6

Compound Sets and Indexing

Most linear programming models involve indexing over combinations of members from several different sets. Indeed, the interaction of indexing sets is often the most complicated aspect of a model; once you have the arrangement of sets worked out, the rest of the model can be written clearly and concisely.

All but the simplest models employ compound sets whose members are pairs, triples, quadruples, or even longer ‘‘tuples’’ of objects. This chapter begins with the declaration and use of sets of ordered pairs. We concentrate first on the set of all pairs from two sets, then move on to subsets of all pairs and to ‘‘slices’’ through sets of pairs. Subsequent sections explore sets of longer tuples, and extensions of AMPL’s set operators and indexing expressions to sets of tuples.

The final section of this chapter introduces sets that are declared in collections indexed over other sets. An indexed collection of sets often plays much the same role as a set of tuples, but it represents a somewhat different way of thinking about the formulation. Each kind of set is appropriate in certain situations, and we offer some guidelines for choosing between them.

6.1 Sets of ordered pairs

An ordered pair of objects, whether numbers or strings, is written with the objects separated by a comma and surrounded by parentheses:

```
("PITT","STL")
("bands",5)
(3,101)
```

As the term ‘‘ordered’’ suggests, it makes a difference which object comes first; `("STL","PITT")` is not the same as `("PITT","STL")`. The same object may appear both first and second, as in `("PITT","PITT")`.

Pairs can be collected into sets, just like single objects. A comma-separated list of pairs may be enclosed in braces to denote a literal set of ordered pairs:

```
{("PITT","STL"),("PITT","FRE"),("PITT","DET"),("CLEV","FRE")}
{(1,1),(1,2),(1,3),(2,1),(2,2),(2,3),(3,1),(3,2),(3,3)}
```

Because sets of ordered pairs are often large and subject to change, however, they seldom appear explicitly in AMPL models. Instead they are described symbolically in a variety of ways.

The set of all ordered pairs from two given sets appears frequently in our examples. In the transportation model of Figure 3-1a, for instance, the set of all origin-destination pairs is written as either of

```
{ORIG, DEST}
{i in ORIG, j in DEST}
```

depending on whether the context requires dummy indices `i` and `j`. The multiperiod production model of Figure 4-4 uses a set of all pairs from a set of strings (representing products) and a set of numbers (representing weeks):

```
{PROD, 1..T}
{p in PROD, t in 1..T}
```

Various collections of model components, such as the parameter `revenue` and the variable `Sell`, are indexed over this set. When individual components are referenced in the model, they must have two subscripts, as in `revenue[p,t]` or `Sell[p,t]`. The order of the subscripts is always the same as the order of the objects in the pairs; in this case the first subscript must refer to a string in `PROD`, and the second to a number in `1..T`.

An indexing expression like `{p in PROD, t in 1..T}` is the AMPL transcription of a phrase like ''for all p in P, $t = 1, \ldots, T$'' from algebraic notation. There is no compelling reason to think in terms of ordered pairs in this case, and indeed we did not mention ordered pairs when introducing the multiperiod production model in Chapter 4. On the other hand, we can modify the transportation model of Figure 3-1a to emphasize the role of origin-destination pairs as ''links'' between cities, by defining this set of pairs explicitly:

```
set LINKS = {ORIG,DEST};
```

The shipment costs and amounts can then be indexed over links:

```
param cost {LINKS} >= 0;
var Trans {LINKS} >= 0;
```

In the objective, the sum of costs over all shipments can be written like this:

```
minimize Total_Cost:
   sum {(i,j) in LINKS} cost[i,j] * Trans[i,j];
```

Notice that when dummy indices run over a set of pairs like `LINKS`, they must be defined in a pair like `(i,j)`. It would be an error to sum over `{k in LINKS}`. The complete model is shown in Figure 6-1, and should be compared with Figure 3-1a. The specification of the data could be the same as in Figure 3-1b.

```
set ORIG;   # origins
set DEST;   # destinations

set LINKS = {ORIG,DEST};

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

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

param cost {LINKS} >= 0;   # shipment costs per unit
var Trans {LINKS} >= 0;    # units to be shipped

minimize Total_Cost:
   sum {(i,j) in LINKS} cost[i,j] * Trans[i,j];

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

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

Figure 6-1: Transportation model with all pairs (`transp2.mod`).

6.2 Subsets and slices of ordered pairs

In many applications, we are concerned only with a subset of all ordered pairs from two sets. For example, in the transportation model, shipments may not be possible from every origin to every destination. The shipping costs per unit may be provided only for the usable origin-destination pairs, so that it is desirable to index the costs and the variables only over these pairs. In AMPL terms, we want the set `LINKS` defined above to contain just a subset of pairs that are given in the data, rather than all pairs from `ORIG` and `DEST`.

It is not sufficient to declare `set LINKS`, because that declares only a set of single members. At a minimum, we need to say

```
set LINKS dimen 2;
```

to indicate that the data must consist of members of ‘‘dimension’’ two — that is, pairs. Better yet, we can say that `LINKS` is a subset of the set of all pairs from `ORIG` and `DEST`:

```
set LINKS within {ORIG,DEST};
```

This has the advantage of making the model’s intent clearer; it also helps catch errors in the data. The subsequent declarations of parameter `cost`, variable `Trans`, and the objective function remain the same as they are in Figure 6-1. But the components `cost[i,j]` and `Trans[i,j]` will now be defined only for those pairs given in the data as members of `LINKS`, and the expression

```
sum {(i,j) in LINKS} cost[i,j] * Trans[i,j]
```

will represent a sum over the specified pairs only.

How are the constraints written? In the original transportation model, the supply limit constraint was:

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

This does not work when LINKS is a subset of pairs, because for each i in ORIG it tries to sum Trans[i,j] over every j in DEST, while Trans[i,j] is defined only for pairs (i,j) in LINKS. If we try it, we get an error message like this:

```
error processing constraint Supply['GARY']:
        invalid subscript Trans['GARY','FRA']
```

What we want to say is that for each origin i, the sum should be over all destinations j such that (i,j) is an allowed link. This statement can be transcribed directly to AMPL, by adding a condition to the indexing expression after sum:

```
subject to Supply {i in ORIG}:
   sum {j in DEST: (i,j) in LINKS} Trans[i,j] = supply[i];
```

Rather than requiring this somewhat awkward form, however, AMPL lets us drop the j in DEST from the indexing expression to produce the following more concise constraint:

```
subject to Supply {i in ORIG}:
   sum {(i,j) in LINKS} Trans[i,j] = supply[i];
```

Because {(i,j) in LINKS} appears in a context where i has already been defined, AMPL interprets this indexing expression as the set of all j such that (i,j) is in LINKS. The demand constraint is handled similarly, and the entire revised version of the model is shown in Figure 6-2a. A small representative collection of data for this model is shown in Figure 6-2b; AMPL offers a variety of convenient ways to specify the membership of compound sets and the data indexed over them, as explained in Chapter 9.

You can see from Figure 6-2a that the indexing expression

```
{(i,j) in LINKS}
```

means something different in each of the three places where it appears. Its membership can be understood in terms of a table like this:

```
        FRA  DET  LAN  WIN  STL  FRE  LAF
GARY          x    x          x          x
CLEV     x    x    x    x     x          x
PITT     x               x    x    x
```

The rows represent origins and the columns destinations, while each pair in the set is marked by an x. A table for {ORIG,DEST} would be completely filled in with x's, while the table shown depicts {LINKS} for the ''sparse'' subset of pairs defined by the data in Figure 6-2b.

At a point where i and j are not currently defined, such as in the objective

```
minimize Total_Cost:
   sum {(i,j) in LINKS} cost[i,j] * Trans[i,j];
```

```
set ORIG;   # origins
set DEST;   # destinations

set LINKS within {ORIG,DEST};

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

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

param cost {LINKS} >= 0;   # shipment costs per unit
var Trans {LINKS} >= 0;    # units to be shipped

minimize Total_Cost:
   sum {(i,j) in LINKS} cost[i,j] * Trans[i,j];

subject to Supply {i in ORIG}:
   sum {(i,j) in LINKS} Trans[i,j] = supply[i];

subject to Demand {j in DEST}:
   sum {(i,j) in LINKS} Trans[i,j] = demand[j];
```

Figure 6-2a: Transportation model with selected pairs (`transp3.mod`).

```
param: ORIG:  supply :=
    GARY  1400    CLEV  2600    PITT  2900 ;

param: DEST:  demand :=
    FRA   900   DET  1200   LAN   600    WIN  400
    STL  1700   FRE  1100   LAF  1000 ;

param: LINKS:  cost :=
    GARY DET 14   GARY LAN 11   GARY STL 16   GARY LAF  8
    CLEV FRA 27   CLEV DET  9   CLEV LAN 12   CLEV WIN  9
    CLEV STL 26   CLEV LAF 17
    PITT FRA 24   PITT WIN 13   PITT STL 28   PITT FRE 99 ;
```

Figure 6-2b: Data for transportation model (`transp3.dat`).

the indexing expression `{(i,j) in LINKS}` represents all the pairs in this table. But at a point where `i` has already been defined, such as in the `Supply` constraint

```
subject to Supply {i in ORIG}:
   sum {(i,j) in LINKS} Trans[i,j] = supply[i];
```

the expression `{(i,j) in LINKS}` is associated with just the row of the table corresponding to `i`. You can think of it as taking a one-dimensional ‘‘slice’’ through the table in the row corresponding to the already-defined first component. Although in this case the first component is a previously defined dummy index, the same convention applies when the first component is any expression that can be evaluated to a valid set object; we could write

```
{("GARY",j) in LINKS}
```

for example, to represent the pairs in the first row of the table.

Similarly, where `j` has already been defined, such as in the `Demand` constraint

```
subject to Demand {j in DEST}:
   sum {(i,j) in LINKS} Trans[i,j] = demand[j];
```

the expression `{(i,j) in LINKS}` selects pairs from the column of the table corresponding to `j`. Pairs in the third column of the table could be specified by `{(i,"LAN") in LINKS}`.

6.3 Sets of longer tuples

AMPL's notation for ordered pairs extends in a natural way to triples, quadruples, or ordered lists of any length. All tuples in a set must have the same dimension. A set can't contain both pairs and triples, for example, nor can the determination as to whether a set contains pairs or triples be made according to some value in the data.

The multicommodity transportation model of Figure 4-1 offers some examples of how we can use ordered triples, and by extension longer tuples. In the original version of the model, the costs and amounts shipped are indexed over origin-destination-product triples:

```
param cost {ORIG,DEST,PROD} >= 0;
var Trans {ORIG,DEST,PROD} >= 0;
```

In the objective, `cost` and `Trans` are written with three subscripts, and the total cost is determined by summing over all triples:

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

The indexing expressions are the same as before, except that they list three sets instead of two. An indexing expression that listed *k* sets would similarly denote a set of *k*-tuples.

If instead we define `LINKS` as we did in Figure 6-2a, the multicommodity declarations come out like this:

```
set LINKS within {ORIG,DEST};

param cost {LINKS,PROD} >= 0;
var Trans {LINKS,PROD} >= 0;

minimize Total_Cost:
   sum {(i,j) in LINKS, p in PROD} cost[i,j,p] * Trans[i,j,p];
```

Here we see how a set of triples can be specified as combinations from a set of pairs (`LINKS`) and a set of single members (`PROD`). Since `cost` and `Trans` are indexed over `{LINKS,PROD}`, their first two subscripts must come from a pair in `LINKS`, and their third subscript from a member of `PROD`. Sets of longer tuples can be built up in an analogous way.

As a final possibility, it may be that only certain combinations of origins, destinations, and products are workable. Then it makes sense to define a set that contains only the triples of allowed combinations:

```
set ROUTES within {ORIG,DEST,PROD};
```

The costs and amounts shipped are indexed over this set:

```
param cost {ROUTES} >= 0;
var Trans {ROUTES} >= 0;
```

and in the objective, the total cost is a sum over all triples in this set:

```
minimize Total_Cost:
   sum {(i,j,p) in ROUTES} cost[i,j,p] * Trans[i,j,p];
```

Individual triples are written, by analogy with pairs, as a parenthesized and comma-separated list `(i,j,p)`. Longer lists specify longer tuples.

In the three constraints of this model, the summations must be taken over three different slices through the set `ROUTES`:

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

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

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

In the `Supply` constraint, for instance, indices `i` and `p` are defined before the `sum`, so `{(i,j,p) in ROUTES}` refers to all `j` such that `(i,j,p)` is a triple in `ROUTES`. AMPL allows comparable slices through any set of tuples, in any number of dimensions and any combination of coordinates.

When you declare a high-dimensional set such as `ROUTES`, a phrase like `within {ORIG,DEST,PROD}` may specify a set with a huge number of members. With 10 origins, 100 destinations and 100 products, for instance, this set potentially has 100,000 members. Fortunately, AMPL does not create this set when it processes the declaration, but merely checks that each tuple in the data for `ROUTES` has its first component in `ORIG`, its second in `DEST`, and its third in `PROD`. The set `ROUTES` can thus be handled efficiently so long as it does not itself contain a huge number of triples.

When using high-dimensional sets in other contexts, you may have to be more careful that you do not inadvertently force AMPL to generate a large set of tuples. As an example, consider how you could constrain the volume of all products shipped out of each origin to be less than some amount. You might write either

```
subject to Supply_All {i in ORIG}:
   sum {j in DEST, p in PROD: (i,j,p) in ROUTES}
      Trans[i,j,p] <= supply_all[i];
```

or, using the more compact slice notation,

```
subject to Supply_All {i in ORIG}:
   sum {(i,j,p) in ROUTES} Trans[i,j,p] <= supply_all[i];
```

In the first case, AMPL explicitly generates the set `{j in DEST, p in PROD}` and checks for membership of `(i,j,p)` in `ROUTES`, while in the second case it is able to use a more efficient approach to finding all `(i,j,p)` from `ROUTES` that have a given `i`. In our small examples this may not seem critical, but for problems of realistic size the slice version may be the only one that can be processed in a reasonable amount of time and space.

6.4 Operations on sets of tuples

Operations on compound sets are, as much as possible, the same as the operations introduced for simple sets in Chapter 5. Sets of pairs, triples, or longer tuples can be combined with `union`, `inter`, `diff`, and `symdiff`; can be tested by `in` and `within`; and can be counted with `card`. Dimensions of operands must match appropriately, so for example you may not form the union of a set of pairs with a set of triples. Also, compound sets in AMPL cannot be declared as `ordered` or `circular`, and hence also cannot be arguments to functions like `first` and `next` that take only ordered sets.

Another set operator, `cross`, gives the set of all pairs of its arguments — the cross or Cartesian product. Thus the set expression

```
ORIG cross DEST
```

represents the same set as the indexing expression `{ORIG,DEST}`, and

```
ORIG cross DEST cross PROD
```

is the same as `{ORIG,DEST,PROD}`.

Our examples so far have been constructed so that every compound set has a domain within a cross product of previously specified simple sets; `LINKS` lies within `ORIG cross DEST`, for example, and `ROUTES` within `ORIG cross DEST cross PROD`. This practice helps to produce clear and correct models. Nevertheless, if you find it inconvenient to specify the domains as part of the data, you may define them instead within the model. AMPL provides an iterated `setof` operator for this purpose, as in the following example:

```
set ROUTES dimen 3;

set PROD = setof {(i,j,p) in ROUTES} p;
set LINKS = setof {(i,j,p) in ROUTES} (i,j);
```

Like an iterated `sum` operator, `setof` is followed by an indexing expression and an argument, which can be any expression that evaluates to a legal set member. The argument is evaluated for each member of the indexing set, and the results are combined into a new set that is returned by the operator. Duplicate members are ignored. Thus these

expressions for PROD and LINKS give the sets of all objects p and pairs (i,j) such that there is some member (i,j,p) in ROUTES.

As with simple sets, membership in a compound set may be restricted by a logical condition at the end of an indexing expression. For example, the multicommodity transportation model could define

```
set DEMAND = {j in DEST, p in PROD: demand[j,p] > 0};
```

so that DEMAND contains only those pairs (j,p) with positive demand for product p at destination j. As another example, suppose that we also wanted to model transfers of the products from one origin to another. We could simply define

```
set TRANSF = {ORIG,ORIG};
```

to specify the set of all pairs of members from ORIG. But this set would include pairs like ("PITT","PITT"); to specify the set of all pairs of *different* members from ORIG, a condition must be added:

```
set TRANSF = {i1 in ORIG, i2 in ORIG: i1 <> i2};
```

This is another case where two different dummy indices, i1 and i2, need to be defined to run over the same set; the condition selects those pairs where i1 is not equal to i2.

If a set is ordered, the condition within an indexing expression can also refer to the ordering. We could declare

```
set ORIG ordered;
set TRANSF = {i1 in ORIG, i2 in ORIG: ord(i1) < ord(i2)};
```

to define a ''triangular'' set of pairs from ORIG that does not contain any pair and its reverse. For example, TRANSF would contain either of the pairs ("PITT","CLEV") or ("CLEV","PITT"), depending on which came first in ORIG, but it would not contain both.

Sets of numbers can be treated in a similar way, since they are naturally ordered. Suppose that we want to accommodate inventories of different ages in the multiperiod production model of Figure 4-4, by declaring:

```
set PROD;     # products
param T > 0;  # number of weeks
param A > 0;  # maximum age of inventory

var Inv {PROD,0..T,0..A} >= 0;       # tons inventoried
```

Depending on how initial inventories are handled, we might have to include a constraint that no inventory in period t can be more than t weeks old:

```
subject to Too_Old
   {p in PROD, t in 1..T, a in 1..A: a > t}: Inv[p,t,a] = 0;
```

In this case, there is a simpler way to write the indexing expression:

```
subject to Too_Old
   {p in PROD, t in 1..T, a in t+1..A}: Inv[p,t,a] = 0;
```

Here the dummy index defined by `t in 1..T` is immediately used in the phrase `a in t+1..A`. In this and other cases where an indexing expression specifies two or more sets, the comma-separated phrases are evaluated from left to right. Any dummy index defined in one phrase is available for use in all subsequent phrases.

6.5 Indexed collections of sets

Although declarations of individual sets are most common in AMPL models, sets may also be declared in collections indexed over other sets. The principles are much the same as for indexed collections of parameters, variables or constraints.

As an example of how indexed collections of sets can be useful, let us extend the multiperiod production model of Figure 4-4 to recognize different market areas for each product. We begin by declaring:

```
set PROD;
set AREA {PROD};
```

This says that for each member `p` of `PROD`, there is to be a set `AREA[p]`; its members will denote the market areas in which product `p` is sold.

The market demands, expected sales revenues and amounts to be sold should be indexed over areas as well as products and weeks:

```
param market {p in PROD, AREA[p], 1..T} >= 0;
param revenue {p in PROD, AREA[p], 1..T} >= 0;
var Sell {p in PROD, a in AREA[p], t in 1..T}
                            >= 0, <= market[p,a,t];
```

In the declarations for `market` and `revenue`, we define only the dummy index `p` that is needed to specify the set `AREA[p]`, but for the `Sell` variables we need to define all the dummy indices, so that they can be used to specify the upper bound `market[p,a,t]`. This is another example in which an index defined by one phrase of an indexing expression is used by a subsequent phrase; for each `p` from the set `PROD`, `a` runs over a different set `AREA[p]`.

In the objective, the expression `revenue[p,t] * Sell[p,t]` from Figure 4-4 must be replaced by a sum of revenues over all areas for product `p`:

```
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 only other change is in the `Balance` constraints, where `Sell[p,t]` is similarly replaced by a summation:

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

```
set PROD;          # products
set AREA {PROD};   # market areas for each product
param T > 0;       # number of weeks

param rate {PROD} > 0;           # tons per hour produced
param inv0 {PROD} >= 0;          # initial inventory
param avail {1..T} >= 0;         # hours available in week
param market {p in PROD, AREA[p], 1..T} >= 0;
                                 # limit on tons sold in week

param prodcost {PROD} >= 0;      # cost per ton produced
param invcost {PROD} >= 0;       # carrying cost/ton of inventory
param revenue {p in PROD, AREA[p], 1..T} >= 0;
                                 # revenue per ton sold

var Make {PROD,1..T} >= 0;       # tons produced
var Inv {PROD,0..T} >= 0;        # tons inventoried
var Sell {p in PROD, a in AREA[p], t in 1..T}   # tons sold
                     >= 0, <= market[p,a,t];

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]);

           # Total revenue less costs for all products in all weeks

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

           # Total of hours used by all products
           # may not exceed hours available, in each week

subject to Init_Inv {p in PROD}:  Inv[p,0] = inv0[p];

           # Initial inventory must equal given value

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

           # Tons produced and taken from inventory
           # must equal tons sold and put into inventory
```

Figure 6-3: Multiperiod production with indexed sets (`steelT3.mod`).

The complete model is shown in Figure 6-3.

In the data for this model, each set within the indexed collection `AREA` is specified like an ordinary set:

```
set PROD := bands coils;
set AREA[bands] := east north ;
set AREA[coils] := east west export ;
```

The parameters `revenue` and `market` are now indexed over three sets, so their data values are specified in a series of tables. Since the indexing is over a different set

```
param T := 4;

set PROD := bands coils;
set AREA[bands] := east north ;
set AREA[coils] := east west export ;

param avail :=  1 40  2 40  3 32  4 40 ;

param rate :=  bands  200  coils  140 ;
param inv0 :=  bands   10  coils    0 ;

param prodcost :=  bands 10    coils 11 ;
param invcost  :=  bands  2.5  coils  3 ;

param revenue :=

  [bands,*,*]:   1      2      3      4 :=
      east     25.0   26.0   27.0   27.0
      north    26.5   27.5   28.0   28.5

  [coils,*,*]:   1     2     3     4 :=
      east      30    35    37    39
      west      29    32    33    35
      export    25    25    25    28 ;

param market :=

  [bands,*,*]:     1     2     3     4 :=
       east     2000  2000  1500  2000
       north    4000  4000  2500  4500

  [coils,*,*]:     1     2     3     4 :=
       east     1000   800  1000  1100
       west     2000  1200  2000  2300
       export   1000   500   500   800 ;
```

Figure 6-4: Data for multiperiod production with indexed sets (`steelT3.dat`).

`AREA[p]` for each product `p`, the values are most conveniently arranged as one table for each product, as shown in Figure 6-4. (Chapter 9 explains the general rules behind this arrangement.)

We could instead have written this model with a set `PRODAREA` of pairs, such that product `p` will be sold in area `a` if and only if `(p,a)` is a member of `PRODAREA`. Our formulation in terms of `PROD` and `AREA[p]` seems preferable, however, because it emphasizes the hierarchical relationship between products and areas. Although the model must refer in many places to the set of all areas selling one product, it never refers to the set of all products sold in one area.

As a contrasting example, we can consider how the multicommodity transportation model might use indexed collections of sets. As shown in Figure 6-5, for each product we define a set of origins where that product is supplied, a set of destinations where the product is demanded, and a set of links that represent possible shipments of the product:

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

set orig {PROD} within ORIG;
set dest {PROD} within DEST;
set links {p in PROD} = orig[p] cross dest[p];

param supply {p in PROD, orig[p]} >= 0; # available at origins
param demand {p in PROD, dest[p]} >= 0; # required at destinations
   check {p in PROD}: sum {i in orig[p]} supply[p,i]
                        = sum {j in dest[p]} demand[p,j];

param limit {ORIG,DEST} >= 0;

param cost {p in PROD, links[p]} >= 0;  # shipment costs per unit
var Trans {p in PROD, links[p]} >= 0;   # units to be shipped

minimize Total_Cost:
   sum {p in PROD, (i,j) in links[p]} cost[p,i,j] * Trans[p,i,j];

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

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

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

Figure 6-5: Multicommodity transportation with indexed sets (`multic.mod`).

```
set orig {PROD} within ORIG;
set dest {PROD} within DEST;
set links {p in PROD} = orig[p] cross dest[p];
```

The declaration of `links` demonstrates that it is possible to have an indexed collection of compound sets, and that an indexed collection may be defined through set operations from other indexed collections. In addition to the operations previously mentioned, there are iterated union and intersection operators that apply to sets in the same way that an iterated sum applies to numbers. For example, the expressions

```
union {p in PROD} orig[p]
inter {p in PROD} orig[p]
```

represent the subset of origins that supply at least one product, and the subset of origins that supply all products.

The hierarchical relationship based on products that was observed in Figure 6-3 is seen in most of Figure 6-5 as well. The model repeatedly deals with the sets of all origins, destinations, and links associated with a particular product. The only exception comes in the last constraint, where the summation must be over all products shipped via a particular link:

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

Here it is necessary, following `sum`, to use a somewhat awkward indexing expression to describe a set that does not match the hierarchical organization.

In general, almost any model that can be written with indexed collections of sets can also be written with sets of tuples. As our examples suggest, indexed collections are most suitable for entities such as products and areas that have a hierarchical relationship. Sets of tuples are preferable, on the other hand, in dealing with entities like origins and destinations that are related symmetrically.

Exercises

6-1. Return to the production and transportation model of Figures 4-6 and 4-7. Using the `display` command, together with indexing expressions as demonstrated in Section 6.4, you can determine the membership of a variety of compound sets; for example, you can use

```
ampl: display {j in DEST, p in PROD: demand[j,p] > 500};
set {j in DEST, p in PROD: demand[j,p] > 500}  :=
(DET,coils)   (STL,bands)   (STL,coils)   (FRE,coils);
```

to show the set of all combinations of products and destinations where the demand is greater than 500.

(a) Use `display` to determine the membership of the following sets, which depend only on the data:

- All combinations of origins and products for which the production rate is greater than 150 tons per hour.
- All combinations of origins, destinations and products for which there is a shipping cost of ≤ $10 per ton.
- All combinations of origins and destinations for which the shipping cost of coils is ≤ $10 per ton.
- All combinations of origins and products for which the production cost per hour is less than $30,000.
- All combinations of origins, destinations and products for which the transportation cost is more than 15% of the production cost.
- All combinations of origins, destinations and products for which the transportation cost is more than 15% but less than 25% of the production cost.

(b) Use `display` to determine the membership of the following sets, which depend on the optimal solution as well as on the data:

- All combinations of origins and products for which there is production of at least 1000 tons.
- All combinations of origins, destinations and products for which there is a nonzero amount shipped.
- All combinations of origins and products for which more than 10 hours are used in production.
- All combinations of origins and products such that the product accounts for more than 25% of the hours available at the origin.

– All combinations of origins and products such that the total amount of the product shipped from the origin is at least 1000 tons.

6-2. This exercise resembles the previous one, but asks about the ordered-pair version of the transportation model in Figure 6-2.

(a) Use `display` and indexing expressions to determine the membership of the following sets:

– Origin-destination links that have a transportation cost less than $10 per ton.

– Destinations that can be served by `GARY`.

– Origins that can serve `FRE`.

– Links that are used for transportation in the optimal solution.

– Links that are used for transportation from `CLEV` in the optimal solution.

– Destinations to which the total cost of shipping, from all origins, exceeds $20,000.

(b) Use the `display` command and the `setof` operator to determine the membership of the following sets:

– Destinations that have a shipping cost of more than 20 from any origin.

– All destination-origin pairs `(j,i)` such that the link from `i` to `j` is used in the optimal solution.

6-3. Use `display` and appropriate set expressions to determine the membership of the following sets from the multiperiod production model of Figures 6-3 and 6-4:

– All market areas served with any of the products.

– All combinations of products, areas and weeks such that the amount actually sold in the optimal solution equals the maximum that can be sold.

– All combinations of products and weeks such that the total sold in all areas is greater than or equal to 6000 tons.

6-4. To try the following experiment, first enter these declarations:

```
ampl: set Q = {1..10,1..10,1..10,1..10,1..10,1..10};
ampl: set S within Q;
ampl: data;
ampl: set S := 1 2 3 3 4 5  2 3 4 4 5 6  3 4 5 5 6 7  4 5 6 7 8 9 ;
```

(a) Now try the following two commands:

```
display S;
display {(a,b,c,d,e,f) in Q: (a,b,c,d,e,f) in S};
```

The two expressions in these commands represent the same set, but do you get the same speed of response from AMPL? Explain the cause of the difference.

(b) Predict the result of the command `display Q`.

6-5. This exercise asks you to reformulate the diet model of Figure 2-1 in a variety of ways, using compound sets.

(a) Reformulate the diet model so that it uses a declaration

```
set GIVE within {NUTR,FOOD};
```

to define a subset of pairs `(i,j)` such that nutrient `i` can be found in food `j`.

(b) Reformulate the diet model so that it uses a declaration

```
set FN {NUTR} within FOOD;
```

to define, for each nutrient `i`, the set `FN[i]` of all foods that can supply that nutrient.

(c) Reformulate the diet model so that it uses a declaration

```
set NF {FOOD} within NUTR;
```

to define, for each food `j`, the set `NF[j]` of all nutrients supplied by that food. Explain why you find this formulation more or less natural and convenient than the one in (b).

6-6. Re-read the suggestions in Section 6.3, and complete the following reformulations of the multicommodity transportation model:

(a) Use a subset `LINKS` of origin-destination pairs.

(b) Use a subset `ROUTES` of origin-destination-product triples.

(c) Use a subset `MARKETS` of destination-product pairs, with the property that product `p` can be sold at destination `j` if and only if `(j,p)` is in the subset.

6-7. Carry through the following two suggestions from Section 6.4 for enhancements to the multicommodity transportation problem of Figure 4-1.

(a) Add a declaration

```
set DEMAND = {j in DEST, p in PROD: demand[j,p] > 0};
```

and index the variables over `{ORIG,DEMAND}`, so that variables are defined only where they might be needed to meet demand. Make all of the necessary changes in the rest of the model to use this set.

(b) Add the declarations

```
set LINKS within {ORIG,DEST};
set TRANSF = {i1 in ORIG, i2 in ORIG: i1 <> i2};
```

Define variables over `LINKS` to represent shipments to destinations, and over `TRANSF` to represent shipments between origins. The constraint at each origin now must say that total shipments out — to other origins as well as to destinations — must equal supply plus shipments in from other origins. Complete the formulation for this case.

6-8. Reformulate the model from Exercise 3-3(b) so that it uses a set `LINK1` of allowable plant-mill shipment pairs, and a set `LINK2` of allowable mill-factory shipment pairs.

6-9. As chairman of the program committee for a prestigious scientific conference, you must assign submitted papers to volunteer referees. To do so in the most effective way, you can formulate an LP model along the lines of the assignment model discussed in Chapter 3, but with a few extra twists.

After looking through the papers and the list of referees, you can compile the following data:

```
set Papers;
set Referees;
set Categories;

set PaperKind within {Papers,Categories};
set Willing within {Referees,Categories};
```

The contents of the first two sets are self-evident, while the third set contains subject categories into which papers may be classified. The set `PaperKind` contains a pair `(p,c)` if paper `p` falls

into category c; in general, a paper can fit into several categories. The set Willing contains a pair (r,c) if referee r is willing to handle papers in category c.

(a) What is the dimension of the set

```
{(r,c) in Willing, (p,c) in PaperKind}
```

and what is the significance of the tuples contained in this set?

(b) Based on your answer to (a), explain why the declaration

```
set CanHandle = setof {(r,c) in Willing, (p,c) in PaperKind} (r,p);
```

gives the set of pairs (r,p) such that referee r can be assigned paper p.

Your model could use parameters ppref and variables Review indexed over CanHandle; ppref[r,p] would be the preference of referee r for paper p, and Review[r,p] would be 1 if referee r were assigned paper p, or 0 otherwise. Assuming higher preferences are better, write out the declarations for these components and for an objective function to maximize the sum of preferences of all assignments.

(c) Unfortunately, you don't have the referees' preferences for individual papers, since they haven't seen any papers yet. What you have are their preferences for different categories:

```
param cpref {Willing} integer >= 0, <= 5;
```

Explain why it would make sense to replace ppref[r,p] in your objective by

```
max {(r,c) in Willing: (p,c) in PaperKind} cpref[r,c]
```

(d) Finally, you must define the following parameters that indicate how much work is to be done:

```
param nreferees integer > 0;     # referees needed per paper
param minwork integer > 0;       # min papers to each referee
param maxwork integer > minwork; # max papers to each referee
```

Formulate the appropriate assignment constraints. Complete the model, by formulating constraints that each paper must have the required number of referees, and that each referee must be assigned an acceptable number of papers.