Expressing Special Structures in an Algebraic Modeling Language for Mathematical Programming

Robert Fourer

Northwestern University
Evanston, Illinois 60201

David M. Gay

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

PUBLISHED VERSION

Robert Fourer and David M. Gay, "Expressing Special Structures in an Algebraic Modeling Language for Mathematical Programming." *ORSA Journal on Computing* **7** (1995) 166–190.

ABSTRACT

A knowledge of the presence of certain special structures can be advantageous in both the formulation and solution of linear programming problems. Thus it is desirable that linear programming software offer the option of specifying such structures explicitly. As a step in this direction, we describe extensions to an algebraic modeling language that encompass piecewise-linear, network and related structures. Our emphasis is on the modeling considerations that motivate these extensions, and on the design issues that arise in integrating these extensions with the general-purpose features of the language. We observe that our extensions sometimes make models faster to translate as well as to solve, and that they permit a "column-wise" formulation of the constraints as an alternative to the "row-wise" formulation most often associated with algebraic languages.

1. Introduction

Certain special structures play a valuable role in both the formulation and the solution of linear programs. When these structures are recognized explicitly, an LP model can be formulated in a simpler and more natural way. When these structures are communicated to correspondingly specialized algorithms, the resulting LP problems can also be solved faster.

The benefits of special structures are not always so easy to realize, however. For structured linear programs of typical size and complexity, some kind of computer software is invariably employed to bridge the gap between the modeler's conception and the algorithm's requirements. To take advantage of a structure, this software must be able to recognize the structure's presence in its input from the modeler, and must be able to communicate the structure in its output to the algorithm.

For any particular application, the necessary software can be provided by writing a "matrix generation" program that is specialized to the structure at hand; but the difficulties of debugging and maintaining such a program are well known [16]. As an alternative, numerous software systems have been specifically designed to read a modeler's LP formulation and to convert it automatically for solution. Most of these systems are intended for general-purpose linear programming, however, and so lack any facilities for dealing with structures. Other systems that do recognize structure are highly specialized to particular structures or applications.

We are thus led to consider the possibility that a general-purpose linear programming system might offer the *option* of specifying certain structures. The structure features of such a system would be implemented as extensions to the basic facilities for linear programming. Ideally, these extensions would be independent, in the sense that a user concerned with a certain structure would only need to learn about the extensions supporting that structure. At the same time, all extensions would be tightly integrated with the rest of the system, so that they would not burden users with too much in the way of new syntax or concepts.

As an example of how general-purpose LP software can accommodate special structures, this paper presents extensions to the AMPL mathematical programming language and translator for the purpose of handling network, piecewise-linear and related structures. The emphasis is on the modeling considerations that motivate these extensions, and on the design issues that must be resolved in adding these extensions to the general-purpose features of the language. Close attention to design is seen to be essential in providing a syntax that is natural to use, yet that achieves the goals of independence and integration stated above.

The remainder of this introduction presents a brief self-contained summary of the essential features of AMPL, and outlines the rest of the paper.

1.1 Essentials of AMPL

AMPL is an *algebraic modeling language:* a computer-readable language for describing objective functions and constraints in the kind of algebraic notation that many modelers use. Languages of this kind, such as GAMS [5, 6] and MGG [40], were first developed in the 1970s, and have seen increasingly wide use in recent

$\mathcal{R}$, a set of raw materials
$\mathcal{P}$, a set of products
	`set R;  # raw materials`
	`set P;  # products`

a_{ij}, $i \in \mathcal{R}$, $j \in \mathcal{P}$:
input-output coefficients
	`param a {R,P} >= 0;`
	`   # input-output coefficients`

b_i, $i \in \mathcal{R}$: units available	`param b {R} > 0;  # units available`

c_j, $j \in \mathcal{P}$: profit per unit
u_j, $j \in \mathcal{P}$: production limit
	`param c {P} > 0;  # profit per unit`
	`param u {P} > 0;  # production limit`

x_j, $j \in \mathcal{P}$, with $0 \le x_j \le u_j$:
units of j produced
	`var x {j in P} >= 0, <= u[j];`
	`  # units of j produced`

Maximize $\sum_{j \in \mathcal{P}} c_j x_j$: total profit	`maximize tot: sum {j in P} c[j] * x[j];`
	`  # total profit`

Subject to $\sum_{j \in \mathcal{P}} a_{ij} x_j \le b_i$, $i \in \mathcal{R}$:
limited availability of material
	`subj to supply {i in R}:`
	`  sum {j in P} a[i,j] * x[j] <= b[i];`
	`  # limited availability of material`

Figure 1–1. *A simple linear program in algebraic notation.* At left is a traditional informal algebraic description; at right is an equivalent in the AMPL modeling language.

years. AMPL is notable for its particularly natural and general syntax, and for its variety of set and indexing expressions.

The AMPL language employs a standard computer character set, and permits none of the ambiguity that is tolerated in informal algebraic notation. Thus a term such as $\sum_{j \in \mathcal{P}} a_{ij} x_j$ becomes `sum {j in P} a[i,j] * x[j]`, in which the subscripts of `a` must be separated by a comma, and the multiplication is indicated by an explicit operator. Figure 1–1 displays a linear programming model side-by-side with its equivalent in AMPL. Even in this simple case one can see the essential features of the five basic kinds of AMPL declarations: `set` (index set), `param` (parameter), `var` (variable), `maximize` or `minimize` (objective), and `subject to` (constraint).

The design of AMPL enforces a distinction between an optimization *model*, such as is seen in Figure 1–1, and the *instances* of that model that correspond to particular choices of set and parameter data. To employ AMPL in solving a particular instance, one must supply both a model and a data description, which are processed by special software—the AMPL *translator*—to produce the information required by a solver. Since the extensions described in this paper affect only the way that a model is formulated, we do not show the data description in our examples.

An earlier paper [19] has introduced the linear programming features of AMPL. Much of the same material, together with four extended examples, is presented in a technical report [18] available from the authors. The Scientific Press publishes a book [20] and software that support the complete AMPL language, including all features described in this paper, as well as extensions for nonlinear and integer programming.

1.2 Outline

Section 2 considers the problem of specifying terms that are piecewise-linear in individual variables. With some effort, a modeler may transform piecewise-linear terms to linear ones by substituting special structures of auxiliary variables or constraints. We propose instead a new syntax that permits piecewise-linearities to be described directly in an AMPL model, while appropriate transformations are provided automatically by the AMPL translator.

Section 3 deals with the specification of network flow linear programs, which have a particularly distinctive constraint structure that relates directly to the arrangement of network nodes and arcs. We examine the drawbacks of specifying network constraints algebraically, and propose an alternative in which `node` and `arc` declarations specify the network directly.

Section 4 extends our analysis to more general kinds of network structures in LPs. We show that slight further extensions of AMPL suffice to permit a natural description of the network structure in three common cases: weighted flows, side constraints, and side variables.

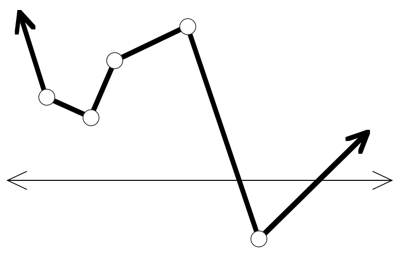
The approach of the previous two sections is applied in Section 5 to another special structure, that of set covering problems. We then observe that the AMPL extensions for both networks and set covering can be regarded as cases in which the constraint coefficient matrix is specified "by column" rather than "by row". This leads to our final extension, which permits column-wise specification of any constraint structure.

Our concluding comments in Section 6 address some of the broader issues raised by this paper. AMPL's structure features allow people to specify a model in more than one way, by use of one or another extension; we argue that this sort of redundancy is a virtue of the design. AMPL's extensions for column-wise constraint specification go against the inherent row-wise orientation of algebraic languages; we point out that many of the strengths of an algebraic modeling language are realized regardless of how the constraints are specified.

We present statistics to show that the extensions described in this paper often allow the AMPL translator to work faster. We also argue that, when structures are specified explicitly in AMPL, certain preprocessing steps of structure-exploiting algorithms can be simplified or eliminated. Finally, we comment on several promising areas of related research, including algebraic language extensions for additional structures, and graphical interfaces for network optimization.

2. Piecewise-Linear Programming Models

A continuous function of one variable is piecewise-linear (P-L) if its domain can be partitioned into intervals on which the function is linear. The P-L functions that we consider here are defined by a finite number of intervals, but may otherwise be quite arbitrary:



The linear piece on each interval can be defined in the customary way by its *slope* and by its value at zero, or *intercept.* The boundaries of the intervals, where the slope changes, are the P-L function's *breakpoints.* For present purposes, we define a *piecewise-linear program* (or P-LP) to be the generalization of a linear program that results from using some of these P-L terms in the objective and constraints, in addition to the usual linear terms.

Piecewise-linear programs have a long history of application, dating back to the 1950's [7, 12]. They are widely used for a variety of purposes in large-scale mathematical programming: for modeling bidirectional flows, inventories/backorders, tension/compression and other "reversible" activities; for approximating nonlinearities, especially when the relevant nonlinear functions are only rough estimates to begin with; and for penalizing the deviations of objectives or constraints from desired levels.

Any P-LP can be converted, by any of several transformations, to an equivalent mathematical program in which the P-L terms are replaced by linear terms. For every linear piece, such a transformation adds a few constraints or variables (or both), including a zero-one integer variable except in special cases. The resulting linear or mixed-integer formulation is then readily expressed and solved with the help of an algebraic modeling language such as AMPL. Yet although this sort of transformation approach is sufficient in a mathematical sense, it fails to deal with two important practical considerations in piecewise-linear programming:

- People conceive of a model in its original P-L formulation, and would prefer to work with it in that form.
- Some optimization packages can employ a knowledge of the original P-L formulation to solve P-LPs much more efficiently than they can solve the equivalent linear or mixed-integer programs.

To address these issues, the AMPL language must provide a syntax for expressing P-L terms explicitly.

Whereas AMPL's linear expressions are based on standard algebraic notation, there is no commonly accepted terminology for piecewise-linear functions. We have had to invent a terminology, which is described in this section. We first explain the underlying motivation for our design, and then describe the specifics. Finally, we discuss the issues involved in translating piecewise-linear programs for various solvers.

2.1 Design requirements

Our goal in this work was to design an AMPL expression that could concisely and clearly describe any piecewise-linear function of one variable. Before we could address the details of the syntax, we had to decide how AMPL would specify a P-L function unambiguously. Certainly we wanted to use no more information than necessary, but there are several different minimal collections of numbers that can define the same P-L function:

1. A list of breakpoints, and either
   (a) the change in slope at each breakpoint, or
   (b) the value of the function at each breakpoint.
2. A list of slopes, and either
   (a) the distance between the breakpoints bounding each slope, or
   (b) the value of the intercept associated with each slope.
3. A list of breakpoints and a list of slopes.

In the case of (1a), (2a) or (3), these values actually only define the P-L function up to an additive constant; the value at one arbitrary point must be given in order to fix the function. Also for (1a) it is necessary to give the value of one particular slope, and for (2a) the value of one particular breakpoint.

Any of these representations might correspond most closely to the data that are available for a given application. Thus from the standpoint of convenience and clarity we might prefer to make all five representations available in the AMPL language. From the standpoint of complexity, on the other hand, we are reluctant to add any new representations at all. Every new syntax interacts with existing features of AMPL, often raising thorny design issues that involve all aspects of language translation from parsing to coefficient generation; even when these issues can be resolved, any new feature results in a new collection of rules that users must learn. After considering these tradeoffs between convenience and complexity, we decided to initially add just one piecewise-linear representation to the AMPL language.

The syntax that we now proceed to describe is based on representation (3) above. It explicitly specifies separate slope and breakpoint sequences, with conventions to deal with the fine points: which slopes surround which breakpoints, and which function is intended among all those differing by an additive constant. We find this representation to be the clearest and most convenient for expressing common piecewise-linear terms of a few pieces, such as those employed in modeling penalties and reversible activities. Other representations may be preferable for expressing approximations to nonlinear functions, but representation (3) is reasonably convenient for that purpose as well.

2.2 Examples

A piecewise-linear function is written in AMPL by use of an expression of the following form:

`<<` *breakpoints*; *slopes* `>>` *variable*

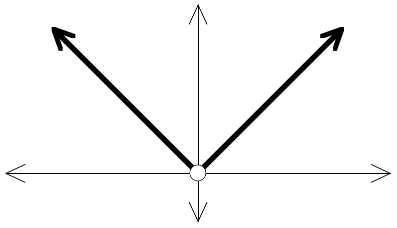
The components denoted *breakpoints* and *slopes* are the defining breakpoint and slope lists, whose syntax is to be described shortly. They are enclosed in `<<`... `>>` so that they comprise an entity clearly different from any of the other algebraic constructs in AMPL. The entire entity is followed by the variable to which it applies; one can think of the entity as "multiplying" the variable by several slopes, rather than by just the one slope of a linear term.

The breakpoints and slopes are given, in the simplest case, by comma-separated lists of values. AMPL adopts the conventions that

- the first slope refers to the linear piece between $-\infty$ and the first breakpoint;
- the last slope refers to the linear piece between the last breakpoint and $+\infty$; and
- the function's value at zero is zero.

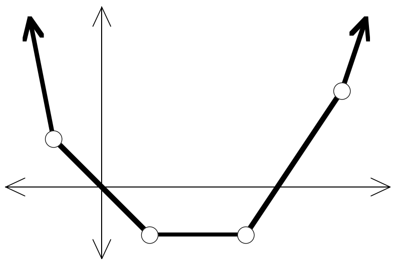
Thus, for example, the absolute value function of an AMPL variable `x[j]` is written

```
<<0; -1, 1>> x[j]
```



since the single breakpoint is at zero, and the slopes are -1 to the left and $+1$ to the right. A more complicated instance is expressed in the same way:

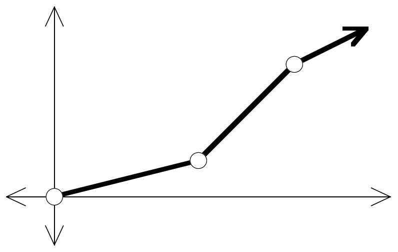
```
<<-1,1,3,5; -5,-1,0,1.5,3>> x[j]
```



In an objective function, this term would tend to penalize deviation of `x[j]` from the interval between 1 and 3.

Sometimes it is appealing to regard a lower bound on a variable as its first breakpoint, or an upper bound on a variable as its last breakpoint. For example, given the AMPL declarations `set S` and `var x {j in S} >= 0`, an increasing cost function of the variable `x[j]` can be represented as

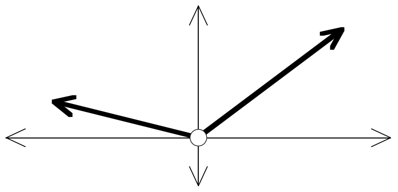
```
<<3,5; 0.25,1.00,0.50>> x[j]
```



where the first “breakpoint” of zero is really just the lower bound on `x[j]`. We did consider allowing the first or last item in a breakpoint list to represent a bound; but then, in cases of just one bound as above, there would be an equal number of breakpoints and slopes, so that an additional convention would be necessary to indicate whether the first breakpoint was intended to come before or after the first slope. We chose instead to keep the syntax simple by requiring bounds to be defined as part of a variable's declaration (or possibly by means of explicit constraints) while only “interior” breakpoints are listed in a P-L expression.

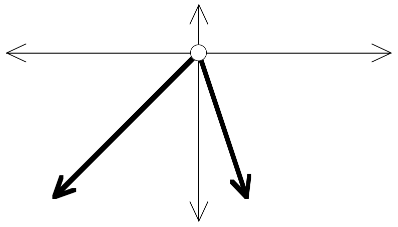
Multiplication of a P-L term has the effect of multiplying all its slopes. Thus,

```
0.25 * <<0; -1,3>> x[j]
```



is the same as the convex function `<<0; -0.25,0.75>> x[j]`. On the other hand, multiplication by a negative number,

```
-1 * <<0; -1,3>> x[j]
```



gives a concave function that could just as well be written `-<<0; -1,3>> x[j]` or `<<0; 1,-3>> x[j]`.

The slopes and breakpoints need not be literal numbers as in the above examples. They could just as well be any AMPL expressions. However, these examples do exhibit a fixed number of slopes and breakpoints. This is no limitation when the terms are simple functions such as absolute values; but in other cases, such as where the terms represent penalties or convex increasing costs, it can be desirable to let the number of linear pieces be itself a function of the data. AMPL already provides for data-dependent collections of many other kinds, by allowing model entities to be indexed over sets. Thus it is natural to allow indexed slope and breakpoint series as well.

As an example, we could let `npce` be an AMPL parameter that specifies the number of linear pieces in a piecewise-linear function of `x[j]`. Then we could define parameter arrays such that `bkp[k]` and `slp[k]` are the `k`th breakpoint and slope values, respectively:

```
param npce integer > 1;
param bkp {1..npce-1};
param slp {1..npce};
```

Our expression for a piecewise-linear term having these breakpoints and slopes is as follows:

```
<<{k in 1..npce-1} bkp[k]; {k in 1..npce} slp[k]>> x[j]
```

The construct `{k in 1..npce-1}` is an AMPL indexing expression, such as might be used elsewhere in the model to declare an indexed collections of parameters or to describe an indexed sum. Here it serves to indicate that the breakpoints are `bkp[1]`, `bkp[2]`, . . . , `bkp[npce-1]`; the number of breakpoints clearly depends on the value for `npce` that is supplied as data. Analogous comments apply to the slopes.

The indexed breakpoint and slope notation is readily applied to more intricate cases, in which P-L functions of the variables `x[j]` may have different values of the breakpoints and slopes for each `j`, or even different numbers of breakpoints and slopes for each `j`. As an example, suppose that the variables `x[j]` are indexed over some set `S`; we index `npce` over `S`, and index `bkp` and `slp` over both `S` and the number of pieces:

```
set S;
var x {S};

param npce {S} integer > 1;
param bkp {j in S, 1..npce[j]-1};
param slp {j in S, 1..npce[j]};

minimize cost:
     sum {j in S} <<{k in 1..npce[j]-1} bkp[j,k];
                    {k in 1..npce[j]} slp[j,k]>> x[j];
```

In the objective, the term for `x[j]` has `npce[j]` pieces. The breakpoints in order are `bkp[j,1]`, . . . , `bkp[j,npce[j]-1]`, and the corresponding slopes are `slp[j,1]`, . . . , `slp[j,npce[j]]`.

A "piecewise-linear" function whose form depends on the data may turn out to have only one piece, in which case AMPL properly interprets it as a linear function. For instance, in the preceding illustration, if `npce[j]` takes the value 1 for some `j` then the set `1..npce[j]-1` is empty, and no breakpoints `bkp[j,k]` are defined. The P-L term involving `x[j]` is interpreted as a linear term having the one slope `slp[j,1]` everywhere.

Two complete examples are exhibited in the appendices. STRUC is a structural design model that uses simple two-piece terms (Appendix C). SCORE is an equation-fitting model for credit scoring, in which the number and value of the breakpoints and slopes depend in various ways on the problem data (Appendix B).

2.3 General form

The general AMPL syntax for a piecewise-linear term is

> `<<` *bkpt1*`,` *bkpt2*`,` *...*`;` *slope1*`,` *slope2*`,` *...* `>>` *variable*

where each of the items *bkpt1*, *bkpt2*, ... and *slope1*, *slope2*, ... has one of the forms

> *expr*
> `{`*indexing*`}` *expr*

The *expr* must be an AMPL expression that evaluates to a number. The optional *indexing* must specify an ordered AMPL set: either a set entirely of numbers (as in our examples) or a set that has been defined to be ordered (by specifying `ordered` or `circular` in its declaration). Any indexed item is expanded to a sequence of numbers by taking each member of the set in order, substituting it into *expr*, and evaluating the result. After all expansions are made, the breakpoints must be nondecreasing, and the number of slopes must be one more than the number of breakpoints.

Not all piecewise-linear functions of interest are zero at zero, as the AMPL convention assumes. If it is desired to have a P-L function evaluate to other than zero at zero, one may simply add a constant term, observing that

> *const-expr* `+ <<` *bkpt1*`,` *bkpt2*`,` *...*`;` *slope1*`,` *slope2*`,` *...* `>>` *variable*

has the value of *const-expr* at zero. A P-L function's value at zero is sometimes of no particular significance, however, especially when the variable is bounded away from zero. The function's value may instead be known to be zero at the first or last breakpoint, or (for penalty functions) at a breakpoint where the value is smallest; as a result, the modeler may be forced to construct a messy *const-expr* to adjust the function to the proper level. For convenience in these situations, AMPL also recognizes the more general notation

> `<<` *bkpt1*`,` *bkpt2*`,` *...*`;` *slope1*`,` *slope2*`,` *...* `>> (`*variable*`,` *const-expr*`)`

which is interpreted as before, except that *const-expr* gives a value of *variable* at which the function is zero.

2.4 Translation issues

An AMPL user typically issues commands to read a model and data from files, and to choose a *solver* which may be an implementation of one algorithm or a package of algorithms such as MINOS [38], CPLEX [9] or OSL [29]. Upon receiving a subsequent `solve` command, the AMPL translator generates a particular instance of the model, writes it to a file in a compact format, and passes control to a *driver* for the chosen solver. The driver reads the newly-created file, translates the instance to the solver's internal data structure, and invokes an appropriate algorithm. At the completion of the algorithm, the driver also translates the optimal solution back to a file that AMPL can read.

The AMPL translator can deal with P-L functions most straightforwardly by writing a description of each such function's slope and breakpoint structure to the generated file. This would be the right approach for a solver such as CPLP [14, 23] that can handle some kinds of P-L functions directly, through the use of an algorithm that has been adapted to solve piecewise-linear problems in an efficient way.

For the many solvers that cannot operate directly on P-L functions, some kind of transformation to a linear program is necessary. Because this is such a common situation, we have built the transformation logic directly into the AMPL translator, rather than duplicating it in the drivers for numerous solvers. The transformation is turned on or off by use of AMPL's `option` command; it is on by default in the current implementation.

A piecewise-linear program can be transformed to an equivalent linear program, provided that certain conditions are met [1]. In the objective, all P-L terms must be convex (if minimizing) or concave (if maximizing). In the constraints, P-L terms are limited to the following situations:

- convex on the left-hand side of a $\leq$ constraint, or the right-hand side of a $\geq$ constraint; or
- concave on the left-hand side of a $\geq$ constraint, or the right-hand side of a $\leq$ constraint.

A P-L term is readily identified as convex or concave through its slope sequence, which is increasing for a convex function and decreasing for a concave one. Of the several available transformations (surveyed in [17]), AMPL employs the so-called Δ-form [10], which replaces each P-L term by a linear combination of bounded auxiliary variables. The coefficients of these variables correspond to slopes, and their bounds correspond to differences between breakpoints.

When the above conditions are not satisfied, the transformation must be to some kind of integer linear program. For this purpose it is most convenient to use the Λ-form transformation [12], which replaces each P-L term by a convex combination of the function values at the breakpoints. To describe the convex combination, this transformation introduces a nonnegative auxiliary variable for each breakpoint, and a constraint that these variables sum to 1; in addition, these variables must be explicitly constrained so that at most two of them, corresponding to adjacent breakpoints, are positive in any feasible solution. Many current solvers for integer programming impose this adjacency constraint directly and efficiently, by treating the auxiliary variables as a "special ordered set of type 2" [2, 42].

To support solvers that do not implement special ordered sets, AMPL offers the option of transforming a P-LP to an equivalent mixed-integer linear program. A transformation for this purpose must introduce a zero-one integer variable as well as a continuous linear variable corresponding to each piece, and it must add constraints to enforce the appropriate relationships between adjacent variables. It can be based on either the Δ-form [37] or the Λ-form [11]; AMPL employs the latter.

Because convexity and concavity depend upon an appropriate ordering of the slopes, any error in the slope data may cause these properties to be lost. As a result, AMPL may generate a difficult integer program when a transformation to a simpler linear program is expected. To prevent occurrences of this kind, AMPL's `check` statement can be used to verify that the slopes are in the expected order; in the case of the preceding example, a `check` for convexity could be written as

```
check {j in S, k in 1..npce[j]-1}: slp[j,k] <= slp[j,k+1];
```

It is not necessary to check the breakpoints in this way, since it is an AMPL error to specify them in other than an increasing order.

AMPL's transformation strategy enables it to successfully address the two practical considerations that were cited at the beginning of this section. When a model uses AMPL's notation for piecewise-linear terms, the AMPL translator generates problem instances in the form best suited to the chosen solver. Subsequently, when the optimal solution is passed back from the solver, AMPL undoes the effects of any transformation, so that people can work with the solution in terms of the original variables and constraints.

3. Network Flow Models

The linear network flow model is one of the best known and most important in linear programming. In its purest form, it is based on the sort of directed network exhibited in Figure 3–1, in which nodes are connected by "one-way" arcs. Some kind of entity is imagined to flow, literally or figuratively, from node to node along the arcs, so that the decision variables are the levels of these flows. The objective is to minimize or maximize some linear function of the flows; the constraints are simple bounds on the flows, and balances of flow at the nodes. Details are supplied in Chvátal [8] and many other linear programming texts.

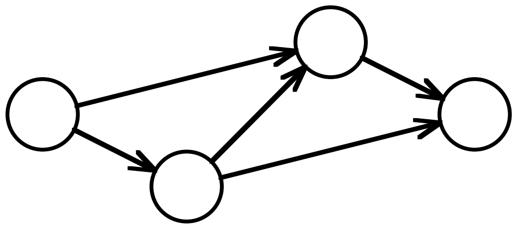


Figure 3–1. *Diagram of a directed network,* in which circles represent the nodes, and arrows the arcs. For the generic model introduced in the text, a cost and bounds on the flow are associated with each arc, and a required net flow is associated with each node.

Network flow models are important because they have many varied applications, and because they are particularly easy to solve by means of specialized algorithms. Although the general-purpose features of AMPL suffice to express virtually any of these models, the resulting constraint expressions are often not as natural as we would like. Indeed, it can be hard to tell whether a model's algebraic constraints, taken together, represent a valid collection of flow balances on a network. As a result, when specialized network optimization software is used, additional checking of the model must be made outside of the language; Zenios [44] describes a comparable arrangement in conjunction with the GAMS algebraic modeling system [6].

To overcome the awkwardness of expressing network flow models in AMPL, we have added the option of declaring nodes and arcs more explicitly. We first explain the source of the awkwardness at greater length below, then detail the design of the network extensions, and finally comment on translation issues.

Other, more graphically oriented systems for expressing network models are compared to AMPL in the concluding remarks of Section 6.

3.1 Design requirements

The issues faced by our design can be illustrated through a simple generic model of minimum-cost flows on a directed network. The structure of the network is determined by a set $\mathcal{N}$ of nodes, and a set $\mathcal{A} \subseteq \mathcal{N} \times \mathcal{N}$ of arcs, such that $(i, j) \in \mathcal{A}$ if and only if there is an arc from node i to node j. The objective is to minimize the total cost of the flows x_{ij} along all the arcs,

$$\sum_{(i,j)\in\mathcal{A}} c_{ij} x_{ij},$$

subject to bounds on the flows along the arcs,

$$l_{ij} \le x_{ij} \le u_{ij}, \quad (i,j) \in \mathcal{A},$$

and balance of flow at the nodes. The latter constraints have the form

$$\sum(\text{flows out of } i) - \sum(\text{flows into } i) = b_i, \quad i \in \mathcal{N},$$

where b_i is the amount of flow produced at node i if $b_i \ge 0$, or $-b_i$ is the amount consumed at node i if $b_i \le 0$. (The special case of $b_i = 0$ defines a pure "transshipment" node at which flow out equals flow in.)

The flow balance constraints pose the main difficulty of this model for algebraic notation. To be precise, they should be written as

$$\sum_{j \in \mathcal{N}:(i,j)\in\mathcal{A}} x_{ij} - \sum_{j \in \mathcal{N}:(j,i)\in\mathcal{A}} x_{ji} = b_i, \quad i \in \mathcal{N},$$

but the required indexing expressions are then so long as to be awkward. Instead we can abbreviate them, writing

$$\sum_{(i,j)\in\mathcal{A}} x_{ij} - \sum_{(j,i)\in\mathcal{A}} x_{ji} = b_i, \quad i \in \mathcal{N}.$$

Since this constraint is being defined for each i, it makes sense to interpret $(i,j) \in \mathcal{A}$ in this context as the set of all arcs out of i, and $(j,i) \in \mathcal{A}$ as the set of all arcs into i.

We will refer to the AMPL version of this model as TRANSSHIP. Its data can be declared as

```
set N;
set A within N cross N;

param b {N};
param c {A} >= 0;

param l {A} >= 0;
param u {(i,j) in A} >= l[i,j];
```

The variables can then be defined, along with their bounds, by

```
var Flow {(i,j) in A} >= l[i,j], <= u[i,j];
```

We name the variables `Flow` rather than `x`, to make the model a little clearer, and to help with the subsequent exposition. We can now easily write the objective function as

```
minimize Total_Cost: sum {(i,j) in A} c[i,j] * Flow[i,j];
```

Finally, the flow balance constraints are expressed by

```
subject to Balance {i in N}:
     sum {j in N: (i,j) in A} Flow[i,j] -
     sum {j in N: (j,i) in A} Flow[j,i] = b[i];
```

or, more concisely,

```
subject to Balance {i in N}:
     sum {(i,j) in A} Flow[i,j] - sum {(j,i) in A} Flow[j,i] = b[i];
```

These representations of the constraints are direct transcriptions of the two algebraic alternatives formulated previously.

Whether in the language of mathematics or of AMPL, the expressions for the flow balance constraint are fundamentally unsatisfying. The idea of constraining "flow out minus flow in" is intuitively obvious, yet its algebraic equivalents are not so quickly comprehended. The problem is only worse for models of realistic complexity; as two examples, we give a planning and distribution model (DIST) in Appendix A and a scheduling model (TRAIN) in Appendix D.

Our difficulty arises from the fact that standard algebra can directly describe only variables and constraints, while the existence of nodes and arcs is an implicit property of the formulation. People tend to approach network flow problems in just 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, we have designed an extension that permits AMPL to express the relevant network concepts directly.

3.2 Design specifics: `node` declarations

The network extensions to AMPL comprise two new kinds of declaration, `arc` and `node`, which take the place of the `var` and `subject to` declarations in an algebraic constraint formulation.

AMPL's `node` declarations name the nodes of a network, and characterize the flow balance constraints at the nodes. The `arc` declarations then name and define the arcs, by specifying the nodes that arcs connect, and by providing a variety of optional information associated with arcs. Since at least some of the nodes must be declared before any of the arcs, we begin by describing our design for the `node` declarations, and then introduce the `arc` declarations. A completed statement of the TRANSSHIP example using `node` and `arc` is exhibited in Figure 3–2, with the equivalent example using `var` and `subject to` placed alongside for comparison.

A `node` declaration names and defines a network node, or an indexed collection of nodes. Thus the TRANSSHIP example declares

```
node Balance {i in N}: net_out = b[i];
```

to define a node `Balance[i]` for each member `i` of the set `N`. The optional part of the declaration after the colon is a description of the flow balance constraint, as we will explain shortly.

One might informally think of `N` above as actually being the set of nodes; but in AMPL it is just a set of objects over which the collection `Balance` of nodes happens to be defined. Such a distinction permits a collection of nodes to be indexed over any set that can be described in AMPL, such as a set of pairs in the TRAIN model of Appendix D:

```
### NODE-ARC FORMULATION ###           ### ALGEBRAIC FORMULATION ###

set N;                                 set N;
set A within N cross N;                set A within N cross N;

param b {N};                           param b {N};
param c {A} >= 0;                      param c {A} >= 0;

param l {A} >= 0;                      param l {A} >= 0;
param u {(i,j) in A} >= l[i,j];        param u {(i,j) in A} >= l[i,j];

                                       var Flow {(i,j) in A}
                                         >= l[i,j], <= u[i,j];

minimize Total_Cost;                   minimize Total_Cost:
                                         sum {(i,j) in A} c[i,j] * Flow[i,j];

node Balance {i in N}:                 subject to Balance {i in N}:
  net_out = b[i];                        sum {(i,j) in A} Flow[i,j] -
                                         sum {(j,i) in A} Flow[j,i] = b[i];
arc Flow {(i,j) in A}:
  from Balance[i], to Balance[j],
  >= l[i,j], <= u[i,j],
  obj Total_Cost c[i,j];
```

Figure 3–2. *Two AMPL descriptions of the* TRANSSHIP *model described in the text.* At the left, the `node` and `arc` extensions are employed; at right, the constraints are given in the customary algebraic way.

```
node N {cities,times};
```

This declaration specifies one node for each city at each time in the planning period.

More complicated network models have several kinds of nodes, each defined by a separate `node` declaration. In the DIST example of Appendix A there are in fact 7 such declarations,

```
node RT: rtmin <= net_out <= rtmax;   # source of regular crews
node OT: otmin <= net_out <= otmax;   # source of overtime crews
node P_RT {fact};                     # regular crews at factories
node P_OT {fact};                     # overtime crews at factories
node M {prd,fact};                    # manufacturing locations
node D {prd,dctr};                    # distribution centers
node W {p in prd, w in whse}: net_in = dem[p,w];   # warehouses
```

The first two of these define individual nodes, while the others are declarations of indexed collections. Dummy indices, such as `p in prd`, are not specified except (as for `W`) where needed in the remainder of the declaration, to characterize the balance condition at the nodes.

In the case of TRANSSHIP, the balance condition is described by the phrase `net_out = b[i]`, which indicates that the net flow out of node `Balance[i]`—that is, flow out minus flow in—must equal `b[i]`. This is precisely the condition that was previously specified by the algebraic constraint `Balance[i]` in the network model, except that the outflow is represented by just the keyword `net_out`, rather than by the long expression `sum {(i,j) in A} Flow[i,j] - sum {(j,i) in A} Flow[j,i]`.

The DIST model exhibits some other possibilities. The balance condition may involve inequalities, as in the case of nodes `RT` and `OT`. When a node is a destination of flow, such as a warehouse, then it is more natural to specify the condition in terms of the net flow in; thus the condition for node `W[p,w]` is `net_in = dem[p,w]`. A pure transshipment node can be specified by either `net_in = 0` or `net_out = 0`; such a node is the default in AMPL, so it can be defined by giving no condition at all, as in several of the DIST declarations.

In general, the balance condition may take 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*

The *arith-expr* may be any AMPL arithmetic expression in previously declared sets and parameters, while the *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`

Any balance condition written in these ways is guaranteed to be a constraint on net flow of the kind that must appear in a network linear program.

As an alternative, we could allow the balance condition to be any AMPL expression that uses `net_in` or `net_out`. The keywords `flow_in` and `flow_out` might be provided as well, to represent just the flows in or flows out. This approach does have the appeal of introducing no new syntactic rules; but it allows the modeler to specify a balance condition that is not a proper restriction on flow out minus flow in, or indeed that does not represent any kind of "balance" at all. Such an alternative would go well beyond our immediate goal of allowing network flow linear programs to be specified in an obvious way. Hence we have chosen to proceed more cautiously, by permitting only the forms of the balance condition specified above.

3.3 Design specifics: `arc` declarations

An `arc` declaration names and defines a network arc, or an indexed collection of arcs. The arcs of the TRANSSHIP example are declared as follows:

```
arc Flow {(i,j) in A}:
     from Balance[i], to Balance[j],
     >= l[i,j], <= u[i,j], obj Total_Cost c[i,j];
```

There is one arc `Flow[i,j]` for each member `(i,j)` of the set `A`. This arc takes the place of the like-named variable in the algebraic model. Much as in the case of the nodes, one might informally think of `A` as actually being the set of arcs, but in AMPL it is a set of pairs of objects over which the collection `Flow` of arcs happens to be indexed. More complex models have an `arc` declaration for each different kind of arc; in the TRAIN model, for example, two of the declarations begin:

```
arc U {c in cities, t in times} ...
arc X {(c1,t1,c2,t2) in schedule} ...
```

An arc `U[c,t]` represents passenger cars held over in city `c` at time `t`, while an arc `X[c1,t1,c2,t2]` stands for cars traveling in a train from city `c1` (at time `t1`) to city `c2` (at time `t2`).

The body of an `arc` declaration consists of a series of phrases, in any convenient order. (Commas separating the phrases are optional; their only use is to make the boundaries between phrases a little more obvious.) At the least, there must be `from` and `to` phrases to show where the arc appears in the network; their general form is

`from` *node-name*
`to` *node-name*

In TRANSSHIP, the arc `Flow[i,j]` is naturally declared to be `from Balance[i]` and `to Balance[j]`. More elaborate models use these phrases to establish the structure of the network. Thus within the TRAIN example, the `U[c,t]` arcs are `from N[c,t]` and `to N[c,next(t)]`, while the `X[c1,t1,c2,t2]` arcs are `from N[c1,t1]` and `to N[c2,t2]`.

Other phrases specify bounds on the flow along the arc. They have the general forms

`>=` *arith-expr*
`<=` *arith-expr*

In TRANSSHIP the bounds for arc `Flow[i,j]` are `>= l[i,j]` and `<= u[i,j]`; in DIST most of the bounds are `>= 0`. AMPL recognizes similar phrases with the operator being `=` to fix certain flows, or `:=` to suggest initial flows for algorithmic purposes.

Finally, there is a phrase for a linear objective function coefficient:

`obj` *objective-name arith-expr*

The *arith-expr* is evaluated to give the arc's contribution, per unit of flow, to the named objective's value. As in other AMPL models, the objective must be previously declared. For example, in TRANSSHIP, prior to the use of the objective function phrase `obj Total_Cost c[i,j]`, there is a declaration

```
minimize Total_Cost;
```

Unlike other `minimize` or `maximize` declarations, this one requires no expression for the objective function. The objective coefficients are fully defined by the `obj` phrase in the subsequent `arc` declaration.

As another example, the TRAIN model declares two objectives,

```
minimize cars;
minimize miles;
```

and the subsequent `arc X` declaration is seen to have two corresponding `obj` phrases:

```
obj {if t2 < t1} cars 1
obj miles distance[c1,c2]
```

The special indexing expression `{if t2 < t1}` causes a coefficient of `1` to be generated in the `cars` objective for those arcs `X[c1,t1,c2,t2]` that have `t2` less than `t1`. By default, all other `X` arcs receive coefficients of zero in the `cars` objective, with the result that this objective only counts cars in trains that are still running at the end of the day. Similarly, because the `arc U` declaration has only a phrase for the `cars` objective, all `U` arcs are assumed to have coefficients of zero in the `miles` objective.

3.4 Design specifics: source and sink variables

As described up to this point, AMPL's `node` and `arc` declarations conveniently describe a broad variety of models based on the kind of network exhibited in Figure 3–1. Nonzero flows in these networks are necessitated by positive lower bounds on the arcs, or by balance conditions at the nodes. The formulations may be very general as in TRANSSHIP, or may reflect the structure of the application as in DIST and TRAIN. (The general and structured approaches to network model formulation are contrasted in greater detail in Chapter 11 of [20].)

Another class of network models is more naturally symbolized by the diagram in Figure 3–3. Here the flow into a designated *source* node, and the flow out of a designated *sink* node, are variables rather than data of the model. At least one of these variables plays a fundamental role in forcing flow through the network.

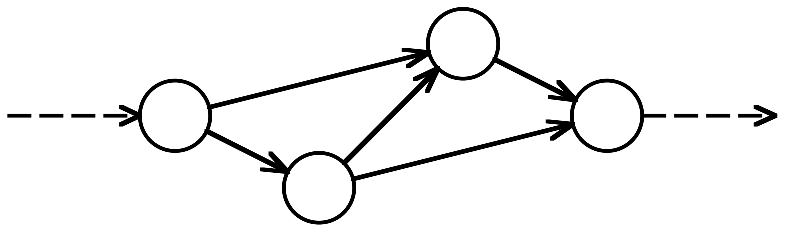


Figure 3–3. *Directed network with source and sink flows.* The dashed arrows at the left and right depict flow into the network and flow out of the network, respectively; either may be a variable of the associated linear program.

As an illustration, we consider a generic maximum-flow model. The objective is to maximize the flow into the source (or equivalently, the flow out of the sink), subject to conservation of flow at the nodes and bounds on the flows along the arcs. The formulation begins by specifying a set of nodes as in Figure 3–2, and by designating two specific nodes to be the source and sink:

```
set N;

param srce symbolic in N;
param sink symbolic in N, != srce;
```

The parameters `srce` and `sink` are declared `symbolic` to indicate that they may be any set members; by default AMPL assumes parameters to be numbers. The phrase `!= srce` checks that the sink is not the same as the source.

As in our previous example, the arcs are indexed over a subset of `N cross N`. In this case, we choose to restrict the arc set a bit more, by specifying that arcs cannot be from the sink or to the source:

```
set A within (N diff {sink}) cross (N diff {srce});
```

We can now proceed to declare the bound data, the nodes, and the arcs between nodes, in much the same way as before:

```
param l {A} >= 0;
param u {(i,j) in A} >= l[i,j];

node Balance {i in N};

arc Flow {(i,j) in A}:
     from Balance[i], to Balance[j],
     >= l[i,j], <= u[i,j];
```

We specify no explicit balance condition for the nodes, since in this application the flow in equals the flow out at every node.

It remains to declare the source and sink arcs. The diagram in Figure 3–3 strongly suggests that we regard a source arc as directing flow to some node but not from any node, and a sink arc as directing flow from some node but not to any node. Thus AMPL allows for an arc declaration that contains only a `to` phrase or only a `from` phrase:

```
set N;

param srce symbolic in N;
param sink symbolic in N, != srce;

set A within (N diff {sink}) cross (N diff {srce});

param l {A} >= 0;
param u {(i,j) in A} >= l[i,j];

node Balance {i in N};

arc Flow {(i,j) in A}:
    from Balance[i], to Balance[j],
    >= l[i,j], <= u[i,j];

arc Flow_In >= 0, to Balance[srce];
arc Flow_Out >= 0, from Balance[sink];

maximize Total_Flow: Flow_In;
```

Figure 3–4. *An AMPL formulation of a maximum flow model,* using the `node` and `arc` declarations.

```
        arc Flow_In >= 0, to Balance[srce];
        arc Flow_Out >= 0, from Balance[sink];
```

The objective is to maximize one or the other of these variables, and so is very easily specified by a conventional objective declaration at the end:

```
        maximize Total_Flow: Flow_In;
```

Alternatively, the model could declare `maximize Total_Flow` before the arc declarations, and could then specify the one term in the objective by adding the phrase `obj Total_Flow 1` to the declaration of `Flow_In`.

The completed model is shown in Figure 3–4. A similar approach can be used to model maximum flow problems having a more complex structure, including any combination of multiple source and sink flows. Similar formulations are useful for other problems that can be viewed as "sending flow through the network" from one designated node to another. For example, the well-known shortest path problem is conveniently modeled by putting a cost on each arc `Flow[i,j]` equal to its length, and fixing the `Flow_In` or `Flow_Out` variable to 1.

3.5 Translation issues

When a network model is processed (along with appropriate data) by the AMPL translator, each `node` declaration gives rise to a constraint; each `arc` declaration defines a variable, which is given coefficients of -1 and $+1$ in the constraints indicated by `from` and `to` phrases. The translator's output can thus be sent to any solver that recognizes linear programs. The output includes sufficient information, however, to permit drivers for such optimization packages as CPLEX [9] and OSL [29] to recognize a network linear program and to apply their much faster network optimization routines. Regardless of the algorithm applied, each driver returns an optimal solution in the AMPL standard form; AMPL's commands for examining solutions then treat arcs like any other variables, and nodes like any other constraints.

It is in fact quite easy to determine whether a linear program generated by the AMPL translator is a pure network LP, no matter what kind of declarations the model has employed. In the easiest case, any constraints that represent simple bounds are first folded into the bounds on the variables. Then each variable is checked to insure that it has at most two nonzero coefficients in the remaining constraints, consisting of at most one $+1$ and one -1.

When the constraints are written using `subject to` declarations, however, the test for a pure network becomes somewhat more involved. Depending on how the model is expressed and how the translator converts AMPL declarations into a constraint matrix, some of the constraints may end up being in the form *flow in* $-$ *flow out* $=$ *constant,* while others are in the form *flow out* $-$ *flow in* $=$ *constant;* some variables may as a result have two $+1$ or two -1 coefficients. Hence it may be necessary to reflect certain constraints (that is, to scale them by -1) in order to reveal the network structure. Fortunately, there exist fast and simple algorithms that find the necessary combination of reflections (or determine that none exists). We have implemented one such algorithm in our AMPL/OSL driver, and a similar algorithm is planned for introduction into the CPLEX package itself.

The advantage of AMPL's `node` and `arc` declarations thus lies not so much in helping solvers to identify pure network flow structures, as in making network models easier for people to formulate and understand. As a secondary benefit, the use of these declarations helps to enforce the formulation of optimization problems as network flow LPs. When a model is formulated instead by use of `var` and `subject to`, it is up to the modeler to ensure that the constraints have the pure network flow structure that fast algorithms require. In the case of the DIST model, for example, our original version in [18] used an equivalent but slightly different formulation that departs from the network structure by defining certain variables to have nonzero coefficients in three different constraints.

4. Generalizations of the Network Model Features

Many network applications diverge from the "pure" network flow formulation considered in the previous section. With only modest further extensions, however, our AMPL `node` and `arc` declarations can be used to specify several more general kinds of network.

This section describes extensions for network flow models with "multipliers" on the flows, and with extra ("side") constraints or variables.

4.1 Network flows with multipliers

The network balance-of-flow constraints may be generalized by allowing arbitrary coefficients on the flows in and out:

$$\sum_{(i,j)\in\mathcal{A}} v_{ij}x_{ij} - \sum_{(j,i)\in\mathcal{A}} w_{ji}x_{ji} = b_i, \quad i \in \mathcal{N}.$$

In terms of the network, one can imagine that the flow x_{ij} is from node i with multiplier v_{ij}, and to node j with multiplier w_{ij}. A pure network flow problem (as in Section 3) is just the default case in which all multipliers are 1.

AMPL supports this generalization by allowing an optional expression for the multipliers in a `from` or `to` phrase:

> `from` *node-name mult-expression*
> `to` *node-name mult-expression*

The *mult-expression* can be any AMPL expression in the previously declared sets and parameters. It defaults to 1 if absent.

An example of this feature is found in the DIST model's declaration of the arcs `Manu_OT`. Each arc `Manu_OT[p,f]` carries a "flow" of overtime labor—measured in crew-hours—from the overtime pool at factory `f` (node `P_OT[f]`) to the manufacturing facilities for product `p` at factory `f` (node `M[p,f]`). The balance constraint at the "to" node is not in terms of crew-hours, however, but rather in 1000s of cases produced. To correct for this difference, the flow must be multiplied by `1/pt[p,f]`, where `pt[p,f]` is the previously defined parameter that represents crew-hours needed to produce 1000 cases.

The declaration of `Manu_OT` thus contains the phrases

```
from P_OT[f]
to M[p,f] (1/pt[p,f])
```

The multiplier at the "from" node is left at its default of 1, since that node does have a balance constraint in terms of crew-hours.

Besides changes in units, this generalization serves to model such diverse phenomena as losses incurred in transportation and interest accrued on cash flows.

4.2 Side constraints

Some network problems are constrained by more than just balance of flow at the nodes and bounded flows on the arcs. The additional "side constraints" take

```
set N;
set A within N cross N;

set P;  # products

param b {N,P};
param c {A,P} >= 0;

param l {A,P} >= 0;
param u {(i,j) in A, p in P} >= l[i,j,p];

param u_mult {A} >= 0;

minimize Total_Cost;

node Balance {i in N, p in P}: net_out = b[i,p];

arc Flow {(i,j) in A, p in P}:
     from Balance[i,p], to Balance[j,p],
     >= l[i,j,p], <= u[i,j,p], obj Total_Cost c[i,j,p];

subject to Mult {(i,j) in A}:
     sum {p in P} Flow[i,j,p] <= u_mult[i,j];
```

Figure 4–1. *A multicommodity flow model with side constraints.*

many forms, but in general they do not correspond to individual nodes or arcs of the network. Thus it is desirable to let an AMPL model specify arbitrary algebraic constraints in addition to the constraints implied by `node` and `arc` declarations.

The problem of multicommodity flows provides a simple example. In the network model of Section 3, we can imagine the nodes as representing locations, and the flows along the arcs as being the transportation of some product. To accommodate multiple products, we simply replicate the model so that there are separate network data, nodes and arcs for each product in some set. Instead of an arc `Flow[i,j]` for each `(i,j)` in `A`, for example, we may have an arc `Flow[i,j,p]` for each `(i,j)` in `A` and `p` in `P`.

If we carry out this replication and nothing more, we have a model that decomposes into separate pure network linear programs, one for each product. In a more likely scenario, however, total shipments of all products from location `i` to location `j` are bounded by some overall upper limit. This additional multicommodity constraint ties the different product networks together; since it simultaneously involves many disjoint arcs `Flow[i,j,p]` for different products `p`, it cannot be modeled by merely adding more nodes and arcs. Rather, we recall from Section 3 that arcs correspond to variables in the associated network linear program. We can thus use the names `Flow[i,j,p]` to stand for the flow variables in an algebraic description of the multicommodity restriction.

The resulting AMPL model, as seen in Figure 4–1, introduces a new parameter to represent overall shipment bounds for all `(i,j)` in `A`:

```
param u_mult {A} >= 0;
```

The desired constraints are then written as

```
subject to Mult {(i,j) in A}:
     sum {p in P} Flow[i,j,p] <= u_mult[i,j];
```

The entire model now clearly appears as a network (defined by `node` and `arc`) with side constraints (defined by `subject to`).

This example generalizes in the obvious way. Once an entity has been defined in an `arc` declaration, it represents a model variable, and may be used as such in any subsequent `subject to` declaration.

4.3 Side variables

Once we have decided that `subject to` can be employed in conjunction with `node` and `arc`, we can allow supplementary `var` declarations as well. A `var` declaration defines “side variables” that can be used, along with variables implicitly defined by `arc` declarations, in any constraint defined by use of `subject to`.

The side variables of some network models appear not in the side constraints, however, but as complicating terms in the balance-of-flow constraints at the nodes. Figure 4–2 shows how these variables can be handled in an obvious way within the `node` declarations of AMPL. This model is based on a replication of Section 3’s network model, just like the previous one. The separate product networks are held together not by additional constraints, however, but by additional variables representing feedstocks that can be used at the nodes.

```
set N;
set A within N cross N;

set F;  # feedstocks
set P;  # products

param a {P,F} >= 0;
param u_feed {F,N} >= 0;

param b {N,P};
param c {A,P} >= 0;

param l {A,P} >= 0;
param u {(i,j) in A, p in P} >= l[i,j,p];

minimize Total_Cost;

var Feed {f in F, i in N} >= 0, <= u_feed[f,i];

node Balance {i in N, p in P}:
     net_out = b[i,p] + sum {f in F} a[p,f] * Feed[f,i];

arc Flow {(i,j) in A, p in P}:
     from Balance[i,p], to Balance[j,p],
     >= l[i,j,p], <= u[i,j,p], obj Total_Cost c[i,j,p];
```

Figure 4–2. *A multicommodity flow model with side variables.*

The new parameter declarations in Figure 4–2 define the amount `a[p,f]` of product `p` that can be derived from one ton of feedstock `f`, and the limit `u_feed[f,i]` on feedstock `f` available at location `i`:

```
param a {P,F} >= 0;
param u_feed {F,N} >= 0;
```

The new connecting variables `Feed[f,i]` represent the amount of feed `f` used at location `i`:

```
var Feed {f in F, i in N} >= 0, <= u_feed[f,i];
```

Thus the amount of product `p` derived from feedstock `f` at location `i` is given by `a[p,f] * Feed[f,i]`. The total of product `p` derived from feedstocks at location `i` is the sum of this quantity over all feedstocks `f` in `F`.

At the node `Balance[i,p]`, we want to specify that flow out minus flow in is equal to `b[i,p]` (as before) plus the total of product `p` that is derived from feedstocks at location `i`. Thus the AMPL node declaration must be:

```
node Balance {i in N, p in P}:
     net_out = b[i,p] + sum {f in F} a[p,f] * Feed[f,i];
```

To permit this formulation in AMPL, we need only slightly expand our previous definition of the `node` declaration, to allow the use of variables in an *arith-expr* within the balance condition.

The arcs of this model are defined as before, using an `arc` declaration. In this way, the distinction between network and side variables is emphasized by the AMPL formulation.

5. Other Column-Wise Structures

Although the AMPL network extensions can be motivated entirely by a desire to more naturally represent network flow models, they can also be viewed as a special case of an alternative "column-wise" approach to the specification of linear programs. As a result, some of the ideas of the `node` and `arc` declarations can reasonably be introduced into the `var` and `subject to` declarations as well.

In this section, we first present an extension for another special structure, that of set covering problems. We then show how all of the preceding ideas can be adapted to permit column-wise specification for linear programs of any structure.

5.1 Set covering

When we tried to represent various set covering models in AMPL, we encountered much the same kinds of problems that we had with networks. Thus we have used much the same approach in devising a solution, though some important details are necessarily different.

In a simple generic set covering model, we are given subsets $\mathcal{S}_j \subset \mathcal{U}$ and costs c_j, for $j = 1, \ldots, n$. Our goal is to find the cheapest collection of subsets whose union contains all of $\mathcal{U}$. This problem is readily formulated as a zero-one integer program, in which the variable x_j is 1 if and only if the subset $\mathcal{S}_j$ is in the desired collection:

$$\begin{array}{lll} \text{Minimize} & \sum_{1\le j\le n} c_j x_j & \\ \text{Subject to} & \sum_{1\le j\le n\,:\, i\in\mathcal{S}_j} x_j \ge 1, & \text{for each } i \in \mathcal{U} \\ & x_j \in \text{–}0,1\text{''}, & \text{for each } j = 1,\ldots,n \end{array}$$

As in the network case, the summation within the main constraint seems awkward, particularly compared to the very simple original description of the model.

The awkwardness persists when the common algebraic notation is transcribed to AMPL, as shown in Figure 5–1. Moreover, the processing of this formulation is likely to be inefficient. When it comes to evaluating the summation in the constraints,

```
subject to complete {i in U}:
     sum {j in 1..n: i in S[j]} x[j] >= 1;
```

the condition `i in S[j]` must be tested for every one of the `n` subsets `S[j]`. In our implementation of an AMPL translator, these constraints are generated individually for each instance of `i in U`; as a result, the number of tests of the form `i in S[j]` that the translator must make is equal to `n` times the cardinality of `U`. Since the number of possible subsets of `U` grows exponentially, the size of `n` can easily be 100,000 or more, while values in the millions are not unknown for crew scheduling applications. As a result, this kind of AMPL model can become very expensive to translate.

To circumvent these difficulties, AMPL offers a more natural and efficient way to describe the set covering model's coefficients. A variable `x[j]` has nonzero coefficients of 1 in the constraints `complete[i]` for each `i in S[j]`, as well as a coefficient `c[j]` in the objective. The following AMPL alternative says this directly:

```
var x {j in 1..n} binary,
     cover {i in S[j]} complete[i], obj cost c[j];
```

```
### SET-COVERING EXTENSION ###

param n > 0;
param c {1..n} >= 0;

set U;
set S {1..n} within U;


minimize cost;


subject to complete {i in U}:
  to_come >= 1;

var x {j in 1..n} binary,
  cover {i in S[j]} complete[i],
  obj cost c[j];
```

```
### ALGEBRAIC FORMULATION ###

param n > 0;
param c {1..n} >= 0;

set U;
set S {1..n} within U;

var x {1..n} binary;

minimize cost:
  sum {j in 1..n} c[j] * x[j];

subject to complete {i in U}:
  sum {j in 1..n: i in S[j]} x[j] >= 1;
```

Figure 5–1. *AMPL formulations of the set-covering problem.* At left, the description uses the `cover` extension; at right, the covering constraints are specified in the usual algebraic way.

As in the network models that use `node` and `arc`, the objective and constraints are declared before the variables:

```
minimize cost;
subject to complete {i in U}: to_come >= 1;
```

In the constraints, the keyword `to_come` is a placeholder that works much like `net_in` or `net_out`; it shows where the linear expression in the variables is to be placed. The complete model is exhibited in Figure 5–1 alongside its algebraic equivalent.

In `var` declarations generally, a `cover` phrase specifies nonzero coefficients within previously declared constraints. It has the forms

`cover` *constraint-name*
`cover {`*indexing*`}` *constraint-name*

The first form places a coefficient of 1 in the named constraint; it is the analogue of the previously discussed `to` and `from` phrases. The second form uses `{`*indexing*`}` to specify an entire indexed collection of constraints in which a coefficient of 1 is to be placed. This feature is required by our example, because the number of coefficients of `x[j]` is determined by the cardinality of `S[j]`, which varies according to the data supplied. (By contrast, a network model has at most two coefficients per variable.)

To process this alternative AMPL formulation, the translator need only scan each subset `S[j]` once. This can represent a particularly great savings when `n` is large and each subset is just a small part of `U`.

5.2 Arbitrary column-wise formulations

The most significant feature common to our network examples (using `node` and `arc`) and our set-covering examples (using `cover`) is the specification of the coefficients within the declarations of the variables. In the argot of linear programming, these are column-wise formulations, since a variable's coefficients lie in one column of the constraint matrix. By contrast, a fully algebraic formulation is row-wise, specifying the coefficients within the declarations of the constraints.

Most efficient implementations of linear programming algorithms have used a data structure in which the coefficients are grouped by column. As a result, early "matrix generators" tended to encourage column-wise formulations. This view has persisted, out of habit or convenience, so that in certain applications it is still customary to view linear programs "by activity" rather than by constraint [34, 43].

Once the syntactic forms for networks and set covering have been introduced, it is only a small step to provide for column-wise specification of coefficients in general. In the same way that we allow `node` declarations to employ `net_in` and `net_out`, we allow `subject to` declarations to employ the placeholder `to_come` within constraint expressions of the forms

> *variable-expr* `=` *arith-expr*
> *variable-expr* `<=` *arith-expr*
> *variable-expr* `>=` *arith-expr*
>
> *arith-expr* `=` *variable-expr*
> *arith-expr* `<=` *variable-expr*
> *arith-expr* `>=` *variable-expr*
>
> *arith-expr* `<=` *variable-expr* `<=` *arith-expr*
> *arith-expr* `>=` *variable-expr* `>=` *arith-expr*

where the *variable-expr* is any of

> `to_come`
> `to_come +` *arith-expr*
> *arith-expr* `+ to_come`

As in the case of set covering, the keyword `to_come` indicates where AMPL will place the linear terms that are to be specified in a column-wise fashion.

In subsequent `var` declarations, we specify the variables' coefficients by use of a `coeff` phrase that combines the features of the `from`/`to` and `cover` phrases:

> `coeff` *constraint-name value*
> `coeff {`*indexing*`}` *constraint-name value*

The first form says that the variable has a coefficient with the given value in the named constraint. The second form is similar except that it defines a collection of coefficients, one for each member of the set specified by `{`*indexing*`}`.

Objectives are handled in an analogous way. A `minimize` or `maximize` declaration may use any *variable-expr* as above to specify an objective function, though the function consisting of `to_come` alone is most common and is assumed by default. In subsequent `var` declarations, `obj` phrases specify variables' coefficients in the objectives. The syntax is exactly the same as for `coeff` above, except for the initial keyword `obj`.

Figure 5–2 shows how the simple production model of Figure 1–1 can be ex-

```
### COLUMN-WISE FORMULATION ###

set R;
set P;

param a {R,P} > 0;
param b {R} > 0;
param c {P} > 0;
param u {P} > 0;

maximize tot;

subject to supply {i in R}:
 to_come >= b[i];

var x {j in P} >= 0, <= u[j],
 coeff {i in R} supply[i] a[i,j],
 obj tot c[j];
```

```
### ROW-WISE FORMULATION ###

set R;
set P;

param a {R,P} > 0;
param b {R} > 0;
param c {P} > 0;
param u {P} > 0;

var x {j in P} >= 0, <= u[j];

maximize tot:
 sum {j in P} c[j] * x[j];

subject to supply {i in R}:
 sum {j in P} a[i,j] * x[j] >= b[i];
```

Figure 5–2. *Column-wise (left) and row-wise forms for a linear program in AMPL. This simple production model was introduced in Figure 1–1.*

```
set N;
set A within N cross N;
set P;

param b {N,P};
param c {A,P} >= 0;

param l {A,P} >= 0;
param u {(i,j) in A, p in P} >= l[i,j,p];
param u_mult {A} >= 0;

minimize Total_Cost;

node Balance {i in N, p in P}: net_out = b[i,p];

subject to Mult {(i,j) in A}: to_come <= u_mult[i,j];

arc Flow {(i,j) in A, p in P}:
     from Balance[i,p], to Balance[j,p],
     >= l[i,j,p], <= u[i,j,p],
     coeff Mult[i,j] 1.0,
     obj Total_Cost c[i,j,p];
```

Figure 5–3. *An entirely column-wise formulation for the multicommodity flow model introduced in Figure 4–1.*

pressed either row-wise or column-wise through different uses of `var`, `maximize`, and `subject to`. The `coeff` phrase can also be used within an `arc` declaration, to specify the coefficients in side constraints; Figure 5–3 shows how this option may be applied so as to convert the model of Figure 4–1 to an entirely column-wise formulation.

6. Reflections and Conclusions

To this point, our presentation has concentrated on specific design decisions motivated by the extension of AMPL to several model structures. We conclude by considering some of the broader issues raised by our designs.

We assert that AMPL's structure extensions add valuable "redundancies" to the language, and permit the benefits of algebraic notation to be enjoyed even when the objective and constraints are not specified in a conventional algebraic way. We then summarize the various efficiencies that the extensions make possible in translation and solution of models. Finally, we indicate some likely directions of future work on modeling languages and systems for structured optimization problems.

6.1 Redundancy in the language design

In the preceding discussion of column-wise coefficient specification, we observed that an `obj` phrase places coefficients within objective functions in just the same way that a `coeff` phrase places coefficients within constraints, using the same syntax except for the initial keyword. Evidently the `obj` phrase is redundant, in that the `coeff` phrase could just as well serve the same purpose. AMPL enforces a distinction, however, for two reasons: to promote the readability of `var` declarations, and to allow for some additional error-checking. This approach is consistent with an earlier and more fundamental AMPL design decision, to syntactically distinguish the declarations of objectives from the declarations of constraints. It is possible to design a language that uses the same syntax for both, as in the case of GAMS [6].

The `cover` phrase is a similar case of redundancy. Its role is to provide a convenient, suggestive and reliable way of specifying coefficients equal to 1.

The special network syntaxes of Section 3 are also redundant in a sense. The `node` and `arc` declarations could be subsumed by `subject to` and `var`; the effect of the `from` and `to` phrases could be accomplished by `coeff`; and the `net_in` and `net_out` keywords could be replaced by `to_come`. In this situation, however, the special syntax is particularly advantageous. It permits the AMPL model to more closely correspond to people's common notions of a network linear programming formulation, and ensures that every instance generated by the model translator has the proper network coefficient structure.

As this discussion suggests, there are many redundancies built into AMPL, only some of which are introduced by the extensions described in this paper. Examples of others have been cited in our analysis of fundamental language features [19]. In each case, some degree of naturalness and error-checking has been achieved, at some loss of simplicity and generality. We have sought to limit redundancies to cases in which this tradeoff is reasonably favorable.

6.2 Benefits of an algebraic language

Algebraic notation and modeling languages are most often associated with a row-wise specification of the constraint coefficient matrix, simply because they are most often used to express a constraint-by-constraint formulation. The discussions in [34] and [43] are representative of this view. As our examples have shown, however, many of the features of an algebraic language are just as valuable in a column-wise specification.

AMPL's apparatus for describing data—in `set` and `param` declarations—is seen in Figures 3–2, 5–1 and 5–2 to provide equally good support for row-wise and column-wise constraint specifications. AMPL indexing expressions also prove to be equally useful whether they are indexing sums in an algebraic declaration, or constraint names in a `coeff` phrase.

Within a `from`, `to`, `coeff` or `obj` phrase, the coefficient value may be specified by a nontrivial algebraic expression, as in this example from the DIST model of Appendix A:

```
arc Manu_RT {p in prd, f in fact: rpc[p,f] <> 0} >= 0
     from P_RT[f]  to M[p,f] (dp[f] * hd[f] / pt[p,f])
     obj cost (rpc[p,f] * dp[f] * hd[f] / pt[p,f]);
```

Column-wise specifications can make extensive use of an algebraic language in this way, even though they do not describe any complete objective or constraint algebraically.

Finally, an algebraic modeling language can accommodate convenient hybrids of column-wise and row-wise specifications. Figure 4–1 offers an example in a model of a network with side constraints. As another possibility, a model could have a nonlinear objective function expressed algebraically in a `minimize` or `maximize` declaration, but subject to network constraints specified by means of `node` and `arc` declarations.

6.3 Efficiency of translation

We previously remarked that the AMPL language translator should be able to process certain set covering models much faster when they are specified column-wise, by use of the `cover` phrase. Table 6–1 presents evidence of this advantage; in a randomly constructed 100 by 25000 example, the speedup is by a factor of about 9.

Other tests reported in Table 6–1 show that the special syntaxes for network and piecewise-linear structures also permit a speedup, though to a much more modest degree. In the network examples, both alternatives generate the same linear program, but the translator can process the `node` and `arc` declarations somewhat more efficiently than the corresponding `var` and `subject to` declarations. In the piecewise-linear examples, the advantage is due to the P-L formulation's smaller number of variables and coefficients than the equivalent LP formulation.

6.4 Efficiency of solution

Previous investigations have established that network and piecewise-linear models can be solved faster by methods that take advantage of their special structures. When AMPL is used to formulate models for these methods, its special structure syntax can make possible some additional efficiencies.

For an algorithm to take advantage of structure in a linear program, it must receive input indicating where the structure is present. If piecewise-linear functions

	COVER	DIST	SCORE	STRUC	TRANS
Without extensions	92.20	1.38	11.23	0.87	14.35
With extensions	10.37	1.18	8.80	0.77	7.90

Table 6–1. *Efficiencies afforded by the AMPL structure extensions.* Timings are for the "genmod" phase of the AMPL translator, on a Sun SPARCstation 2; other phases (as described in [19]) were much more nearly the same. The test problems are as follows:

COVER: The set covering model of Figure 5–1 with randomly generated data. There are 100 members in the set `U`, and 25000 subsets of one to five members each.

DIST: The network model of Appendix A with a realistic data file; similar to test problem SHIP12L in the netlib set [24]. There are 1268 nodes and 2295 arcs.

SCORE: The piecewise-linear model of Appendix B with a realistic data file; equivalent to test problem FIT2P in the netlib set [24]. The piecewise-linear version has 3000 constraints in 3025 variables, with 4 or 5 linear pieces per variable.

STRUC: The piecewise-linear model of Appendix C with a realistic data file; equivalent to test problem SCSD8 in the netlib set [24]. The piecewise-linear version has 397 constraints in 1375 variables.

TRANS: The network model of Figure 3–2 with randomly generated data. There are 5000 nodes and 50455 arcs.

or network nodes and arcs are declared explicitly, then an AMPL translator can automatically generate the input that a specialized algorithm requires. If ordinary algebraic declarations are employed, however, then the algorithm must incorporate a pre-processing stage in order to make the structure explicit for its purposes.

Previous studies have described pre-processors both for network constraints [44] and for piecewise-linear objective terms [23]. These routines necessarily involve some extra cost, but they are usually cheap compared to the overall cost of translation and solution. A greater source of difficulty lies in the possibility that, due to some error in the formulation, the generated model may fail to exhibit the expected structure. A pre-processor must be relied upon to detect any such error and to communicate it in a useful way to the modeler. No analogous error handling is required when the desired structure is already explicit in the AMPL model.

6.5 Directions for further investigation

There are special algorithms for other kinds of structures in linear programming. For which other structures might comparable benefits be realized, through extension of an algebraic modeling language such as AMPL? Our observations suggest that a structure is a good candidate if it is well known to modelers, and if it tends to simplify a formulation when its presence is made explicit. Introduction of a new syntax for such a structure can offer benefits in the formulation and translation of a model as well as in its solution.

Among the likely candidates for AMPL extensions, we have been investigating structures associated with stochastic programming and robust optimization. We also envision several extensions for special structures in mixed-integer programming, particularly for expressing certain logical conditions in a way that permits them to be identified as giving rise to so-called special ordered sets [2, 42]. The range of combinatorial optimization models representable in AMPL might be further expanded by introducing more ambitious extensions, such as a way of optimizing over all subsets of a given set; however, as explained in [4], these pose much greater difficulties of translation than AMPL's current structure extensions.

For the case of network optimization models, there has been considerable interest in modeling systems that employ a natural graphical representation. The simple diagrams in Figures 3–1 and 3–3 can be generalized to a broad variety of cases; Glover, Klingman and Philips [25, 26] have shown how these diagrams (or "netforms") often provide for a very natural and convenient interface between modelers and computer systems. Implementations based on this approach have been described by Dean, Mevenkamp and Monma [13], by Jones [31, 32], by Ogryczak, Studziński and Zorychta [39], and by Steiger, Sharda and LeClaire [41].

Netform interfaces to network optimization may be regarded as an alternative to the algebraic interfaces exemplified by AMPL. The netform approach tends to be most attractive when the network diagram is central to the modeler's conception of the problem, and when the relevant parameters and variables of the problem have a direct relationship to the nodes and arcs of the diagram. Algebraic representations are most advantageous when the network diagram is incidental to the modeler's conception, when substantial manipulations of the data are described within the model, or when there are nontrivial side constraints or side variables. Rather than force a user to decide between these two approaches, several investigators have proposed to maintain parallel netform and algebraic "views" of a model. The principles of a multiple-view optimization system have been set forth by Greenberg and Murphy [27]. Kendrick [35, 36] describes a system that maintains both an algebraic view in the GAMS language [6] and a netform view, while Jones and D'Souza [33] suggest how an implementation might allow network models to be manipulated both through netforms and through AMPL formulations that use `node` and `arc` declarations.

Despite the many advantages of special-structure extensions to modeling languages, extensions are often discouraged by the complication and cost of implementation. Any new feature for a particular structure inevitably interacts with many existing features, giving rise to a variety of new language rules governing their interaction, and requiring programming changes throughout the existing language translator. The design of modeling languages to facilitate extensions is thus itself a topic for further study. A framework that permits independent extensions to be "embedded" in an existing language has been proposed by Bhargava and Kimbrough [3].

Appendix A. DIST, a generalized network distribution model

This generalized network model determines a production and distribution plan to meet given demands for a set of goods. An equivalent model, employing a somewhat different formulation, is described by use of algebraic constraints in [18].

```
###  SHIPPING SETS AND PARAMETERS  ###

set whse 'warehouses';  # Locations from which demand is satisfied

set dctr 'distribution centers' within whse;
                        # Locations from which product may be shipped

param sc 'shipping cost' {dctr,whse} >= 0;
                        # Shipping costs, to whse from dctr, in $ / 100 lb

param huge 'largest shipping cost' > 0;
                        # Largest cost allowed for a usable shipping route

param msr 'minimum size restriction' {dctr,whse} logical;
                        # True indicates a minimum-size restriction on
                        # direct shipments using this dctr --> whse route

param dsr 'direct shipment requirement' {dctr} >= 0;
                        # Minimum total demand, in pallets, needed to
                        # allow shipment on routes subject to the
                        # minimum size restriction

###  PLANT SETS AND PARAMETERS  ###

set fact 'factories' within dctr;
                        # Locations where product is manufactured

param rtmin 'regular-time total minimum' >= 0;
                        # Lower limit on (average) total regular-time
                        # crews employed at all factories

param rtmax 'regular-time total maximum' >= rtmin;
                        # Upper limit on (average) total regular-time
                        # crews employed at all factories

param otmin 'overtime total minimum' >= 0;
                        # Lower limit on total overtime hours at all factories

param otmax 'overtime total maximum' >= otmin;
                        # Upper limit on total overtime hours at all factories

param rmin 'regular-time minimums' {fact} >= 0;
                        # Lower limits on (average) regular-time crews

param rmax 'regular-time maximums' {f in fact} >= rmin[f];
                        # Upper limits on (average) regular-time crews

param omin 'overtime minimums' {fact} >= 0;
                        # Lower limits on overtime hours

param omax 'overtime maximums' {f in fact} >= omin[f];
                        # Upper limits on overtime hours

param hd 'hours per day' {fact} >= 0;
                        # Regular-time hours per working day

param dp 'days in period' {fact} > 0;
                        # Working days in the current planning period
```

```
###  PRODUCT SETS AND PARAMETERS  ###

set prd 'products';     # Elements of the product group

param wt 'weight' {prd} > 0;
                        # Weight in 100 lb / 1000 cases

param cpp 'cases per pallet' {prd} > 0;
                        # Cases of product per shipping pallet

param tc 'transshipment cost' {prd} >= 0;
                        # Transshipment cost in $ / 1000 cases

param pt 'production time' {prd,fact} >= 0;
                        # Crew-hours to produce 1000 cases

param rpc 'regular-time production cost' {prd,fact} >= 0;
                        # Cost of production on regular time,
                        # in $ / 1000 cases

param opc 'overtime production cost' {prd,fact} >= 0;
                        # Cost of production on overtime, in $ / 1000 cases


###  DEMAND SETS AND PARAMETERS  ###

param dt 'total demand' {prd} >= 0;
                        # Total demands for products, in 1000s

param ds 'demand shares' {prd,whse} >= 0.0, <= 1.0;
                        # Historical demand data, from which each
                        # warehouse's share of total demand is deduced

param dstot {p in prd} := sum {w in whse} ds[p,w];
                        # Total of demand shares; should be 1, but often isn't

param dem 'demand' {p in prd, w in whse} := dt[p] * ds[p,w] / dstot[p];
                        # Projected demands to be satisfied, in 1000s

set rt 'shipping routes available' :=
 {d in dctr, w in whse:
         d <> w  and  sc[d,w] < huge  and
         (w in dctr or sum {p in prd} dem[p,w] > 0)  and
         not (msr[d,w] and sum {p in prd} 1000*dem[p,w]/cpp[p] < dsr[d]) };
                        # List of ordered pairs that represent routes
                        # on which shipments are allowed


###  OBJECTIVE  ###

minimize cost;          # Total cost:  regular production, overtime
                        # production, shipping, and transshipment


###  NODES  ###

node RT:  rtmin <= net_out <= rtmax;
                        # Source of all regular-time crews allocated

node OT:  otmin <= net_out <= otmax;
                        # Source of all overtime hours allocated

node P_RT {fact};       # Sources of regular-time crews at factories

node P_OT {fact};       # Sources of overtime hours at factories
node M {prd,fact};       # Sources of manufacturing:
                        # send to factory's W node for local demand;
                        # send to factory's D node for distribution

node D {prd,dctr};       # Sources of distribution:
                        # receive transshipped goods from center's W node;
                        # receive manufactured goods from center's M node;
                        # send to W nodes elsewhere

node W {p in prd, w in whse}:  net_in = dem[p,w];
                        # Locations of warehousing:
                        # receive from D nodes and local M node (if any),
                        # to satisfy local demand;
                        # send to local D node (if any) for transshipment


###  ARCS  ###

arc Work_RT {f in fact}
      from RT  to P_RT[f]  >= rmin[f],  <= rmax[f];
                        # Regular-time crews allocated to each factory

arc Work_OT {f in fact}
      from OT  to P_OT[f]  >= omin[f],  <= omax[f];
                        # Overtime hours allocated to each factory

arc Manu_RT {p in prd, f in fact: rpc[p,f] <> 0} >= 0
      from P_RT[f]  to M[p,f] (dp[f] * hd[f] / pt[p,f])
      obj cost (rpc[p,f] * dp[f] * hd[f] / pt[p,f]);
                        # Regular-time crews allocated to
                        # manufacture of each product at each factory

arc Manu_OT {p in prd, f in fact: opc[p,f] <> 0} >= 0
      from P_OT[f]  to M[p,f] (1 / pt[p,f])  obj cost (opc[p,f] / pt[p,f]);
                        # Overtime hours allocated to
                        # manufacture of each product at each factory

arc Prod_L {p in prd, f in fact} >= 0
      from M[p,f]  to W[p,f];
                        # Manufacture of each product at each factory
                        # to satisfy local demand, in 1000s of units

arc Prod_D {p in prd, f in fact} >= 0
      from M[p,f]  to D[p,f];
                        # Manufacture of each product at each factory,
                        # for distribution elsewhere, in 1000s of units

arc Ship {p in prd, (d,w) in rt} >= 0
      from D[p,d]  to W[p,w] obj cost (sc[d,w] * wt[p]);
                        # Shipments of each product on each allowed route

arc Trans {p in prd, d in dctr} >= 0
      from W[p,d]  to D[p,d]  obj cost (tc[p]);
                        # Transshipments of each product at each
                        # distribution center
```

Appendix B. SCORE, a piecewise-linear data fitting model

This model fits linear inequalities to data, by minimizing a sum of convex piecewise-linear penalties on the deviations from an acceptable fit. The motivation comes from a problem of weighting certain customer attributes so as to distinguish "good" from "bad" credit risks.

```
###  DATA ON PEOPLE  ###

set Good;                              # Good risks
set Bad;                               # Bad risks

set people := Good union Bad;          # Everyone is either good or bad

param app_amt > 0;                     # General credit-approval amount
param bal_amt {people} >= app_amt;     # Maximum-balance amounts of individuals

###  DATA ON FACTORS  ###

set factors;                           # Questions posed to individuals

set wt_types := {'pos','neg','free'};  # Required signs of weights
param wttyp {factors} symbolic within wt_types;

param answer {people,factors} >= 0;    # Numerical responses to all questions

###  DATA DEFINING THE PENALTY FUNCTION  ###
# Parameters starting with G (or B) are for the good (or bad) risks

param Gpce > 1;
param Bpce > 1;                        # Linear pieces in penalty term

param Gslope {1..Gpce};  check {k in 1..Gpce-1}: Gslope[k] < Gslope[k+1];
param Bslope {1..Bpce};  check {k in 1..Bpce-1}: Bslope[k] < Bslope[k+1];
                                       # Increasing slopes in penalty term

set bkpt_types := {'A','B'};
param Gbktyp {1..Gpce-1} symbolic within bkpt_types;
param Bbktyp {1..Bpce-1} symbolic within bkpt_types;
param Gbkfac {1..Gpce-1};  check {k in 1..Gpce-2}: Gbkfac[k] <= Gbkfac[k+1];
param Bbkfac {1..Bpce-1};  check {k in 1..Bpce-2}: Bbkfac[k] <= Bbkfac[k+1];
                                       # Information to define the
                                       # increasing breakpoints in penalty
                                       # terms (see objective function)

param Gprop > 0;                       # Scale objective to simulate ratio
param Bprop > 0;                       # Gprop/Bprop of goods to bads

param Gratio := (Gprop/(Gprop+Bprop)) / (card {Good}/card {people});
param Bratio := (Bprop/(Gprop+Bprop)) / (card {Bad}/card {people});

###  VARIABLES  ###

var Wt_const;                          # Constant term in computing all scores

var Wt {j in factors} >= if wttyp[j] = 'pos' then 0 else -Infinity
                      <= if wttyp[j] = 'neg' then 0 else +Infinity;
                                       # Weights on the factors

var Sc {i in people};                  # Scores for the individuals
```

```
###  OBJECTIVE  ###

minimize penalty:                      # Sum of penalties for all individuals
   Gratio * sum {i in Good} << {k in 1..Gpce-1} if Gbktyp[k] = 'A'
                                   then Gbkfac[k]*app_amt
                                   else Gbkfac[k]*bal_amt[i];
                               {k in 1..Gpce} Gslope[k] >> Sc[i] +
   Bratio * sum {i in Bad}  << {k in 1..Bpce-1} if Bbktyp[k] = 'A'
                                   then Bbkfac[k]*app_amt
                                   else Bbkfac[k]*bal_amt[i];
                               {k in 1..Bpce} Bslope[k] >> Sc[i];

###  CONSTRAINTS  ###

def_Sc {i in people}:
   Sc[i] = Wt_const + sum {j in factors} answer[i,j] * Wt[j];
                                       # Score = sum of answers times weights
```

Appendix C. STRUC, a piecewise-linear structural design model

Given a set of admissible joints and a set of admissible bars connecting the joints, this model determines bar widths that minimize the total weight of the structure while maintaining an equilibrium with external loads [15, 28]. Each term of the objective is the length of a bar times the absolute value of the force (positive for tension, negative for compression) on the bar.

```
###  DATA  ###

set joints;
set bars within {i in joints, j in joints: i <> j};
                                     # Definition of admissible structure:
                                     # each bar connects two joints

param fixed symbolic in joints;      # Designated position of fixed support
param rolling symbolic in joints;    # Designated position of roller support

param density > 0;                   # Density of bar material
param yield_stress > 0;              # Yield stress of bar material

param xpos {joints};                 # Horizontal positions of joints
param ypos {joints};                 # Vertical positions of joints
   check {(i,j) in bars}: xpos[i] <> xpos[j] or ypos[i] <> ypos[j];

param xload {joints};                # Horizontal external loads on joints
param yload {joints};                # Vertical external loads on joints

param length {(i,j) in bars} :=
                      sqrt ((xpos[j]-xpos[i])^2 + (ypos[j]-ypos[i])^2);
                                     # Bar lengths calculated from positions

param xcos {(i,j) in bars} := (xpos[j]-xpos[i]) / length[i,j];
param ycos {(i,j) in bars} := (ypos[j]-ypos[i]) / length[i,j];
                                     # Cosines of bar angles with
                                     # horizontal and vertical axes


###  VARIABLES  ###

var Force {bars};                    # Forces on bars:
                                     # positive in tension, negative in compression


###  OBJECTIVE  ###

minimize weight:  (density / yield_stress) *
   sum {(i,j) in bars} length[i,j] * <<0; -1,+1>> Force[i,j];
                                     # Weight is proportional to length
                                     # times absoluted value of force


###  CONSTRAINTS  ###

subject to xbal {k in joints: k <> fixed}:
     sum {(i,k) in bars} xcos[i,k] * Force[i,k]
   - sum {(k,j) in bars} xcos[k,j] * Force[k,j] = xload[k];

subject to ybal {k in joints: k <> fixed and k <> rolling}:
     sum {(i,k) in bars} ycos[i,k] * Force[i,k]
   - sum {(k,j) in bars} ycos[k,j] * Force[k,j] = yload[k];
                                     # Net sum of forces must balance external
                                     # load, horizontally and vertically
```

Appendix D. TRAIN, a network allocation model

Given a day's schedule, this network flow model allocates passenger cars to trains so as to minimize either the number of cars required or the number of car-miles run [21, 22]. The same model is formulated in [18] using algebraic constraints.

```
###  SCHEDULE SETS AND PARAMETERS  ###

set cities;
set links within {c1 in cities, c2 in cities: c1 <> c2};
                        # Set of cities, and set of intercity links

param last > 0 integer;
set times circular := 1..last;
                        # Number of time intervals in a day, and
                        # set of time intervals in a day

set schedule within
      {c1 in cities, t1 in times,
       c2 in cities, t2 in times: (c1,c2) in links};
                        # Member (c1,t1,c2,t2) of this set represents
                        # a train that leaves city c1 at time t1
                        # and arrives in city c2 at time t2

###  DEMAND PARAMETERS  ###

param section > 0 integer;
                        # Maximum number of cars in one section of a train

param demand {schedule} > 0;
                        # For each scheduled train:
                        # the smallest number of cars that
                        # can meet demand for the train

param low {(c1,t1,c2,t2) in schedule} := ceil(demand[c1,t1,c2,t2]);
                        # Minimum number of cars needed to meet demand

param high {(c1,t1,c2,t2) in schedule}
   := max (2, min (ceil(2*demand[c1,t1,c2,t2]),
                   section*ceil(demand[c1,t1,c2,t2]/section) ));
                        # Maximum number of cars allowed on a train:
                        # 2 if demand is for less than one car;
                        # otherwise, lesser of
                        # number of cars needed to hold twice the demand, and
                        # number of cars in minimum number of sections needed

###  DISTANCE PARAMETERS  ###

param dist_table {links} >= 0 default 0.0;
param distance {(c1,c2) in links} > 0
   := if dist_table[c1,c2] > 0 then dist_table[c1,c2] else dist_table[c2,c1];
                        # Inter-city distances: distance[c1,c2] is miles
                        # between city c1 and city c2

###  OBJECTIVES  ###

minimize cars;          # Number of cars in the system:
                        # sum of unused cars and cars in trains during
                        # the last time interval of the day

minimize miles;         # Total car-miles run by all scheduled trains in a day
```

```
###  NODES  ###

node N {cities,times};  # For every city and time:
                        # unused cars in present interval will equal
                        # unused cars in previous interval,
                        # plus cars just arriving in trains,
                        # minus cars just leaving in trains

###  ARCS  ###

arc U {c in cities, t in times} >= 0
      from N[c,t]  to N[c,next(t)]
      obj {if t = last} cars 1;
                        # U[c,t] is the number of unused cars stored
                        # at city c in the interval beginning at time t

arc X {(c1,t1,c2,t2) in schedule}
      >= low[c1,t1,c2,t2]  <= high[c1,t1,c2,t2]
      from N[c1,t1]  to N[c2,t2]
      obj {if t2 < t1} cars 1
      obj miles distance[c1,c2];
                        # X[c1,t1,c2,t2] is the number of cars assigned
                        # to the scheduled train that leaves c1 at t1
                        # and arrives in c2 at t2
                        # The bounds insure that the cars meet demand,
                        # but that they are not so far in excess of demand
                        # that unnecessary sections are required
```

References

[1] E.M.L. Beale, 1970. Advanced Algorithmic Features for General Mathematical Programming Systems. In J. Abadie, ed., *Integer and Nonlinear Programming,* American Elsevier Publishing Company, New York, pp. 119–137.

[2] E.M.L. Beale and J.A. Tomlin, 1970. Special Facilities in a General Mathematical Programming System for Non-Convex Problems Using Ordered Sets of Variables. In J. Lawrence, ed., *OR 69: Proceedings of the Fifth International Conference on Operational Research,* Tavistock Publications, London, pp. 447–454.

[3] H.K. Bhargava and S.O. Kimbrough, 1993. Model Management: An Embedded Languages Approach. Technical report, Naval Postgraduate School, Monterey, CA; to appear in *Decision Support Systems.*

[4] J.J. Bisschop and R. Fourer, 1990. New Constructs for the Description of Combinatorial Optimization Problems in Algebraic Modeling Languages. Memorandum no. 901, Faculty of Applied Mathematics, University of Twente, Enschede, The Netherlands; to appear in *Annals of Operations Research.*

[5] J. Bisschop and A. Meeraus, 1982. On the Development of a General Algebraic Modeling System in a Strategic Planning Environment. *Mathematical Programming Study* **20,** 1–29.

[6] A. Brooke, D. Kendrick and A. Meeraus, 1992. *GAMS: A User's Guide, Release 2.25.* The Scientific Press, South San Francisco, CA.

[7] A. Charnes, W.W. Cooper and R.O. Ferguson, 1955. Optimal Estimation of Executive Compensation by Linear Programming. *Management Science* **1,** 138–151.

[8] V. Chvátal, 1983. *Linear Programming.* W.H. Freeman and Company, New York.

[9] CPLEX Optimization Inc., 1992. *Using the CPLEX Callable Library and CPLEX Mixed Integer Library.* Incline Village, NV.

[10] G.B. Dantzig, 1956. Recent Advances in Linear Programming. *Management Science* **2,** 131–144.

[11] G.B. Dantzig, 1960. On the Significance of Solving Linear Programming Problems with Some Integer Variables. *Econometrica* **28,** 30–44.

[12] G.B. Dantzig, S. Johnson and W. White, 1958. A Linear Programming Approach to the Chemical Equilibrium Problem. *Management Science* **5,** 38–43.

[13] N. Dean, M. Mevenkamp and C.L. Monma, 1992. NETPAD: An Interactive Graphics System for Network Modeling and Optimization. In O. Balci, R. Sharda and S.A. Zenios, eds., *Computer Science and Operations Research: New Developments in Their Interfaces,* Pergamon Press, New York, pp. 231–243.

[14] D. De Wolf, O. Janssens de Bisthoven and Y. Smeers, 1991. The Simplex Algorithm Extended to Piecewise-Linearly Constrained Problems I: the Method and an Implementation. Discussion paper, Center for Operations Research and Econometrics, Louvain-La-Neuve, Belgium.

[15] W.S. Dorn, R.E. Gomory and H.J. Greenberg, 1964. Automatic Design of Optimal Structures. *Journal de Mécanique* **3,** 25–52.

[16] R. Fourer, 1983. Modeling Languages versus Matrix Generators for Linear Programming. *ACM Transactions on Mathematical Software* **9,** 143–183.

[17] R. Fourer, 1992. A Simplex Algorithm for Piecewise-Linear Programming III: Computational Analysis and Applications. *Mathematical Programming* **53,** 213–235.

[18] R. Fourer, D.M. Gay and B.W. Kernighan, 1987. AMPL: A Mathematical Programming Language. Computing Science Technical Report 133, AT&T Bell Laboratories, Murray Hill, NJ; also Technical Report 87-03, Department of Industrial Engineering and Management Sciences, Northwestern University, Evanston, IL.

[19] R. Fourer, D.M. Gay and B.W. Kernighan, 1990. A Modeling Language for Mathematical Programming. *Management Science* **36,** 519–554.

[20] R. Fourer, D.M. Gay and B.W. Kernighan, 1992. *AMPL: A Modeling Language for Mathematical Programming.* The Scientific Press, South San Francisco, CA.

[21] R. Fourer, J.B. Gertler and H.J. Simkowitz, 1977. Models of Railroad Passenger-Car Requirements in the Northeast Corridor. *Annals of Economic and Social Measurement* **6,** 367–398.

[22] R. Fourer, J.B. Gertler and H.J. Simkowitz, 1978. Optimal Fleet Sizing and Allocation for Improved Rail Service in the Northeast Corridor. *Transportation Research Record* **656,** 40–45.

[23] R. Fourer and R.E. Marsten, 1992. Solving Piecewise-Linear Programs: Experiments with a Simplex Approach. *ORSA Journal on Computing* **4,** 16–31.

[24] D.M. Gay, 1985. Electronic Mail Distribution of Linear Programming Test Problems. *Committee on Algorithms Newsletter* **13,** 10–12. Also Numerical Analysis Manuscript 86-0, AT&T Bell Laboratories, Murray Hill, NJ.

[25] F. Glover, D. Klingman and N. Phillips, 1990. Netform Modeling and Applications. *Interfaces* **20:**4, 7–27.

[26] F. Glover, D. Klingman and N.V. Phillips, 1992. *Network Models in Optimization and their Applications in Practice.* Wiley, New York.

[27] H.J. Greenberg and F.H. Murphy, 1992. Views of Mathematical Programming Models and Their Instances. Technical report, University of Colorado at Denver, Denver, CO.

[28] J.K. Ho, 1975. Optimal Design of Multi-Stage Structures: A Nested Decomposition Approach. *Computers & Structures* **5,** 249–255.

[29] M.S. Hung, W.O. Rom and A.D. Waren, 1993. *Optimization with OSL.* The Scientific Press, South San Francisco, CA.

[30] J.P. Jarvis and D.R. Shier, 1990. Netsolve: Interactive Software for Network Optimization. *Operations Research Letters* **9,** 275–282.

[31] C.V. Jones, 1990. An Introduction to Graph-Based Modeling Systems, Part I: Overview. *ORSA Journal on Computing* **2,** 136–151.

[32] C.V. Jones, 1991. An Introduction to Graph-Based Modeling Systems, Part II: Graph-Grammars and the Implementation. *ORSA Journal on Computing* **3,** 180–206.

[33] C. Jones and K. D'Souza, 1992. Graph-Grammars for Minimum Cost Network Flow Modeling. Faculty of Business Administration, Simon Fraser University, Burnaby, British Columbia, Canada.

[34] G. Kahan, 1982. Walking through a Columnar Approach to Linear Programming of a Business. *Interfaces* **12:**3, 32–39.

[35] D.A. Kendrick, 1990. Parallel Model Representations. *Expert Systems With Applications* **1,** 383–389.

[36] D.A. Kendrick, 1991. A Graphical Interface for Production and Transportation System Modeling: PTS. *Computer Science in Economics and Management* **4.**

[37] H.M. Markowitz and A.S. Manne, 1957. On the Solution of Discrete Programming Problems. *Econometrica* **25,** 84–110.

[38] B.A. Murtagh and M.A. Saunders, 1987. MINOS 5.1 User's Guide. Technical report SOL 83-20R, Department of Operations Research, Stanford University, Stanford, CA.

[39] W. Ogryczak, K. Studziński and K. Zorychta, 1992. EDINET — A Network Editor for Transshipment Problems with Facility Location. In O. Balci, R. Sharda and S.A. Zenios, eds., *Computer Science and Operations Research: New Developments in Their Interfaces,* Pergamon Press, New York, pp. 197–212.

[40] R.V. Simons, 1987. Mathematical Programming Modeling Using MGG. *IMA Journal of Mathematics in Management* **1,** 267–276.

[41] D. Steiger, R. Sharda and B. LeClaire, 1992. Functional Description of a Graph-Based Interface for Network Modeling (GIN). In O. Balci, R. Sharda and S.A. Zenios, eds., *Computer Science and Operations Research: New Developments in Their Interfaces,* Pergamon Press, New York, pp. 213–229.

[42] J.A. Tomlin, 1970. Branch and Bound Methods for Integer and Non-Convex Programming. In J. Abadie, ed., *Integer and Nonlinear Programming,* American Elsevier Publishing Company, New York, pp. 437–450.

[43] J.S. Welch, Jr., 1987. PAM—A Practitioner's Approach to Modeling. *Management Science* **33,** 610–625.

[44] S.A. Zenios, 1990. Integrating Network Optimization Capabilities into a High-Level Modeling Language. *ACM Transactions on Mathematical Software* **16,** 113–142.