15

Network Linear Programs

Models of networks have appeared in several chapters, notably in the transportation problems in Chapter 3. We now return to the formulation of these models, and AMPL's features for handling them.

Figure 15-1 shows the sort of diagram commonly used to describe a network problem. A circle represents a *node* of the network, and an arrow denotes an *arc* running from one node to another. A *flow* of some kind travels from node to node along the arcs, in the directions of the arrows.

An endless variety of models involve optimization over such networks. Many cannot be expressed in any straightforward algebraic way or are very difficult to solve. Our discussion starts with a particular class of network optimization models in which the decision variables represent the amounts of flow on the arcs, and the constraints are limited to two kinds: simple bounds on the flows, and conservation of flow at the nodes. Models restricted in this way give rise to the problems known as network linear programs. They are especially easy to describe and solve, yet are widely applicable. Some of their benefits extend to certain generalizations of the network flow form, which we also touch upon.

We begin with minimum-cost transshipment models, which are the largest and most intuitive source of network linear programs, and then proceed to other well-known cases: maximum flow, shortest path, transportation and assignment models. Examples are initially given in terms of standard AMPL variables and constraints, defined in `var` and `subject to` declarations. In later sections, we introduce `node` and `arc` declarations that permit models to be described more directly in terms of their network structure. The last section discusses formulating network models so that the resulting linear programs can be solved most efficiently.

15.1 Minimum-cost transshipment models

As a concrete example, imagine that the nodes and arcs in Figure 15-1 represent cities and intercity transportation links. A manufacturing plant at the city marked PITT will

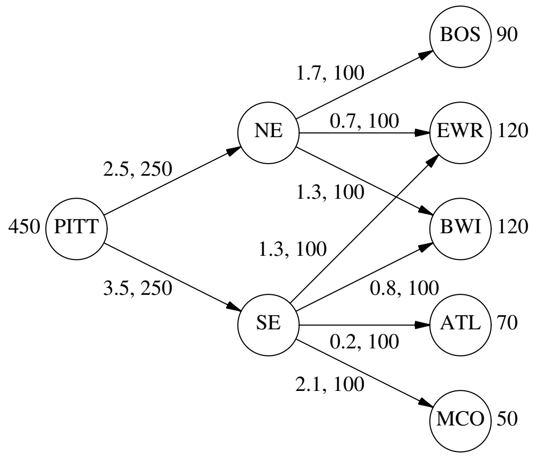


Figure 15-1: A directed network.

make 450,000 packages of a certain product in the next week, as indicated by the 450 at the left of the diagram. The cities marked NE and SE are the northeast and southeast distribution centers, which receive packages from the plant and transship them to warehouses at the cities coded as BOS, EWR, BWI, ATL and MCO. (Frequent flyers will recognize Boston, Newark, Baltimore, Atlanta, and Orlando.) These warehouses require 90, 120, 120, 70 and 50 thousand packages, respectively, as indicated by the numbers at the right. For each intercity link there is a shipping cost per thousand packages and an upper limit on the packages that can be shipped, indicated by the two numbers next to the corresponding arrow in the diagram.

The optimization problem over this network is to find the lowest-cost plan of shipments that uses only the available links, respects the specified capacities, and meets the requirements at the warehouses. We first model this as a general network flow problem, and then consider alternatives that specialize the model to the particular situation at hand. We conclude by introducing a few of the most common variations on the network flow constraints.

A general transshipment model

To write a model for any problem of shipments from city to city, we can start by defining a set of cities and a set of links. Each link is in turn defined by a start city and an end city, so we want the set of links to be a subset of the set of ordered pairs of cities:

```
set CITIES;
set LINKS within (CITIES cross CITIES);
```

Corresponding to each city there is potentially a supply of packages and a demand for packages:

```
param supply {CITIES} >= 0;
param demand {CITIES} >= 0;
```

In the case of the problem described by Figure 15-1, the only nonzero value of `supply` should be the one for PITT, where packages are manufactured and supplied to the distribution network. The only nonzero values of `demand` should be those corresponding to the five warehouses.

The costs and capacities are indexed over the links:

```
param cost {LINKS} >= 0;
param capacity {LINKS} >= 0;
```

as are the decision variables, which represent the amounts to ship over the links. These variables are nonnegative and bounded by the capacities:

```
var Ship {(i,j) in LINKS} >= 0, <= capacity[i,j];
```

The objective is

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

which represents the sum of the shipping costs over all of the links.

It remains to describe the constraints. At each city, the packages supplied plus packages shipped in must balance the packages demanded plus packages shipped out:

```
subject to Balance {k in CITIES}:
   supply[k] + sum {(i,k) in LINKS} Ship[i,k]
      = demand[k] + sum {(k,j) in LINKS} Ship[k,j];
```

Because the expression

```
sum {(i,k) in LINKS} Ship[i,k]
```

appears within the scope of definition of the dummy index `k`, the summation is interpreted to run over all cities `i` such that `(i,k)` is in `LINKS`. That is, the summation is over all links into city `k`; similarly, the second summation is over all links out of `k`. This indexing convention, which was explained in Section 6.2, is frequently useful in describing network balance constraints algebraically. Figures 15-2a and 15-2b display the complete model and data for the particular problem depicted in Figure 15-1.

If all of the variables are moved to the left of the = sign and the constants to the right, the `Balance` constraint becomes:

```
subject to Balance {k in CITIES}:
   sum {(i,k) in LINKS} Ship[i,k]
 - sum {(k,j) in LINKS} Ship[k,j]
     = demand[k] - supply[k];
```

This variation may be interpreted as saying that, at each city `k`, shipments in minus shipments out must equal ‘‘net demand’’. If no city has both a plant and a warehouse (as in

```
set CITIES;
set LINKS within (CITIES cross CITIES);

param supply {CITIES} >= 0;   # amounts available at cities
param demand {CITIES} >= 0;   # amounts required at cities

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

param cost {LINKS} >= 0;      # shipment costs/1000 packages
param capacity {LINKS} >= 0;  # max packages that can be shipped

var Ship {(i,j) in LINKS} >= 0, <= capacity[i,j];
                              # packages to be shipped

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

subject to Balance {k in CITIES}:
   supply[k] + sum {(i,k) in LINKS} Ship[i,k]
      = demand[k] + sum {(k,j) in LINKS} Ship[k,j];
```

Figure 15-2a: General transshipment model (`net1.mod`).

```
set CITIES := PITT  NE SE  BOS EWR BWI ATL MCO ;

set LINKS := (PITT,NE) (PITT,SE)
             (NE,BOS) (NE,EWR) (NE,BWI)
             (SE,EWR) (SE,BWI) (SE,ATL) (SE,MCO);

param supply  default 0 := PITT 450 ;

param demand  default 0 :=
  BOS  90,  EWR 120,  BWI 120,  ATL  70,  MCO  50;

param:      cost  capacity  :=
  PITT NE    2.5    250
  PITT SE    3.5    250

  NE BOS     1.7    100
  NE EWR     0.7    100
  NE BWI     1.3    100

  SE EWR     1.3    100
  SE BWI     0.8    100
  SE ATL     0.2    100
  SE MCO     2.1    100 ;
```

Figure 15-2b: Data for general transshipment model (`net1.dat`).

our example), then positive net demand always indicates warehouse cities, negative net demand indicates plant cities, and zero net demand indicates transshipment cities. Thus we could have gotten by with just one parameter `net_demand` in place of `demand` and `supply`, with the sign of `net_demand[k]` indicating what goes on at city `k`. Alternative formulations of this kind are often found in descriptions of network flow models.

Specialized transshipment models

The preceding general approach has the advantage of being able to accommodate any pattern of supplies, demands, and links between cities. For example, a simple change in the data would suffice to model a plant at one of the distribution centers, or to allow shipment links between some of the warehouses.

The disadvantage of a general formulation is that it fails to show clearly what arrangement of supplies, demands and links is expected, and in fact will allow inappropriate arrangements. If we know that the situation will be like the one shown in Figure 15-1, with supply at one plant, which ships to distribution centers, which then ship to warehouses that satisfy demand, the model can be specialized to exhibit and enforce such a structure.

To show explicitly that there are three different kinds of cities in the specialized model, we can declare them separately. We use a symbolic parameter rather than a set to hold the name of the plant, to specify that only one plant is expected:

```
param p_city symbolic;
set D_CITY;
set W_CITY;
```

There must be a link between the plant and each distribution center, so we need a subset of pairs only to specify which links connect distribution centers to warehouses:

```
set DW_LINKS within (D_CITY cross W_CITY);
```

With the declarations organized in this way, it is impossible to specify inappropriate kinds of links, such as ones between two warehouses or from a warehouse back to the plant.

One parameter represents the supply at the plant, and a collection of demand parameters is indexed over the warehouses:

```
param p_supply >= 0;
param w_demand {W_CITY} >= 0;
```

These declarations allow supply and demand to be defined only where they belong.

At this juncture, we can define the sets `CITIES` and `LINKS` and the parameters `supply` and `demand` as they would be required by our previous model:

```
set CITIES = {p_city} union D_CITY union W_CITY;
set LINKS = ({p_city} cross D_CITY) union DW_LINKS;

param supply {k in CITIES} =
   if k = p_city then p_supply else 0;

param demand {k in CITIES} =
   if k in W_CITY then w_demand[k] else 0;
```

The rest of the model can then be exactly as in the general case, as indicated in Figures 15-3a and 15-3b.

```
param p_city symbolic;

set D_CITY;
set W_CITY;
set DW_LINKS within (D_CITY cross W_CITY);

param p_supply >= 0;           # amount available at plant
param w_demand {W_CITY} >= 0;  # amounts required at warehouses

   check: p_supply = sum {k in W_CITY} w_demand[k];

set CITIES = {p_city} union D_CITY union W_CITY;
set LINKS = ({p_city} cross D_CITY) union DW_LINKS;

param supply {k in CITIES} =
   if k = p_city then p_supply else 0;

param demand {k in CITIES} =
   if k in W_CITY then w_demand[k] else 0;

### Remainder same as general transshipment model ###

param cost {LINKS} >= 0;      # shipment costs/1000 packages
param capacity {LINKS} >= 0;  # max packages that can be shipped

var Ship {(i,j) in LINKS} >= 0, <= capacity[i,j];
                              # packages to be shipped

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

subject to Balance {k in CITIES}:
   supply[k] + sum {(i,k) in LINKS} Ship[i,k]
      = demand[k] + sum {(k,j) in LINKS} Ship[k,j];
```

Figure 15-3a: Specialized transshipment model (net2.mod).

Alternatively, we can maintain references to the different types of cities and links throughout the model. This means that we must declare two types of costs, capacities and shipments:

```
param pd_cost {D_CITY} >= 0;
param dw_cost {DW_LINKS} >= 0;

param pd_cap {D_CITY} >= 0;
param dw_cap {DW_LINKS} >= 0;

var PD_Ship {i in D_CITY} >= 0, <= pd_cap[i];
var DW_Ship {(i,j) in DW_LINKS} >= 0, <= dw_cap[i,j];
```

The ‘‘pd’’ quantities are associated with shipments from the plant to distribution centers; because they all relate to shipments from the same plant, they need only be indexed over D_CITY. The ‘‘dw’’ quantities are associated with shipments from distribution centers to warehouses, and so are naturally indexed over DW_LINKS.

The total shipment cost can now be given as the sum of two summations:

```
param p_city := PITT ;

set D_CITY := NE SE ;
set W_CITY := BOS EWR BWI ATL MCO ;

set DW_LINKS := (NE,BOS) (NE,EWR) (NE,BWI)
                (SE,EWR) (SE,BWI) (SE,ATL) (SE,MCO);

param p_supply := 450 ;

param w_demand :=
  BOS  90,  EWR 120,  BWI 120,  ATL  70,  MCO  50;

param:      cost  capacity  :=
  PITT NE    2.5    250
  PITT SE    3.5    250

  NE BOS     1.7    100
  NE EWR     0.7    100
  NE BWI     1.3    100

  SE EWR     1.3    100
  SE BWI     0.8    100
  SE ATL     0.2    100
  SE MCO     2.1    100 ;
```

Figure 15-3b: Data for specialized transshipment model (`net2.dat`).

```
minimize Total_Cost:
   sum {i in D_CITY} pd_cost[i] * PD_Ship[i]
 + sum {(i,j) in DW_LINKS} dw_cost[i,j] * DW_Ship[i,j];
```

Finally, there must be three kinds of balance constraints, one for each kind of city. Shipments from the plant to the distribution centers must equal the supply at the plant:

```
subject to P_Bal:  sum {i in D_CITY} PD_Ship[i] = p_supply;
```

At each distribution center, shipments in from the plant must equal shipments out to all the warehouses:

```
subject to D_Bal {i in D_CITY}:
   PD_Ship[i] = sum {(i,j) in DW_LINKS} DW_Ship[i,j];
```

And at each warehouse, shipments in from all distribution centers must equal the demand:

```
subject to W_Bal {j in W_CITY}:
   sum {(i,j) in DW_LINKS} DW_Ship[i,j] = w_demand[j];
```

The whole model, with appropriate data, is shown in Figures 15-4a and 15-4b.

The approaches shown in Figures 15-3 and 15-4 are equivalent, in the sense that they cause the same linear program to be solved. The former is more convenient for experimenting with different network structures, since any changes affect only the data for the initial declarations in the model. If the network structure is unlikely to change, however,

```
set D_CITY;
set W_CITY;
set DW_LINKS within (D_CITY cross W_CITY);

param p_supply >= 0;            # amount available at plant
param w_demand {W_CITY} >= 0;   # amounts required at warehouses

   check: p_supply = sum {j in W_CITY} w_demand[j];

param pd_cost {D_CITY} >= 0;    # shipment costs/1000 packages
param dw_cost {DW_LINKS} >= 0;

param pd_cap {D_CITY} >= 0;     # max packages that can be shipped
param dw_cap {DW_LINKS} >= 0;

var PD_Ship {i in D_CITY} >= 0, <= pd_cap[i];
var DW_Ship {(i,j) in DW_LINKS} >= 0, <= dw_cap[i,j];
                                # packages to be shipped

minimize Total_Cost:
   sum {i in D_CITY} pd_cost[i] * PD_Ship[i] +
   sum {(i,j) in DW_LINKS} dw_cost[i,j] * DW_Ship[i,j];

subject to P_Bal:  sum {i in D_CITY} PD_Ship[i] = p_supply;

subject to D_Bal {i in D_CITY}:
   PD_Ship[i] = sum {(i,j) in DW_LINKS} DW_Ship[i,j];

subject to W_Bal {j in W_CITY}:
   sum {(i,j) in DW_LINKS} DW_Ship[i,j] = w_demand[j];
```

Figure 15-4a: Specialized transshipment model, version 2 (`net3.mod`).

the latter form facilitates alterations that affect only particular kinds of cities, such as the generalizations we describe next.

Variations on transshipment models

Some balance constraints in a network flow model may have to be inequalities rather than equations. In the example of Figure 15-4, if production at the plant can sometimes exceed total demand at the warehouses, we should replace `=` by `<=` in the `P_Bal` constraints.

A more substantial modification occurs when the quantity of flow that comes out of an arc does not necessarily equal the quantity that went in. As an example, a small fraction of the packages shipped from the plant may be damaged or stolen before the packages reach the distribution center. Suppose that a parameter `pd_loss` is introduced to represent the loss rate:

```
param pd_loss {D_CITY} >= 0, < 1;
```

Then the balance constraints at the distribution centers must be adjusted accordingly:

```
set D_CITY := NE SE ;

set W_CITY := BOS EWR BWI ATL MCO ;

set DW_LINKS := (NE,BOS) (NE,EWR) (NE,BWI)
                (SE,EWR) (SE,BWI) (SE,ATL) (SE,MCO);

param p_supply := 450 ;

param w_demand :=
  BOS  90,  EWR 120,  BWI 120,  ATL  70,  MCO  50;

param:  pd_cost  pd_cap :=
  NE       2.5    250
  SE       3.5    250 ;

param:    dw_cost  dw_cap :=
  NE BOS     1.7    100
  NE EWR     0.7    100
  NE BWI     1.3    100

  SE EWR     1.3    100
  SE BWI     0.8    100
  SE ATL     0.2    100
  SE MCO     2.1    100 ;
```

Figure 15-4b: Data for specialized transshipment model, version 2 (net3.dat).

```
subject to D_Bal {i in D_CITY}:
   (1-pd_loss[i]) * PD_Ship[i]
      = sum {(i,j) in DW_LINKS} DW_Ship[i,j];
```

The expression to the left of the = sign has been modified to reflect the fact that only (1-pd_loss[i]) * PD_Ship[i] packages arrive at city i when PD_Ship[i] packages are shipped from the plant.

A similar variation occurs when the flow is not measured in the same units throughout the network. If demand is reported in cartons rather than thousands of packages, for example, the model will require a parameter to represent packages per carton:

```
param ppc integer > 0;
```

Then the demand constraints at the warehouses are adjusted as follows:

```
subject to W_Bal {j in W_CITY}:
   sum {(i,j) in DW_LINKS} (1000/ppc) * DW_Ship[i,j]
      = w_demand[j];
```

The term (1000/ppc) * DW_Ship[i,j] represents the number of cartons received at warehouse j when DW_Ship[i,j] thousand packages are shipped from distribution center i.

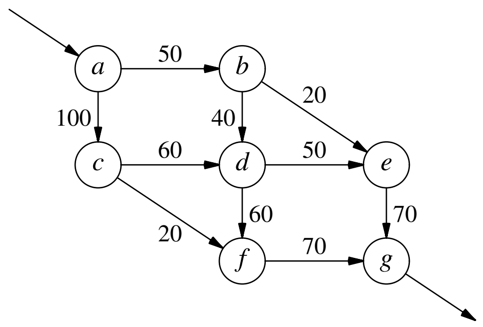


Figure 15-5: Traffic flow network.

15.2 Other network models

Not all network linear programs involve the transportation of things or the minimization of costs. We describe here three well-known model classes — maximum flow, shortest path, and transportation/assignment — that use the same kinds of variables and constraints for different purposes.

Maximum flow models

In some network design applications the concern is to send as much flow as possible through the network, rather than to send flow at lowest cost. This alternative is readily handled by dropping the balance constraints at the origins and destinations of flow, while substituting an objective that stands for total flow in some sense.

As a specific example, Figure 15-5 presents a diagram of a simple traffic network. The nodes and arcs represent intersections and roads; capacities, shown as numbers next to the roads, are in cars per hour. We want to find the maximum traffic flow that can enter the network at a and leave at g.

A model for this situation begins with a set of intersections, and symbolic parameters to indicate the intersections that serve as entrance and exit to the road network:

```
set INTER;

param entr symbolic in INTER;
param exit symbolic in INTER, <> entr;
```

The set of roads is defined as a subset of the pairs of intersections:

```
set ROADS within (INTER diff {exit}) cross (INTER diff {entr});
```

This definition ensures that no road begins at the exit or ends at the entrance.

Next, the capacity and traffic load are defined for each road:

```
set INTER;  # intersections

param entr symbolic in INTER;           # entrance to road network
param exit symbolic in INTER, <> entr;  # exit from road network

set ROADS within (INTER diff {exit}) cross (INTER diff {entr});

param cap {ROADS} >= 0;                             # capacities
var Traff {(i,j) in ROADS} >= 0, <= cap[i,j];  # traffic loads

maximize Entering_Traff: sum {(entr,j) in ROADS} Traff[entr,j];

subject to Balance {k in INTER diff {entr,exit}}:
  sum {(i,k) in ROADS} Traff[i,k] = sum {(k,j) in ROADS} Traff[k,j];

data;

set INTER := a b c d e f g ;

param entr := a ;
param exit := g ;

param:  ROADS:  cap :=
          a b     50,    a c    100
          b d     40,    b e     20
          c d     60,    c f     20
          d e     50,    d f     60
          e g     70,    f g     70 ;
```

Figure 15-6: Maximum traffic flow model and data (`netmax.mod`).

```
param cap {ROADS} >= 0;
var Traff {(i,j) in ROADS} >= 0, <= cap[i,j];
```

The constraints say that except for the entrance and exit, flow into each intersection equals flow out:

```
subject to Balance {k in INTER diff {entr,exit}}:
   sum {(i,k) in ROADS} Traff[i,k]
      = sum {(k,j) in ROADS} Traff[k,j];
```

Given these constraints, the flow out of the entrance must be the total flow through the network, which is to be maximized:

```
maximize Entering_Traff: sum {(entr,j) in ROADS} Traff[entr,j];
```

We could equally well maximize the total flow into the exit. The entire model, along with data for the example shown in Figure 15-5, is presented in Figure 15-6. Any linear programming solver will find a maximum flow of 130 cars per hour.

Shortest path models

If you were to use the optimal solution to any of our models thus far, you would have to send each of the packages, cars, or whatever along some path from a supply (or entrance) node to a demand (or exit) node. The values of the decision variables do not directly say what the optimal paths are, or how much flow must go on each one. Usually it is not too hard to deduce these paths, however, especially when the network has a regular or special structure.

If a network has just one unit of supply and one unit of demand, the optimal solution assumes a quite different nature. The variable associated with each arc is either 0 or 1, and the arcs whose variables have value 1 comprise a minimum-cost path from the supply node to the demand node. Often the ‘‘costs’’ are in fact times or distances, so that the optimum gives a shortest path.

Only a few changes need be made to the maximum flow model of Figure 15-6 to turn it into a shortest path model. There are still a parameter and a variable associated with each road from `i` to `j`, but we call them `time[i,j]` and `Use[i,j]`, and the sum of their products yields the objective:

```
param time {ROADS} >= 0;         # times to travel roads
var Use {(i,j) in ROADS} >= 0;   # 1 iff (i,j) in shortest path

minimize Total_Time: sum {(i,j) in ROADS} time[i,j] * Use[i,j];
```

Since only those variables `Use[i,j]` on the optimal path equal 1, while the rest are 0, this sum does correctly represent the total time to traverse the optimal path. The only other change is the addition of a constraint to ensure that exactly one unit of flow is available at the entrance to the network:

```
subject to Start:  sum {(entr,j) in ROADS} Use[entr,j] = 1;
```

The complete model is shown in Figure 15-7. If we imagine that the numbers on the arcs in Figure 15-5 are travel times in minutes rather than capacities, the data are the same; AMPL finds the solution as follows:

```
ampl: model netshort.mod;
ampl: solve;
MINOS 5.5: optimal solution found.
1 iterations, objective 140

ampl: option omit_zero_rows 1;
ampl: display Use;
Use :=
a b   1
b e   1
e g   1
;
```

The shortest path is `a` → `b` → `e` → `g`, which takes 140 minutes.

Transportation and assignment models

The best known and most widely used special network structure is the ‘‘bipartite’’ structure depicted in Figure 15-8. The nodes fall into two groups, one serving as origins of flow and the other as destinations. Each arc connects an origin to a destination.

```
set INTER;  # intersections

param entr symbolic in INTER;           # entrance to road network
param exit symbolic in INTER, <> entr;  # exit from road network

set ROADS within (INTER diff {exit}) cross (INTER diff {entr});

param time {ROADS} >= 0;         # times to travel roads
var Use {(i,j) in ROADS} >= 0;   # 1 iff (i,j) in shortest path

minimize Total_Time: sum {(i,j) in ROADS} time[i,j] * Use[i,j];

subject to Start:  sum {(entr,j) in ROADS} Use[entr,j] = 1;

subject to Balance {k in INTER diff {entr,exit}}:
   sum {(i,k) in ROADS} Use[i,k] = sum {(k,j) in ROADS} Use[k,j];

data;

set INTER := a b c d e f g ;

param entr := a ;
param exit := g ;

param:  ROADS:  time :=
         a b     50,    a c    100
         b d     40,    b e     20
         c d     60,    c f     20
         d e     50,    d f     60
         e g     70,    f g     70 ;
```

Figure 15-7: Shortest path model and data (`netshort.mod`).

The minimum-cost transshipment model on this network is known as the transportation model. The special case in which every origin is connected to every destination was introduced in Chapter 3; an AMPL model and sample data are shown in Figures 3-1a and 3-1b. A more general example analogous to the models developed earlier in this chapter, where a set `LINKS` specifies the arcs of the network, appears in Figures 6-2a and 6-2b.

Every path from an origin to a destination in a bipartite network consists of one arc. Or, to say the same thing another way, the optimal flow along an arc of the transportation model gives the actual amount shipped from some origin to some destination. This property permits the transportation model to be viewed alternatively as a so-called assignment model, in which the optimal flow along an arc is the amount of something from the origin that is assigned to the destination. The meaning of assignment in this context can be broadly construed, and in particular need not involve a shipment in any sense.

One of the more common applications of the assignment model is matching people to appropriate targets, such as jobs, offices or even other people. Each origin node is associated with one person, and each destination node with one of the targets — for example, with one project. The sets might then be defined as follows:

```
set PEOPLE;
set PROJECTS;
set ABILITIES within (PEOPLE cross PROJECTS);
```

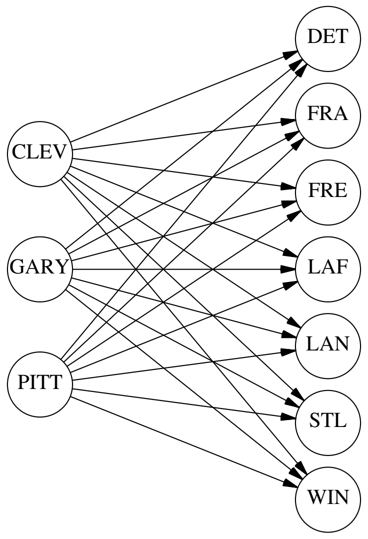


Figure 15-8: Bipartite network.

The set `ABILITIES` takes the role of `LINKS` in our earlier models; a pair `(i,j)` is placed in this set if and only if person `i` can work on project `j`.

As one possibility for continuing the model, the supply at node `i` could be the number of hours that person `i` is available to work, and the demand at node `j` could be the number of hours required for project `j`. Variables `Assign[i,j]` would represent the number of hours of person `i`'s time assigned to project `j`. Also associated with each pair `(i,j)` would be a cost per hour, and a maximum number of hours that person `i` could contribute to job `j`. The resulting model is shown in Figure 15-9.

Another possibility is to make the assignment in terms of people rather than hours. The supply at every node `i` is 1 (person), and the demand at node `j` is the number of people required for project `j`. The supply constraints ensure that `Assign[i,j]` is not greater than 1; and it will equal 1 in an optimal solution if and only if person `i` is assigned to project `j`. The coefficient `cost[i,j]` could be some kind of cost of assigning person `i` to project `j`, in which case the objective would still be to minimize total cost. Or the coefficient could be the ranking of person `i` for project `j`, perhaps on a scale from 1 (highest) to 10 (lowest). Then the model would produce an assignment for which the total of the rankings is the best possible.

Finally, we can imagine an assignment model in which the demand at each node `j` is also 1; the problem is then to match people to projects. In the objective, `cost[i,j]` could be the number of hours that person `i` would need to complete project `j`, in which case the model would find the assignment that minimizes the total hours of work needed to finish all the projects. You can create a model of this kind by replacing all references

```
set PEOPLE;
set PROJECTS;

set ABILITIES within (PEOPLE cross PROJECTS);

param supply {PEOPLE} >= 0;   # hours each person is available
param demand {PROJECTS} >= 0; # hours each project requires

check: sum {i in PEOPLE} supply[i] = sum {j in PROJECTS} demand[j];

param cost {ABILITIES} >= 0;   # cost per hour of work
param limit {ABILITIES} >= 0;  # maximum contributions to projects

var Assign {(i,j) in ABILITIES} >= 0, <= limit[i,j];

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

subject to Supply {i in PEOPLE}:
   sum {(i,j) in ABILITIES} Assign[i,j] = supply[i];

subject to Demand {j in PROJECTS}:
   sum {(i,j) in ABILITIES} Assign[i,j] = demand[j];
```

Figure 15-9: Assignment model (`netasgn.mod`).

to `supply[i]` and `demand[j]` by 1 in Figure 15-9. Objective coefficients representing rankings are an option for this model as well, giving rise to the kind of assignment model that we used as an example in Section 3.3.

15.3 Declaring network models by `node` and `arc`

AMPL's algebraic notation has great power to express a variety of network linear programs, but the resulting constraint expressions are often not as natural as we would like. While the idea of constraining ‘‘flow out minus flow in’’ at each node is easy to describe and understand, the corresponding algebraic constraints tend to involve terms like

```
sum {(i,k) in LINKS} Ship[i,k]
```

that are not so quickly comprehended. The more complex and realistic the network, the worse the problem. Indeed, it can be hard to tell whether a model's algebraic constraints represent a valid collection of flow balances on a network, and consequently whether specialized network optimization software (described later in this chapter) can be used.

Algebraic formulations of network flows tend to be problematical because they are constructed explicitly in terms of variables and constraints, while the nodes and arcs are merely implicit in the way that the constraints are structured. People prefer to approach network flow problems in the opposite way. They imagine giving an explicit definition of nodes and arcs, from which flow variables and balance constraints implicitly arise. To deal with this situation, AMPL provides an alternative that allows network concepts to be declared directly in a model.

The network extensions to AMPL include two new kinds of declaration, `node` and `arc`, that take the place of the `subject to` and `var` declarations in an algebraic constraint formulation. The `node` declarations name the nodes of a network, and characterize the flow balance constraints at the nodes. The `arc` declarations name and define the arcs, by specifying the nodes that arcs connect, and by providing optional information such as bounds and costs that are associated with arcs.

This section introduces `node` and `arc` by showing how they permit various examples from earlier in this chapter to be reformulated conveniently. The following section presents the rules for these declarations more systematically.

A general transshipment model

In rewriting the model of Figure 15-2a using `node` and `arc`, we can retain all of the `set` and `param` declarations and associated data. The changes affect only the three declarations — `minimize`, `var`, and `subject to` — that define the linear program.

There is a node in the network for every member of the set `CITIES`. Using a `node` declaration, we can say this directly:

```
node Balance {k in CITIES}: net_in = demand[k] - supply[k];
```

The keyword `net_in` stands for ‘‘net input’’, that is, the flow in minus the flow out, so this declaration says that net flow in must equal net demand at each node `Balance[k]`. Thus it says the same thing as the constraint named `Balance[k]` in the algebraic version, except that it uses the concise term `net_in` in place of the lengthy expression

```
sum {(i,k) in LINKS} Ship[i,k] - sum {(k,j) in LINKS} Ship[k,j]
```

Indeed, the syntax of `subject to` and `node` are practically the same except for the way that the conservation-of-flow constraint is stated. (The keyword `net_out` may also be used to stand for flow out minus flow in, so that we could have written `net_out = supply[k] - demand[k]`.)

There is an arc in the network for every pair in the set `LINKS`. This too can be said directly, using an `arc` declaration:

```
arc Ship {(i,j) in LINKS} >= 0, <= capacity[i,j],
   from Balance[i], to Balance[j], obj Total_Cost cost[i,j];
```

An arc `Ship[i,j]` is defined for each pair in `LINKS`, with bounds of 0 and `capacity[i,j]` on its flow; to this extent, the `arc` and `var` declarations are the same. The `arc` declaration contains additional phrases, however, to say that the arc runs from the node named `Balance[i]` to the node named `Balance[j]`, with a linear coefficient of `cost[i,j]` in the objective function named `Total_Cost`. These phrases use the keywords `from`, `to`, and `obj`.

Since the information about the objective function is included in the `arc` declaration, it is not needed in the `minimize` declaration, which reduces to:

```
minimize Total_Cost;
```

```
set CITIES;
set LINKS within (CITIES cross CITIES);

param supply {CITIES} >= 0;   # amounts available at cities
param demand {CITIES} >= 0;   # amounts required at cities

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

param cost {LINKS} >= 0;      # shipment costs/1000 packages
param capacity {LINKS} >= 0;  # max packages that can be shipped

minimize Total_Cost;

node Balance {k in CITIES}: net_in = demand[k] - supply[k];

arc Ship {(i,j) in LINKS} >= 0, <= capacity[i,j],
   from Balance[i], to Balance[j], obj Total_Cost cost[i,j];
```

Figure 15-10: General transshipment model with `node` and `arc` (`net1node.mod`).

The whole model is shown in Figure 15-10.

As this description suggests, `arc` and `node` take the place of `var` and `subject to`, respectively. In fact AMPL treats an `arc` declaration as a definition of variables, so that you would still say `display Ship` to look at the optimal flows in the network model of Figure 15-10; it treats a `node` declaration as a definition of constraints. The difference is that `node` and `arc` present the model in a way that corresponds more directly to its appearance in a network diagram. The description of the nodes always comes first, followed by a description of how the arcs connect the nodes.

A specialized transshipment model

The `node` and `arc` declarations make it easy to define a linear program for a network that has several different kinds of nodes and arcs. For an example we return to the specialized model of Figure 15-4a.

The network has a plant node, a distribution center node for each member of `D_CITY`, and a warehouse node for each member of `W_CITY`. Thus the model requires three node declarations:

```
node Plant: net_out = p_supply;
node Dist {i in D_CITY};
node Whse {j in W_CITY}: net_in = w_demand[j];
```

The balance conditions say that flow out of node `Plant` must be `p_supply`, while flow into node `Whse[j]` is `w_demand[j]`. (The network has no arcs into the plant or out of the warehouses, so `net_out` and `net_in` are just the flow out and flow in, respectively.) The conditions at node `Dist[i]` could be written either `net_in = 0` or `net_out = 0`, but since these are assumed by default we need not specify any condition at all.

```
set D_CITY;
set W_CITY;
set DW_LINKS within (D_CITY cross W_CITY);

param p_supply >= 0;           # amount available at plant
param w_demand {W_CITY} >= 0;  # amounts required at warehouses

   check: p_supply = sum {j in W_CITY} w_demand[j];

param pd_cost {D_CITY} >= 0;   # shipment costs/1000 packages
param dw_cost {DW_LINKS} >= 0;

param pd_cap {D_CITY} >= 0;    # max packages that can be shipped
param dw_cap {DW_LINKS} >= 0;

minimize Total_Cost;

node Plant: net_out = p_supply;

node Dist {i in D_CITY};

node Whse {j in W_CITY}: net_in = w_demand[j];

arc PD_Ship {i in D_CITY} >= 0, <= pd_cap[i],
   from Plant, to Dist[i], obj Total_Cost pd_cost[i];

arc DW_Ship {(i,j) in DW_LINKS} >= 0, <= dw_cap[i,j],
   from Dist[i], to Whse[j], obj Total_Cost dw_cost[i,j];
```

Figure 15-11: Specialized transshipment model with node and arc (net3node.mod).

This network has two kinds of arcs. There is an arc from the plant to each member of D_CITY, which can be declared by:

```
arc PD_Ship {i in D_CITY} >= 0, <= pd_cap[i],
   from Plant, to Dist[i], obj Total_Cost pd_cost[i];
```

And there is an arc from distribution center i to warehouse j for each pair (i,j) in DW_LINKS:

```
arc DW_Ship {(i,j) in DW_LINKS} >= 0, <= dw_cap[i,j],
   from Dist[i], to Whse[j], obj Total_Cost dw_cost[i,j];
```

The arc declarations specify the relevant bounds and objective coefficients, as in our previous example. The whole model is shown in Figure 15-11.

Variations on transshipment models

The balance conditions in node declarations may be inequalities, like ordinary algebraic balance constraints. If production at the plant can sometimes exceed total demand at the warehouses, it would be appropriate to give the condition in the declaration of node Plant as net_out <= p_supply.

An arc declaration can specify losses in transit by adding a factor at the end of the to phrase:

```
arc PD_Ship {i in D_CITY} >= 0, <= pd_cap[i],
   from Plant, to Dist[i] 1-pd_loss[i],
   obj Total_Cost pd_cost[i];
```

This is interpreted as saying that `PD_Ship[i]` is the number of packages that leave node `Plant`, but `(1-pd_loss[i]) * PD_Ship[i]` is the number that enter node `Dist[i]`.

The same option can be used to specify conversions. To use our previous example, if shipments are measured in thousands of packages but demands are measured in cartons, the arcs from distribution centers to warehouses should be declared as:

```
arc DW_Ship {(i,j) in DW_LINKS} >= 0, <= dw_cap[i,j],
   from Dist[i], to Whse[j] (1000/ppc),
   obj Total_Cost dw_cost[i,j];
```

If the shipments to warehouses are also measured in cartons, the factor should be applied at the distribution center:

```
arc DW_Ship {(i,j) in DW_LINKS} >= 0, <= dw_cap[i,j],
   from Dist[i] (ppc/1000), to Whse[j],
   obj Total_Cost dw_cost[i,j];
```

A loss factor could also be applied to the `to` phrase in these examples.

Maximum flow models

In the diagram of Figure 15-5 that we have used to illustrate the maximum flow problem, there are three kinds of intersections represented by nodes: the one where traffic enters, the one where traffic leaves, and the others where traffic flow is conserved. Thus a model of the network could have three corresponding node declarations:

```
node Entr_Int: net_out >= 0;
node Exit_Int: net_in >= 0;

node Intersection {k in INTER diff {entr,exit}};
```

The condition `net_out >= 0` implies that the flow out of node `Entr_Int` may be any amount at all; this is the proper condition, since there is no balance constraint on the entrance node. An analogous comment applies to the condition for node `Exit_Int`.

There is one arc in this network for each pair `(i,j)` in the set `ROADS`. Thus the declaration should look something like this:

```
arc Traff {(i,j) in ROADS} >= 0, <= cap[i,j],  # NOT RIGHT
   from Intersection[i], to Intersection[j],
   obj Entering_Traff (if i = entr then 1);
```

Since the aim is to maximize the total traffic leaving the entrance node, the arc is given a coefficient of 1 in the objective if and only if `i` takes the value `entr`. When `i` does take this value, however, the arc is specified to be from `Intersection[entr]`, a node that does not exist; the arc should rather be from node `Entr_Int`. Similarly, when `j` takes the value `exit`, the arc should not be to `Intersection[exit]`, but to

Exit_Int. AMPL will catch these errors and issue a message naming one of the nonexistent nodes that has been referenced.

It might seem reasonable to use an if-then-else to get around this problem, in the following way:

```
arc Traff {(i,j) in ROADS} >= 0, <= cap[i,j],  # SYNTAX ERROR
   from (if i = entr then Entr_Int else Intersection[i]),
   to   (if j = exit then Exit_Int else Intersection[j]),
   obj Entering_Traff (if i = entr then 1);
```

However, the if-then-else construct in AMPL does not apply to model components such as Entr_Int and Intersection[i]; this version will be rejected as a syntax error. Instead you need to use from and to phrases qualified by indexing expressions:

```
arc Traff {(i,j) in ROADS} >= 0, <= cap[i,j],
   from {if i = entr} Entr_Int,
   from {if i <> entr} Intersection[i],
   to   {if j = exit} Exit_Int,
   to   {if j <> exit} Intersection[j],
   obj Entering_Traff (if i = entr then 1);
```

The special indexing expression beginning with if works much the same way here as it does for constraints (Section 8.4); the from or to phrase is processed if the condition following if is true. Thus Traff[i,j] is declared to be from Entr_Int if i equals entr, and to be from Intersection[i] if i is not equal to entr, which is what we intend.

As an alternative, we can combine the declarations of the three different kinds of nodes into one. Observing that net_out is positive or zero for Entr_Int, negative or zero for Exit_Int, and zero for all other nodes Intersection[i], we can declare:

```
node Intersection {k in INTER}:
   (if k = exit then -Infinity)
      <= net_out <= (if k = entr then Infinity);
```

The nodes that were formerly declared as Entr_Int and Exit_Int are now just Intersection[entr] and Intersection[exit], and consequently the arc declaration that we previously marked ‘‘not right’’ now works just fine. The choice between this version and the previous one is entirely a matter of convenience and taste. (Infinity is a predefined AMPL parameter that may be used to specify any ‘‘infinitely large’’ bound; its technical definition is given in Section A.7.2.)

Arguably the AMPL formulation that is most convenient and appealing is neither of the above, but rather comes from interpreting the network diagram of Figure 15-5 in a slightly different way. Suppose that we view the arrows into the entrance node and out of the exit node as representing additional arcs, which happen to be adjacent to only one node rather than two. Then flow in equals flow out at every intersection, and the node declaration simplifies to:

```
node Intersection {k in INTER};
```

```
set INTER;  # intersections

param entr symbolic in INTER;           # entrance to road network
param exit symbolic in INTER, <> entr;  # exit from road network

set ROADS within (INTER diff {exit}) cross (INTER diff {entr});

param cap {ROADS} >= 0;  # capacities of roads

node Intersection {k in INTER};

arc Traff_In >= 0, to Intersection[entr];
arc Traff_Out >= 0, from Intersection[exit];

arc Traff {(i,j) in ROADS} >= 0, <= cap[i,j],
   from Intersection[i], to Intersection[j];

maximize Entering_Traff: Traff_In;

data;

set INTER := a b c d e f g ;

param entr := a ;
param exit := g ;

param:  ROADS:  cap :=
         a b     50,    a c    100
         b d     40,    b e     20
         c d     60,    c f     20
         d e     50,    d f     60
         e g     70,    f g     70 ;
```

Figure 15-12: Maximum flow model with `node` and `arc` (`netmax3.mod`).

The two arcs ‘‘hanging’’ at the entrance and exit are defined in the obvious way, but include only a `to` or a `from` phrase:

```
arc Traff_In >= 0, to Intersection[entr];
arc Traff_Out >= 0, from Intersection[exit];
```

The arcs that represent roads within the network are declared as before:

```
arc Traff {(i,j) in ROADS} >= 0, <= cap[i,j],
   from Intersection[i], to Intersection[j];
```

When the model is represented in this way, the objective is to maximize `Traff_In` (or equivalently `Traff_Out`). We could do this by adding an `obj` phrase to the `arc` declaration for `Traff_In`, but in this case it is perhaps clearer to define the objective algebraically:

```
maximize Entering_Traff: Traff_In;
```

This version is shown in full in Figure 15-12.

15.4 Rules for `node` and `arc` declarations

Having defined `node` and `arc` by example, we now describe more comprehensively the required and optional elements of these declarations, and comment on their interaction with the conventional declarations `minimize` or `maximize`, `subject to`, and `var` when both kinds appear in the same model.

`node` declarations

A `node` declaration begins with the keyword `node`, a name, an optional indexing expression, and a colon. The expression following the colon, which describes the balance condition at the node, may have any of the following forms:

```
net-expr  =  arith-expr
net-expr <=  arith-expr
net-expr >=  arith-expr

arith-expr  =  net-expr
arith-expr <=  net-expr
arith-expr >=  net-expr

arith-expr <= net-expr <= arith-expr
arith-expr >= net-expr >= arith-expr
```

where an *arith-expr* may be any arithmetic expression that uses previously declared model components and currently defined dummy indices. A *net-expr* is restricted to one of the following:

```
± net_in                        ± net_out
± net_in + arith-expr           ± net_out + arith-expr
arith-expr ± net_in             arith-expr ± net_out
```

(and a unary + may be omitted). Each node defined in this way induces a constraint in the resulting linear program. A node name is treated like a constraint name in the AMPL command environment, for example in a `display` statement.

For declarations that use `net_in`, AMPL generates the constraint by substituting, at the place where `net_in` appears in the balance conditions, a linear expression that represents flow into the node minus flow out of the node. Declarations that use `net_out` are handled the same way, except that AMPL substitutes flow out minus flow in. The expressions for flow in and flow out are deduced from the `arc` declarations.

`arc` declarations

An `arc` declaration consists of the keyword `arc`, a name, an optional indexing expression, and a series of optional qualifying phrases. Each arc creates a variable in the resulting linear program, whose value is the amount of flow over the arc; the arc name may be used to refer to this variable elsewhere. All of the phrases that may appear in a `var` definition have the same significance in an `arc` definition; most commonly, the `>=`

and <= phrases are used to specify values for lower and upper bounds on the flow along the arc.

The `from` and `to` phrases specify the nodes connected by an arc. Usually these consist of the keyword `from` or `to` followed by a node name. An arc is interpreted to contribute to the flow out of the `from` node, and to the flow into the `to` node; these interpretations are what permit the inference of the constraints associated with the nodes.

Typically one `from` and one `to` phrase are specified in an `arc` declaration. Either may be omitted, however, as in Figure 15-12. Either may also be followed by an optional indexing expression, which should be one of two kinds:

- An indexing expression that specifies — depending on the data — an empty set (in which case the `from` or `to` phrase is ignored) or a set with one member (in which case the `from` or `to` phrase is used).
- An indexing expression of the special form `{if` *logical-expr*`}`, which causes the `from` or `to` phrase to be used if and only if the *logical-expr* evaluates to true.

It is possible to specify that an arc carries flow out of or into two or more nodes, by giving more than one `from` or `to` phrase, or by using an indexing expression that specifies a set having more than one member. The result is not a network linear program, however, and AMPL displays an appropriate warning message.

At the end of a `from` or `to` phrase, you may add an arithmetic expression representing a factor to multiply the flow, as shown in our examples of shipping-loss and change-of-unit variations in Section 15.3. If the factor is in the `to` phrase, it multiplies the arc variable in determining the flow into the specified node; that is, for a given flow along the arc, an amount equal to the `to`-factor times the flow is considered to enter the `to` node. A factor in the `from` phrase is interpreted analogously. The default factor is 1.

An optional `obj` phrase specifies a coefficient that will multiply the arc variable to create a linear term in a specified objective function. Such a phrase consists of the keyword `obj`, the name of an objective that has previously been defined in a `minimize` or `maximize` declaration, and an arithmetic expression for the coefficient value. The keyword may be followed by an indexing expression, which is interpreted as for the `from` and `to` phrases.

Interaction with objective declarations

If all terms in the objective function are specified through `obj` phrases in `arc` declarations, the declaration of the objective is simply `minimize` or `maximize` followed by an optional indexing expression and a name. This declaration must come before the `arc` declarations that refer to the objective.

Alternatively, arc names may be used as variables to specify the objective function in the usual algebraic way. In this case the objective must be declared after the arcs, as in Figure 15-12.

```
set CITIES;
set LINKS within (CITIES cross CITIES);

set PRODS;

param supply {CITIES,PRODS} >= 0;  # amounts available at cities
param demand {CITIES,PRODS} >= 0;  # amounts required at cities

   check {p in PRODS}:
      sum {i in CITIES} supply[i,p] = sum {j in CITIES} demand[j,p];

param cost {LINKS,PRODS} >= 0;     # shipment costs/1000 packages
param capacity {LINKS,PRODS} >= 0; # max packages shipped
param cap_joint {LINKS} >= 0;      # max total packages shipped/link

minimize Total_Cost;

node Balance {k in CITIES, p in PRODS}:
   net_in = demand[k,p] - supply[k,p];

arc Ship {(i,j) in LINKS, p in PRODS} >= 0, <= capacity[i,j,p],
   from Balance[i,p], to Balance[j,p], obj Total_Cost cost[i,j,p];

subject to Multi {(i,j) in LINKS}:
   sum {p in PRODS} Ship[i,j,p] <= cap_joint[i,j];
```

Figure 15-13: Multicommodity flow with side constraints (`netmulti.mod`).

Interaction with constraint declarations

The components defined in `arc` declarations may be used as variables in additional `subject to` declarations. The latter represent ‘‘side constraints’’ that are imposed in addition to balance of flow at the nodes.

As an example, consider how a multicommodity flow problem can be built from the node-and-arc network formulation in Figure 15-10. Following the approach in Section 4.1, we introduce a set `PRODS` of different products, and add it to the indexing of all parameters, nodes and arcs. The result is a separate network linear program for each product, with the objective function being the sum of the costs for all products. To tie these networks together, we provide for a joint limit on the total shipments along any link:

```
param cap_joint {LINKS} >= 0;

subject to Multi {(i,j) in LINKS}:
   sum {p in PRODS} Ship[p,i,j] <= cap_joint[i,j];
```

The final model, shown in Figure 15-13, is not a network linear program, but the network and non-network parts of it are cleanly separated.

Interaction with variable declarations

Just as an `arc` variable may be used in a `subject to` declaration, an ordinary `var` variable may be used in a `node` declaration. That is, the balance condition in a `node` declaration may contain references to variables that were defined by preceding `var` declarations. These references define ‘‘side variables’’ to the network linear program.

As an example, we again replicate the formulation of Figure 15-10 over the set `PRODS`. This time we tie the networks together by introducing a set of feedstocks and associated data:

```
set FEEDS;
param yield {PRODS,FEEDS} >= 0;
param limit {FEEDS,CITIES} >= 0;
```

We imagine that at city `k`, in addition to the amounts `supply[p,k]` of products available to be shipped, up to `limit[f,k]` of feedstock `f` can be converted into products; one unit of feedstock `f` gives rise to `yield[p,f]` units of each product `p`. A variable `Feed[f,k]` represents the amount of feedstock `f` used at city `k`:

```
var Feed {f in FEEDS, k in CITIES} >= 0, <= limit[f,k];
```

The balance condition for product `p` at city `k` can now say that the net flow out equals net supply plus the sum of the amounts derived from the various feedstocks:

```
node Balance {p in PRODS, k in CITIES}:
   net_out = supply[p,k] - demand[p,k]
      + sum {f in FEEDS} yield[p,f] * Feed[f,k];
```

The arcs are unchanged, leading to the model shown in Figure 15-14. At a given city `k`, the variables `Feed[f,k]` appear in the node balance conditions for all the different products, bringing together the product networks into a single linear program.

15.5 Solving network linear programs

All of the models that we have described in this chapter give rise to linear programs that have a ‘‘network’’ property of some kind. AMPL can send these linear programs to an LP solver and retrieve the optimal values, much as for any other class of LPs. If you use AMPL in this way, the network structure is helpful mainly as a guide to formulating the model and interpreting the results.

Many of the models that we have described belong as well to a more restricted class of problems that (confusingly) are also known as ‘‘network linear programs.’’ In modeling terms, the variables of a network LP must represent flows on the arcs of a network, and the constraints must be only of two types: bounds on the flows along the arcs, and limits on flow out minus flow in at the nodes. A more technical way to say the same thing is that each variable of a network linear program must appear in at most two constraints (aside from lower or upper bounds on the variables), such that the variable has a coefficient of +1 in at most one constraint, and a coefficient of –1 in at most one constraint.

‘‘Pure’’ network linear programs of this restricted kind have some very strong properties that make their use particularly desirable. So long as the supplies, demands, and

```
set CITIES;
set LINKS within (CITIES cross CITIES);
set PRODS;
param supply {PRODS,CITIES} >= 0;  # amounts available at cities
param demand {PRODS,CITIES} >= 0;  # amounts required at cities
   check {p in PRODS}:
      sum {i in CITIES} supply[p,i] = sum {j in CITIES} demand[p,j];
param cost {PRODS,LINKS} >= 0;     # shipment costs/1000 packages
param capacity {PRODS,LINKS} >= 0; # max packages shipped of product
set FEEDS;
param yield {PRODS,FEEDS} >= 0;    # amounts derived from feedstocks
param limit {FEEDS,CITIES} >= 0;   # feedstocks available at cities
minimize Total_Cost;
var Feed {f in FEEDS, k in CITIES} >= 0, <= limit[f,k];
node Balance {p in PRODS, k in CITIES}:
   net_out = supply[p,k] - demand[p,k]
      + sum {f in FEEDS} yield[p,f] * Feed[f,k];
arc Ship {p in PRODS, (i,j) in LINKS} >= 0, <= capacity[p,i,j],
   from Balance[p,i], to Balance[p,j],
   obj Total_Cost cost[p,i,j];
```

Figure 15-14: Multicommodity flow with side variables (netfeeds.mod).

bounds are integers, a network linear program must have an optimal solution in which all flows are integers. Moreover, if the solver is of a kind that finds ‘‘extreme’’ solutions (such as those based on the simplex method) it will always find one of the all-integer optimal solutions. We have taken advantage of this property, without explicitly mentioning it, in assuming that the variables in the shortest path problem and in certain assignment problems come out to be either zero or one, and never some fraction in between.

Network linear programs can also be solved much faster than other linear programs of comparable size, through the use of solvers that are specialized to take advantage of the network structure. If you write your model in terms of node and arc declarations, AMPL automatically communicates the network structure to the solver, and any special network algorithms available in the solver can be applied automatically. On the other hand, a network expressed algebraically using var and subject to may or may not be recognized by the solver, and certain options may have to be set to ensure that it is recognized. For example, when using the algebraic model of Figure 15-4a, you may see the usual response from the general LP algorithm:

```
ampl: model net3.mod; data net3.dat; solve;
CPLEX 8.0.0: optimal solution; objective 1819
1 dual simplex iterations (0 in phase I)
```

But when using the equivalent node and arc formulation of Figure 15-11, you may get a somewhat different response to reflect the application of a special network LP algorithm:

```
ampl: model net3node.mod
ampl: data net3.dat
ampl: solve;
CPLEX 8.0.0: optimal solution; objective 1819
Network extractor found 7 nodes and 7 arcs.
7 network simplex iterations.
0 simplex iterations (0 in phase I)
```

To determine how your favorite solver behaves in this situation, consult the solver-specific documentation that is supplied with your AMPL installation.

Because network linear programs are much easier to solve, especially with integer data, the success of a large-scale application may depend on whether a pure network formulation is possible. In the case of the multicommodity flow model of Figure 15-13, for example, the joint capacity constraints disrupt the network structure — they represent a third constraint in which each variable figures — but their presence cannot be avoided in a correct representation of the problem. Multicommodity flow problems thus do not necessarily have integer solutions, and are generally much harder to solve than single-commodity flow problems of comparable size.

In some cases, a judicious reformulation can turn what appears to be a more general model into a pure network model. Consider, for instance, a generalization of Figure 15-10 in which capacities are defined at the nodes as well as along the arcs:

```
param city_cap {CITIES} >= 0;
param link_cap {LINKS} >= 0;
```

The arc capacities represent, as before, upper limits on the shipments between cities. The node capacities limit the throughput, or total flow handled at a city, which may be written as the supply at the city plus the sum of the flows in, or equivalently as the demand at the city plus the sum of the flows out. Using the former, we arrive at the following constraint:

```
subject to through_limit {k in CITIES}:
   supply[k] + sum {(i,k) in LINKS} Ship[i,k] <= node_cap[k];
```

Viewed in this way, the throughput limit is another example of a ‘‘side constraint’’ that disrupts the network structure by adding a third coefficient for each variable. But we can achieve the same effect without a side constraint, by using two nodes to represent each city; one receives flow into a city plus any supply, and the other sends flow out of a city plus any demand:

```
node Supply {k in CITIES}: net_out = supply[k];
node Demand {k in CITIES}: net_in = demand[k];
```

A shipment link between cities `i` and `j` is represented by an arc that connects the node `Demand[i]` to node `Supply[j]`:

```
set CITIES;
set LINKS within (CITIES cross CITIES);

param supply {CITIES} >= 0;   # amounts available at cities
param demand {CITIES} >= 0;   # amounts required at cities

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

param cost {LINKS} >= 0;      # shipment costs per ton

param city_cap {CITIES} >= 0; # max throughput at cities
param link_cap {LINKS} >= 0;  # max shipment over links

minimize Total_Cost;

node Supply {k in CITIES}: net_out = supply[k];
node Demand {k in CITIES}: net_in = demand[k];

arc Ship {(i,j) in LINKS} >= 0, <= link_cap[i,j],
   from Demand[i], to Supply[j], obj Total_Cost cost[i,j];

arc Through {k in CITIES} >= 0, <= city_cap[k],
   from Supply[k], to Demand[k];
```

Figure 15-15: Transshipment model with node capacities (`netthru.mod`).

```
arc Ship {(i,j) in LINKS} >= 0, <= link_cap[i,j],
   from Demand[i], to Supply[j], obj Total_Cost cost[i,j];
```

The throughput at city `k` is represented by a new kind of arc, from `Supply[k]` to `Demand[k]`:

```
arc Through {k in cities} >= 0, <= city_cap[k],
   from Supply[k], to Demand[k];
```

The throughput limit is now represented by an upper bound on this arc's flow, rather than by a side constraint, and the network structure of the model is preserved. A complete listing appears in Figure 15-15.

The preceding example exhibits an additional advantage of using the `node` and `arc` declarations when developing a network model. If you use only `node` and `arc` in their simple forms — no variables in the node conditions, and no optional factors in the `from` and `to` phrases — your model is guaranteed to give rise only to pure network linear programs. By contrast, if you use `var` and `subject to`, it is your responsibility to ensure that the resulting linear program has the necessary network structure.

Some of the preceding comments can be extended to ''generalized network'' linear programs in which each variable still figures in at most two constraints, but not necessarily with coefficients of +1 and –1. We have seen examples of generalized networks in the cases where there is a loss of flow or change of units on the arcs. Generalized network LPs do not necessarily have integer optimal solutions, but fast algorithms for them do exist. A solver that promises a ''network'' algorithm may or may not have an extension to generalized networks; check the solver-specific documentation before you make any assumptions.

Bibliography

Ravindra K. Ahuja, Thomas L. Magnanti and James B. Orlin, *Network Flows: Theory, Algorithms, and Applications.* Prentice-Hall (Englewood Cliffs, NJ, 1993).

Dimitri P. Bertsekas *Network Optimization: Continuous and Discrete Models.* Athena Scientific (Princeton, NJ, 1998).

L. R. Ford, Jr. and D. R. Fulkerson, *Flows in Networks.* Princeton University Press (Princeton, NJ, 1962). A highly influential survey of network linear programming and related topics, which stimulated much subsequent study.

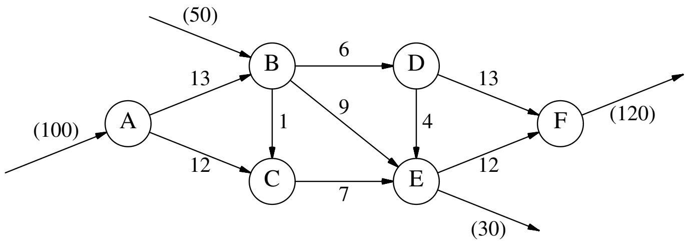
Fred Glover, Darwin Klingman and Nancy V. Phillips, *Network Models in Optimization and their Applications in Practice.* John Wiley & Sons (New York, 1992).

Walter Jacobs, ''The Caterer Problem.'' Naval Research Logistics Quarterly **1** (1954) pp. 154–165. The origin of the network problem described in Exercise 15-8.

Katta G. Murty, *Network Programming.* Prentice-Hall (Englewood Cliffs, NJ, 1992).

Exercises

15-1. The following diagram can be interpreted as representing a network transshipment problem:



The arrows into nodes A and B represent supply in the indicated amounts, 100 and 50; the arrows out of nodes E and F similarly represent demand in the amounts 30 and 120. The remaining arrows indicate shipment possibilities, and the numbers on them are the unit shipping costs. There is a capacity of 80 on every arc.

(a) Solve this problem by developing appropriate data statements to go along with the model of Figure 15-2a.

(b) Check that you get the same solution using the node and arc formulation of Figure 15-10. Does the solver appear to be using the same algorithm as in (a)? Try this comparison with each LP solver available to you.

15-2. Reinterpret the numbers on the arcs *between* nodes, in the diagram of the preceding exercise, to solve the following problems. (Ignore the numbers on the arrows into A and B and on the arrows out of E and F.)

(a) Regarding the numbers on the arcs between nodes as lengths, use a model such as the one in Figure 15-7 to find the shortest path from A to F.

(b) Regarding the numbers on the arcs between nodes as capacities, use a model such as the one in Figure 15-6 to find the maximum flow from A to F.

(c) Generalize the model from (b) so that it can find the maximum flow from any subset of nodes to any other subset of nodes, in some meaningful sense. Use your generalization to find the maximum flow from A and B to E and F.

15-3. Section 4.2 showed how a multiperiod model could be constructed by replicating a static model over time periods, and using inventories to tie the periods together. Consider applying the same approach to the specialized transshipment model of Figure 15-4a.

(a) Construct a multi-week version of Figure 15-4a, with the inventories kept at the distribution centers (the members of `D_CITY`).

(b) Now consider expanding the data of Figure 15-4b for this model. Suppose that the initial inventory is 200 at `NE` and 75 at `SE`, and that the inventory carrying cost per 1000 packages is 0.15 per week at `NE` and 0.12 per week at `SE`. Let the supplies and demands over 5 weeks be as follows:

		Demand				
Week	Supply	BOS	EWR	BWI	ATL	MCO
1	450	50	90	95	50	20
2	450	65	100	105	50	20
3	400	70	100	110	50	25
4	250	70	110	120	50	40
5	325	80	115	120	55	45

Leave the cost and capacity data unchanged, the same in all weeks. Develop an appropriate data file for this situation, and solve the multi-week problem.

(c) The multi-week model in this case can be viewed as a pure network model, with arcs representing inventories as well as shipments. To show that this is the case, reformulate the model from (a) using only `node` and `arc` declarations for the constraints and variables.

15-4. For each of the following network problems, construct a data file that permits it to be solved using the general transshipment model of Figure 15-2a.

(a) The transportation problem of Figure 3-1.

(b) The assignment problem of Figure 3-2.

(c) The maximum flow problem of Figure 15-6.

(d) The shortest path problem of Figure 15-7.

15-5. Reformulate each of the following network models using `node` and `arc` declarations as much as possible:

(a) The transportation model of Figure 6-2a.

(b) The shortest path model of Figure 15-7.

(c) The production/transportation model of Figure 4-6.

(d) The multicommodity transportation model of Figure 6-5.

15-6. The professor in charge of an industrial engineering design course is faced with the problem of assigning 28 students to eight projects. Each student must be assigned to one project, and each project group must have 3 or 4 students. The students have been asked to rank the projects, with 1 being the best ranking and higher numbers representing lower rankings.

(a) Formulate an algebraic assignment model, using `var` and `subject to` declarations, for this problem.

(b) Solve the assignment problem for the following table of rankings:

	A	ED	EZ	G	H1	H2	RB	SC
Allen	1	3	4	7	7	5	2	6
Black	6	4	2	5	5	7	1	3
Chung	6	2	3	1	1	7	5	4
Clark	7	6	1	2	2	3	5	4
Conners	7	6	1	3	3	4	5	2
Cumming	6	7	4	2	2	3	5	1
Demming	2	5	4	6	6	1	3	7
Eng	4	7	2	1	1	6	3	5
Farmer	7	6	5	2	2	1	3	4
Forest	6	7	2	5	5	1	3	4
Goodman	7	6	2	4	4	5	1	3
Harris	4	7	5	3	3	1	2	6
Holmes	6	7	4	2	2	3	5	1
Johnson	7	2	4	6	6	5	3	1
Knorr	7	4	1	2	2	5	6	3
Manheim	4	7	2	1	1	3	6	5
Morris	7	5	4	6	6	3	1	2
Nathan	4	7	5	6	6	3	1	2
Neuman	7	5	4	6	6	3	1	2
Patrick	1	7	5	4	4	2	3	6
Rollins	6	2	3	1	1	7	5	4
Schuman	4	7	3	5	5	1	2	6
Silver	4	7	3	1	1	2	5	6
Stein	6	4	2	5	5	7	1	3
Stock	5	2	1	6	6	7	4	3
Truman	6	3	2	7	7	5	1	4
Wolman	6	7	4	2	2	3	5	1
Young	1	3	4	7	7	6	2	5

How many students are assigned second or third choice?

(c) Some of the projects are harder than others to reach without a car. Thus it is desirable that at least a certain number of students assigned to each project must have a car; the numbers vary by project as follows:

A	ED	EZ	G	H1	H2	RB	SC
1	0	0	2	2	2	1	1

The students who have cars are:

Chung	Eng	Manheim	Nathan	Rollins
Demming	Holmes	Morris	Patrick	Young

Modify the model to add this car constraint, and solve the problem again. How many more students than before must be assigned second or third choice?

(d) Your formulation in (c) can be viewed as a transportation model with side constraints. By defining appropriate network nodes and arcs, reformulate it as a ''pure'' network flow model, as discussed in Section 15.5. Write the formulation in AMPL using only `node` and `arc` declarations for the constraints and variables. Solve with the same data as in (c), to show that the optimal value is the same.

15-7. To manage its excess cash over the next 12 months, a company may purchase 1-month, 2-month or 3-month certificates of deposit from any of several different banks. The current cash on hand and amounts invested are known, while the company must estimate the cash receipts and expenditures for each month, and the returns on the different certificates.

The company's problem is to determine the best investment strategy, subject to cash requirements. (As a practical matter, the company would use the first month of the optimal solution as a guide to its current purchases, and then re-solve with updated estimates at the beginning of the next month.)

(a) Draw a network diagram for this situation. Show each month as a node, and the investments, receipts and expenditures as arcs.

(b) Formulate the relevant optimization problem as an AMPL model, using `node` and `arc` declarations. Assume that any cash from previously-purchased certificates coming due in the early months is included in data for the receipts.

There is more than one way to describe the objective function for this model. Explain your choice.

(c) Suppose that the company's estimated receipts and expenses (in thousands of dollars) over the next 12 months are as follows:

	receipt	expense
1	3200	200
2	3600	200
3	3100	400
4	1000	800
5	1000	2100
6	1000	4500
7	1200	3300
8	1200	1800
9	1200	600
10	1500	200
11	1800	200
12	1900	200

The two banks competing for the business are estimating the following rates of return for the next 12 months:

CIT:	1	2	3	NBD:	1	2	3
1	0.00433	0.01067	0.01988	1	0.00425	0.01067	0.02013
2	0.00437	0.01075	0.02000	2	0.00429	0.01075	0.02025
3	0.00442	0.01083	0.02013	3	0.00433	0.01083	0.02063
4	0.00446	0.01092	0.02038	4	0.00437	0.01092	0.02088
5	0.00450	0.01100	0.02050	5	0.00442	0.01100	0.02100
6	0.00458	0.01125	0.02088	6	0.00450	0.01125	0.02138
7	0.00467	0.01142	0.02113	7	0.00458	0.01142	0.02162
8	0.00487	0.01183	0.02187	8	0.00479	0.01183	0.02212
9	0.00500	0.01217	0.02237	9	0.00492	0.01217	0.02262
10	0.00500	0.01217	0.02250	10	0.00492	0.01217	0.02275
11	0.00492	0.01217	0.02250	11	0.00483	0.01233	0.02275
12	0.00483	0.01217	0.02275	12	0.00475	0.01250	0.02312

Construct an appropriate data file, and solve the resulting linear program. Use `display` to produce a summary of the indicated purchases.

(d) Company policy prohibits investing more than 70% of its cash in new certificates of any one bank in any month. Devise a side constraint on the model from (b) to impose this restriction.

Again solve the resulting linear program, and summarize the indicated purchases. How much income is lost due to the restrictive policy?

15-8. A caterer has booked dinners for the next `T` days, and has as a result a requirement for a certain number of napkins each day. He has a certain initial stock of napkins, and can buy new ones each day at a certain price. In addition, used napkins can be laundered either at a slow service that takes 4 days, or at a faster but more expensive service that takes 2 days. The caterer's problem is to find the most economical combination of purchase and laundering that will meet the forthcoming demand.

(a) It is not hard to see that the decision variables for this problem should be something like the following:

`Buy[t]`	clean napkins bought for day `t`
`Carry[t]`	clean napkins still on hand at the end of day `t`
`Wash2[t]`	used napkins sent to the fast laundry after day `t`
`Wash4[t]`	used napkins sent to the slow laundry after day `t`
`Trash[t]`	used napkins discarded after day `t`

There are two collections of constraints on these variables, which can be described as follows:

– The number of clean napkins acquired through purchase, carryover and laundering on day `t` must equal the number sent to laundering, discarded or carried over after day `t`.

– The number of used napkins laundered or discarded after day `t` must equal the number that were required for that day's catering.

Formulate an AMPL linear programming model for this problem.

(b) Formulate an alternative network linear programming model for this problem. Write it in AMPL using node and arc declarations.

(c) The ''caterer problem'' was introduced in a 1954 paper by Walter Jacobs of the U.S. Air Force. Although it has been presented in countless books on linear and network programming, it does not seem to have ever been used by any caterer. In what application do you suppose it really originated?

(d) Since this is an artificial problem, you might as well make up your own data for it. Use your data to check that the formulations in (a) and (b) give the same optimal value.