator. In these examples the effect is the same, however.

Finally, an `if-then-else` operator produces a linear result if the expressions following `then` and `else` are both linear and no variables appear in the logical expression between `if` and `else`. The following example appeared in a constraint in Section 7.3:

```
Make[j,t] +
   (if t = first(WEEKS) then inv0[j] else Inv[j,prev(t)])
```

The variables in a linear expression may not appear as the operands to any other operators, or in the arguments to any functions. This rule applies to iterated operators like `max`, `min`, `abs`, `forall`, and `exists`, as well as `^` and standard numerical functions like `sqrt`, `log`, and `cos`.

To summarize, a linear expression may be any sum of terms in the forms

```
constant-expr
var-ref
(constant-expr) * (linear-expr)
(linear-expr) * (constant-expr)
(linear-expr) / (constant-expr)
if logical-expr then linear-expr else linear-expr
```

where *constant-expr* is any arithmetic expression that contains no references to variables, and *linear-expr* is any other (simpler) linear expression. Parentheses may be omitted if the result is the same by the rules of operator precedence in Table A-1. AMPL automatically performs the transformations that convert any such expression to a simple sum of linear terms.

8.3 Objectives

The declaration of an objective function consists of one of the keywords `minimize` or `maximize`, a name, a colon, and a linear expression in previously defined sets, parameters and variables. We have seen examples such as

```
minimize Total_Cost: sum {j in FOOD} cost[j] * Buy[j];
```

and

```
maximize Total_Profit:
   sum {p in PROD, t in 1..T}
      (sum {a in AREA[p]} revenue[p,a,t] * Sell[p,a,t] -
         prodcost[p] * Make[p,t] - invcost[p] * Inv[p,t]);
```

The name of the objective plays no further role in the model, with the exception of certain ‘‘columnwise’’ declarations to be introduced in Chapters 15 and 16. Within AMPL commands, the objective’s name refers to its value. Thus for example after solving a feasible instance of the Figure 2-1 diet model we could issue the command

```
ampl: display {j in FOOD} 100 * cost[j] * Buy[j] / Total_Cost;
100*cost[j]*Buy[j]/Total_Cost [*] :=
BEEF  14.4845
 CHK   4.38762
FISH   3.8794
 HAM  24.4792
 MCH  16.0089
 MTL  16.8559
 SPG  15.6862
 TUR   4.21822
;
```

to show the percentage of the total cost spent on each food.

Although a particular linear program must have one objective function, a model may contain more than one objective declaration. Moreover, any `minimize` or `maximize` declaration may define an indexed collection of objective functions, by including an indexing expression after the objective name. In these cases, you may issue an `objective` command, before typing `solve`, to indicate which objective is to be optimized.

As an example, recall that when trying to solve the model of Figure 2-1 with the data of Figure 2-2, we found that no solution could satisfy all of the constraints; we subsequently increased the sodium (`NA`) limit to 50000 to make a feasible solution possible. It is reasonable to ask: How much of an increase in the sodium limit is really necessary to permit a feasible solution? For this purpose we can introduce a new objective equal to the total sodium in the diet:

```
minimize Total_NA: sum {j in FOOD} amt["NA",j] * Buy[j];
```

(We create this objective only for sodium, because we have no reason to minimize most of the other nutrients.) We can solve the linear program for total cost as before, since AMPL chooses the model's first objective by default:

```
ampl: model diet.mod;
ampl: data diet2a.dat;
ampl: display n_max["NA"];
n_max['NA'] = 50000

ampl: minimize Total_NA: sum {j in FOOD} amt["NA",j] * Buy[j];
ampl: solve;
MINOS 5.5: optimal solution found.
13 iterations, objective 118.0594032
Objective = Total_Cost
```

The solver tells us the minimum cost, and we can also use `display` to look at the total sodium, even though it's not currently being minimized:

```
ampl: display Total_NA;
Total_NA = 50000
```

Next we can use the `objective` command to switch the objective to minimization of total sodium. The `solve` command then re-optimizes with this alternative objective, and we display `Total_Cost` to determine the resulting cost:

```
ampl: objective Total_NA;

ampl: solve;
MINOS 5.5: optimal solution found.
1 iterations, objective 48186

ampl: display Total_Cost;
Total_Cost = 123.627
```

We see that sodium can be brought down by about 1800, though the cost is forced up by about $5.50 as a result. (Healthier diets are in general more expensive, because they force the solution away from the one that minimizes costs.)

As another example, here's how we could experiment with different optimal solutions for the office assignment problem of Figure 3-2. First we solve the original problem:

```
ampl: model transp.mod; data assign.dat; solve;
CPLEX 8.0.0: optimal solution; objective 28
24 dual simplex iterations (0 in phase I)

ampl: option display_1col 1000, omit_zero_rows 1;
ampl: option display_eps .000001;

ampl: display Total_Cost,
ampl?    {i in ORIG, j in DEST} cost[i,j] * Trans[i,j];
Total_Cost = 28

cost[i,j]*Trans[i,j] :=
Coullard  C118   6
Daskin    D241   4
Hazen     C246   1
Hopp      D237   1
Iravani   C138   2
Linetsky  C250   3
Mehrotra  D239   2
Nelson    C140   4
Smilowitz M233   1
Tamhane   C251   3
White     M239   1
;
```

To keep the objective value at this optimal level while we experiment, we add a constraint that fixes the expression for the objective equal to the current value, 28:

```
ampl: subject to Stay_Optimal:
ampl?    sum {i in ORIG, j in DEST}
ampl?       cost[i,j] * Trans[i,j] = 28;
```

Next, recall that `cost[i,j]` is the ranking that person `i` has given to office `j`, while `Trans[i,j]` is set to 1 if it's optimal to put person `i` in office `j`, or 0 otherwise. Thus

```
sum {j in DEST} cost[i,j] * Trans[i,j]
```

always equals the ranking of person `i` for the office to which `i` is assigned. We use this expression to declare a new objective function:

```
ampl: minimize Pref_of {i in ORIG}:
ampl?    sum {j in DEST} cost[i,j] * Trans[i,j];
```

This statement creates, for each person `i`, an objective `Pref_of[i]` that minimizes the ranking of `i` for the room that `i` is assigned. Then we can select any one person and optimize his or her ranking in the assignment:

```
ampl: objective Pref_of["Coullard"];
ampl: solve;
CPLEX 8.0.0: optimal solution; objective 3
3 simplex iterations (0 in phase I)
```

Looking at the new assignment, we see that the original objective is unchanged, and that the selected individual's situation is in fact improved, although of course at the expense of others:

```
ampl: display Total_Cost,
ampl?    {i in ORIG, j in DEST} cost[i,j] * Trans[i,j];
Total_Cost = 28

cost[i,j]*Trans[i,j] :=
Coullard  D241   3
Daskin    D237   1
Hazen     C246   1
Hopp      C251   5
Iravani   C138   2
Linetsky  C250   3
Mehrotra  D239   2
Nelson    C140   4
Smilowitz M233   1
Tamhane   C118   5
White     M239   1
;
```

We were able to make this change because there are several optimal solutions to the original total-ranking objective. A solver arbitrarily returns one of these, but by use of a second objective we can force it toward others.

8.4 Constraints

The simplest kind of constraint declaration begins with the keywords `subject to`, a name, and a colon. Even the `subject to` is optional; AMPL assumes that any declaration not beginning with a keyword is a constraint. Following the colon is an algebraic description of the constraint, in terms of previously defined sets, parameters and variables. Thus in the production model introduced in Figure 1-4, we have the following constraint imposed by limited processing time:

```
subject to Time:
   sum {p in PROD} (1/rate[p]) * Make[p] <= avail;
```

The name of a constraint, like the name of an objective, is not used anywhere else in an algebraic model, though it figures in alternative ‘‘columnwise’’ formulations (Chapter 16) and is used in the AMPL command environment to specify the constraint’s dual value and other associated quantities (Chapter 14).

Most of the constraints in large linear programming models are defined as indexed collections, by giving an indexing expression after the constraint name. The constraint `Time`, for example, is generalized in subsequent examples to say that the production time may not exceed the time available in each processing stage `s` (Figure 1-6a):

```
subject to Time {s in STAGE}:
   sum {p in PROD} (1/rate[p,s]) * Make[p] <= avail[s];
```

or in each week `t` (Figure 4-4):

```
subject to Time {t in 1..T}:
   sum {p in PROD} (1/rate[p]) * Make[p,t] <= avail[t];
```

Another constraint from the latter example says that production, sales and inventories must balance for each product p in each week t:

```
subject to Balance {p in PROD, t in 1..T}:
   Make[p,t] + Inv[p,t-1] = Sell[p,t] + Inv[p,t];
```

A constraint declaration can specify any valid indexing expression, which defines a set (as explained in Chapters 5 and 6); there is one constraint for each member of this set. The constraint name can be subscripted, so that Time[1] or Balance[p,t+1] refers to a particular constraint from an indexed collection.

The indexing expression in a constraint declaration should specify a dummy index (like s, t and p in the preceding examples) for each dimension of the indexing set. Then when the constraint corresponding to a particular indexing-set member is processed by AMPL, the dummy indices take their values from that member. This use of dummy indices is what permits a single constraint expression to represent many constraints; the indexing expression is AMPL's translation of a phrase such as ''for all products *p* and weeks $t = 1$ to *T*'' that might be seen in an algebraic statement of the model.

By using more complex indexing expressions, you can specify more precisely the constraints to be included in a model. Consider, for example, the following variation on the production time constraint:

```
subject to Time {t in 1..T: avail[t] > 0}:
   sum {p in PROD} (1/rate[p]) * Make[p,t] <= avail[t];
```

This says that if avail[t] is specified as zero in the data for any week t, it is to be interpreted as meaning ''no constraint on time available in week t'' rather than ''limit of zero on time available in week t''. In the simpler case where there is just one Time constraint not indexed over weeks, you can specify an analogous conditional definition as follows:

```
subject to Time {if avail > 0}:
   sum {p in PROD} (1/rate[p]) * Make[p] <= avail;
```

The pseudo-indexing expression {if avail > 0} causes one constraint, named Time, to be generated if the condition avail > 0 is true, and no constraint at all to be generated if the condition is false. (The same notation can be used to conditionally define other model components.)

AMPL's algebraic description of a constraint may consist of any two linear expressions separated by an equality or inequality operator:

linear-expr <= *linear-expr*
linear-expr = *linear-expr*
linear-expr >= *linear-expr*

While it is customary in mathematical descriptions of linear programming to place all terms containing variables to the left of the operator and all other terms to the right (as in constraint Time), AMPL imposes no such requirement (as seen in constraint Balance).

Convenience and readability should determine what terms you place on each side of the operator. AMPL takes care of canonicalizing constraints, such as by combining linear terms involving the same variable and moving variables from one side of a constraint to the other. The `expand` command described in Section 1.4 shows the canonical forms of the constraints.

AMPL also allows double-inequality constraints such as the following from the diet model of Figure 2-1:

```
subject to Diet {i in NUTR}:
   n_min[i] <= sum {j in FOOD} amt[i,j] * Buy[j] <= n_max[i];
```

This says that the middle expression, the amount of nutrient `i` supplied by all foods, must be greater than or equal to `n_min[i]` and also less than or equal to `n_max[i]`. The permissible forms for a constraint of this kind are

const-expr `<=` *linear-expr* `<=` *const-expr*
const-expr `>=` *linear-expr* `>=` *const-expr*

where each *const-expr* must contain no variables. The effect is to give upper and lower bounds on the value of the *linear-expr*. If your model requires variables in the left-hand or right-hand *const-expr*, you must define two different constraints in separate declarations.

For most applications of linear programming, you need not worry about the form of the constraints. If you simply write the constraints in the most convenient way, they will be recognized as proper linear constraints according to the rules in this chapter. There do exist situations, however, in which your choice of formulation will determine whether AMPL recognizes your model as linear. Imagine that we want to further constrain the production model so that no product `p` may represent more than a certain fraction of total production. We define a parameter `max_frac` to represent the limiting fraction; the constraint then says that production of `p` divided by total production must be less than or equal to `max_frac`:

```
subject to Limit {p in PROD}:
   Make[p] / sum {q in PROD} Make[q] <= max_frac;
```

This is not a linear constraint to AMPL, because its left-hand expression contains a division by a sum of variables. But if we rewrite it as

```
subject to Limit {p in PROD}:
   Make[p] <= max_frac * sum {q in PROD} Make[q];
```

then AMPL does recognize it as linear.

AMPL simplifies constraints as it prepares the model and data for handing to a solver. For example, it may eliminate variables fixed at a value, combine single-variable constraints with the simple bounds on the variables, or drop constraints that are implied by other constraints. You can normally ignore this presolve phase, but there are ways to observe its effects and modify its actions, as explained in Section 14.1.

Exercises

8-1. In the diet model of Figure 5-1, add a := phrase to the `var` declaration (as explained in Section 8.1) to initialize each variable to a value midway between its lower and upper bounds.

Read this model into AMPL along with the data from Figure 5-2. Using `display` commands, determine which constraints (if any) the initial solution fails to satisfy, and what total cost this solution gives. Is the total cost more or less than the optimal total cost?

8-2. This exercise asks you to reformulate various kinds of constraints to make them linear.

(a) The following constraint says that the inventory `Inv[p,t]` for product `p` in any period `t` must not exceed the smallest one-period production `Make[p,t]` of product `p`:

```
subject to Inv_Limit {p in PROD, t in 1..T}:
   Inv[p,t] <= min {tt in 1..T} Make[p,tt];
```

This constraint is not recognized as linear by AMPL, because it applies the `min` operator to variables. Formulate a linear constraint that has the same effect.

(b) The following constraint says that the change in total inventories from one period to the next may not exceed a certain parameter `max_change`:

```
subject to Max_Change {t in 1..T}:
   abs(sum {p in PROD} Inv[p,t-1] - sum {p in PROD} Inv[p,t])
      <= max_change;
```

This constraint is not linear because it applies the `abs` function to an expression involving variables. Formulate a linear constraint that has the same effect.

(c) The following constraint says that the ratio of total production to total inventory in a period may not exceed `max_inv_ratio`:

```
subject to Max_Inv_Ratio {t in 1..T}:
   (sum {p in PROD} Inv[p,t]) / (sum {p in PROD} Make[p,t])
      <= max_inv_ratio;
```

This constraint is not linear because it divides one sum of variables by another. Formulate a linear constraint that has the same effect.

(d) What can you say about formulation of an alternative linear constraint for the following cases?

– In (a), `min` is replaced by `max`.

– In (b), `<= max_change` is replaced by `>= min_change`.

– In (c), the parameter `max_inv_ratio` is replaced by a new variable, `Ratio[t]`.

8-3. This exercise deals with some more possibilities for using more than one objective function in a diet model. Here we consider the model of Figure 5-1, together with the data from Figure 5-2. Suppose that the costs are indexed over stores as well as foods:

```
set STORE:
param cost {STORE,FOOD} > 0;
```

A separate objective function may then be defined for each store:

```
minimize Total_Cost {s in STORE}:
   sum {j in FOOD} cost[s,j] * Buy[j];
```

Consider the following data for three stores:

```
set STORE := "A&P" JEWEL VONS ;

param cost:  BEEF   CHK  FISH   HAM   MCH   MTL   SPG   TUR :=
     "A&P"   3.19  2.59  2.29  2.89  1.89  1.99  1.99  2.49
     JEWEL   3.09  2.79  2.29  2.59  1.59  1.99  2.09  2.30
      VONS   2.59  2.99  2.49  2.69  1.99  2.29  2.00  2.69 ;
```

Using the `objective` command, find the lowest-cost diet for each store. Which store offers the lowest total cost?

Consider now an additional objective that represents total packages purchased, regardless of cost:

```
minimize Total_Number:
   sum {j in FOOD} Buy[j];
```

What is the minimum value of this objective? What are the costs at the three stores when this objective is minimized? Explain why you would expect these costs to be higher than the costs computed in (a).

8-4. This exercise relates to the assignment example of Section 8.3.

(a) What is the best-ranking office that you can assign to each individual, given that the total of the rankings must stay at the optimal value of 28? How many different optimal assignments do there seem to be, and which individuals get different offices in different assignments?

(b) Modify the assignment example so that it will find the best-ranking office that you can assign to each individual, given that the total of the rankings may increase from 28, but may not exceed 30.

(c) After making the modification suggested in (b), the person in charge of assigning offices has tried again to minimize the objective `Pref_of["Coullard"]`. This time, the reported solution is as follows:

```
ampl: display Total_Cost,
ampl?   {i in ORIG, j in DEST} cost[i,j]*Trans[i,j];
Total_Cost = 30

cost[i,j]*Trans[i,j] :=
Coullard  M239   1
Daskin    D241   4
Hazen     C246   1
Hopp      C251   2.5
Hopp      D237   0.5
Iravani   C138   2
Linetsky  C250   3
Mehrotra  D239   2
Nelson    C140   4
Smilowitz M233   1
Tamhane   C118   5
White     C251   2.5
White     D237   1.5
;
```

Coullard is now assigned her first choice, but what is the difficulty with the overall solution? Why doesn't it give a useful resolution to the assignment problem as we have stated it?

8-5. Return to the assignment version of the transportation model, in Figures 3-1a and 3-2.

(a) Add parameters `worst[i]` for each `i` in `ORIG`, and constraints saying that `Trans[i,j]` must equal 0 for every combination of `i` in `ORIG` and `j` in `DEST` such that `cost[i,j]` is greater than `worst[i]`. (See the constraint `Time` in Section 8.4 for a similar example.) In the assignment interpretation of this model, what do the new constraints mean?

(b) Use the model from (a) to show that there is an optimal solution, with the objective equal to 28, in which no one gets an office worse than their fifth choice.

(c) Use the model from (a) to show that at least one person must get an office worse than fourth choice.

(d) Use the model from (a) to show that if you give Nelson his first choice, without any restrictions on the other individuals' choices, the objective cannot be made smaller than 31. Determine similarly how small the objective can be made if each other individual is given first choice.