New Constructs for the Description of Combinatorial Optimization Problems in Algebraic Modeling Languages

J.J. Bisschop
Robert Fourer

Department of Industrial Engineering and Management Sciences

Northwestern University, Evanston, Illinois 60208-3119

November 1990
Revised December 1994
Revised February 1995

ABSTRACT

Algebraic languages are at the heart of many successful optimization modeling systems, yet they have been used with only limited success for combinatorial (or discrete) optimization. We show in this paper, through a series of examples, how an algebraic modeling language might be extended to help with a greater variety of combinatorial optimization problems. We consider specifically those problems that are readily expressed as the choice of a subset from a certain set of objects, rather than as the assignment of numerical values to variables. Since there is no practicable universal algorithm for problems of this kind, we explore a hybrid approach that employs a general-purpose subset enumeration scheme together with problem-specific directives to guide an efficient search.

Published as:

J.J. Bisschop and Robert Fourer,
New Constructs for the Description of Combinatorial
Optimization Problems in Algebraic Modeling Languages.
Computational Optimization and Applications **6** (1996) 83–116.

1. Introduction

Algebraic modeling languages provide a natural and familiar means of describing mathematical programming models to computer systems. These languages express the objective function as an algebraic expression in parameters and variables, and the constraints similarly as algebraic equations and inequalities.

Algebraic languages are at the heart of many successful optimization modeling systems, including AIMMS [2], AMPL [13, 14], GAMS [3, 4], LINGO [34], and MPL [32]. Much of the popularity of these systems can be regarded as deriving from two strengths that algebraic languages possess.

First, although originally developed for linear programming, algebraic modeling languages can be extended in a natural way to describe other important problem classes. Thus they have long been widely used in nonlinear and integer programming, and more recently have been extended in useful ways for network flow optimization [2, 11] and for complementarity problems [9, 10, 41].

Second, although the problems that can be described by an algebraic modeling language vary greatly in structure and origin, they can be translated to a very general mathematical form for which general-purpose algorithms are available. As a result, optimization systems based on algebraic languages can reliably solve many different problem types without much direction from the user.

These advantages notwithstanding, there remains one very important area of mathematical programming in which algebraic modeling languages have had only limited success: combinatorial (or discrete) optimization. The key obstacle in this case is not a matter of language generality — as we will show in this paper — but is rather a lack of general-purpose algorithms. The preponderance of research in combinatorial optimization has been concerned with narrowly targeted algorithms for which performance bounds can be derived, and with heuristics tailored for particular problem classes. Both complexity theory and computational experience suggest, moreover, that efficient general-purpose algorithms for combinatorial optimization are unlikely to exist.

The only major success of algebraic modeling languages in combinatorial optimization has been for those problems that can be formulated as linear integer programs. These can be translated to a general mathematical form suitable for input to sophisticated implicit enumeration (or branch-and-bound) procedures. With sufficient attention to the tightness of the formulation and the parameters of the enumeration scheme, general-purpose packages can achieve acceptable results for many kinds of problems. Even so, useful information is often lost in a model's translation from its original formulation to a solvable formulation in terms of integer decision variables. Without this information, the model is harder to understand, and often the problem is harder to solve.

Many other instances of combinatorial optimization cannot be reliably or efficiently solved through any general-purpose integer programming approach. This is the case for problems as elementary as minimum spanning tree, and as famous as traveling salesman, as well as much more complicated problems in network design, routing, and packing. Successful attacks on problems of these kinds have relied on highly problem-specific software, either written from scratch or built on top of general-purpose branch-and-bound and linear programming libraries such as CPLEX [7] and OSL [23]. Algebraic modeling languages have been of very limited help here.

The aim of this paper is to show, through a series of examples, how an algebraic modeling language might be extended to help with a greater variety of combinatorial optimization problems. We specifically consider those problems that are readily expressed as the choice of a subset from a certain set of objects, rather than as the assignment of numerical values to variables. This broad class, which includes some problems that can be expressed usefully (though less naturally) as integer programs and others that cannot, is not amenable to any known algorithmic procedure that is both general and efficient. We thus explore a hybrid approach that employs a general-purpose subset enumeration scheme, together with

problem-specific directives to guide an efficient search for an optimal subset.

To introduce the fundamentals of our approach, we use in Section 2 a very simple knapsack example. Then in Section 3 we discuss general-purpose subset enumeration schemes in more detail. Sections 4 and 5 are devoted to two examples that exhibit more complex features: a budgeted traveling salesman problem that requires a choice of an ordered subset, and a bulk loading problem for which a subset must be selected in a certain way from a set of pairs.

Our work represents only a start in this area. Considerable research and development remains to be done before modeling languages for combinatorial optimization attain the same success as languages for other areas of mathematical programming. In Section 6 we seek to place our current ideas into perspective, by describing and contrasting a variety of approaches to the design of modeling systems for combinatorial optimization. We consider algebraic modeling languages that may accommodate a broader range of logical and combinatorial expressions — through automatic translation to integer programs, or extension of current branch-and-bound schemes — as well as alternative logic-based and network-based model representations. We conclude by suggesting the circumstances in which the approach that we set forth in this paper is most likely to be successful.

2. A Simple Knapsack Problem

This section introduces our ideas through the use of a very simple knapsack problem. We begin by contrasting the problem's integer programming formulation with its more natural formulation in terms of optimization over subsets. We then describe general principles of subset enumeration that can be applied in solving the subset formulation directly. Finally, we propose directives to guide fathoming, bounding and search strategy for an implicit enumeration of subsets, and describe some updating directives that may streamline the enumeration's description and operation.

Our illustrations throughout this paper are based on the AMPL modeling language [13, 14]. We show in particular the symbolic parts of AMPL models, which declare the relevant sets and parameters and provide a general algebraic formulation. In practice the modeler would also supply set and parameter data to define individual problem instances to be solved.

Formulations

Given a collection of objects $i \in \mathcal{S}$ having weights $w_i > 0$ and values $v_i > 0$, we want to find the most valuable subset whose total weight does not exceed a given "knapsack" capacity W.

A standard algebraic integer programming formulation for this problem defines a variable x_i to be 1 if object i is included in the subset, or 0 otherwise. Then it suffices to

$$\begin{array}{ll} \text{Maximize} & \sum_{i \in \mathcal{S}} v_i x_i \\ \text{Subject to} & \sum_{i \in \mathcal{S}} w_i x_i \le W \end{array}$$

We can dispense with the zero-one variables, however, by instead expressing the problem as one of optimizing over all subsets $\mathcal{T} \subseteq \mathcal{S}$:

$$\begin{array}{ll} \underset{\mathcal{T} \subseteq \mathcal{S}}{\text{Maximize}} & \sum_{i \in \mathcal{T}} v_i \\ \text{Subject to} & \sum_{i \in \mathcal{T}} w_i \le W \end{array}$$

This is arguably a superior description of the problem, since it more directly says what was originally intended.

It is easy to transcribe the algebraic integer programming formulation of the knapsack problem to a model in the AMPL language, as shown in Figure 2–1; the AMPL set `OBJECTS`

```
set OBJECTS;

param value {OBJECTS} > 0;
param weight {OBJECTS} > 0;

param capacity > 0;

var X {OBJECTS} logical;

maximize Total_Value: sum {i in OBJECTS} value[i] * X[i];

subject to Weight_Limit:
     sum {i in OBJECTS} weight[i] * X[i] <= capacity;
```

Figure 2–1. *An integer programming formulation of the knapsack problem,* in the AMPL algebraic modeling language.

```
set OBJECTS;

param value {OBJECTS} > 0;
param weight {OBJECTS} > 0;

param capacity > 0;

var_set Knapsack within OBJECTS;

maximize Total_Value: sum {i in Knapsack} value[i];

subject to Weight_Limit:
     sum {i in Knapsack} weight[i] <= capacity;
```

Figure 2–2. *A decision-set formulation of the knapsack problem,* using the proposed new `var_set` declaration.

plays the role of $\mathcal{S}$ in our formulation. Currently AMPL has no way to describe a maximization over subsets, however, rather than over values of variables. For this purpose we propose to introduce a `var_set` declaration such as the following:

```
var_set Knapsack within OBJECTS;
```

This says that `Knapsack` is a subset of `OBJECTS`, whose membership is to be determined. It may be regarded as a *decision set,* in contrast to the decision variables `X[i]` of the conventional formulation.

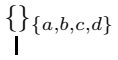
The decision set `Knapsack` plays the role of $\mathcal{T}$ in our mathematical formulation. Thus the full alternative AMPL model can be written as shown in Figure 2–2. Aside from the introduction of the `var_set` declaration, the same AMPL syntactic forms and semantic conventions are employed.

Although this second AMPL formulation is easier to write and understand, it is harder to translate and solve. As one possibility, the language translator could convert it automatically to a zero-one integer programming problem, which could then be submitted to a standard branch-and-bound solver. While it is clear how to make such a transformation in this very simple case, however, formulations using `var_set` do not in general admit any usefully concise and tight equivalents in terms of integer variables, let alone equivalents that can be generated automatically. Examples of more challenging `var_set` declarations are considered in Sections 4 and 5.

As an alternative, we imagine that the user is given the option of specifying an implicit enumeration scheme tailored to the `var_set` in the problem at hand. This scheme is described by additional directives that use AMPL constructs, but that are not part of the model description itself.

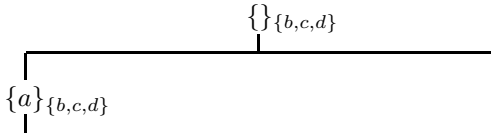
Search principles

To motivate the directives that we will propose, we begin by considering how the subsets of a given set may be enumerated in a systematic way. There is more than one approach to such an enumeration, but for now we imagine that a tree-structured collection of the subsets is "built up" from the empty set. Thus we start with a tree that consists of only a root node:



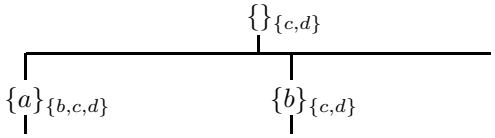
We associate two sets with each node: a *build set* that is one of the subsets that has been built up, and a *free set* of objects available to be added (shown in small type). Thus the above diagram shows that the build set associated with the root is the empty set, while the free set is initially the whole set $\{a, b, c, d\}$.

To create the first child of the root, an object is chosen from the free set (by some appropriate rule, as explained later). Suppose that we choose a. Then the build set of the child becomes the set $\{a\}$:

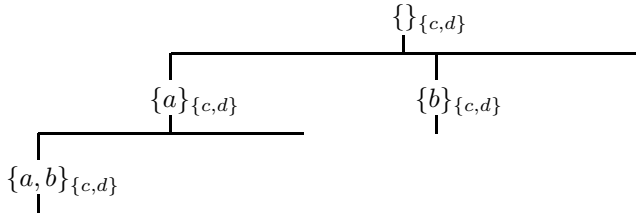


Object a is removed from the the root's free set, which becomes also the free set of the child.

We can next choose an object in either of two ways. If we choose, say, b from the free set of the root, we create a second child at depth one:



The build set of this child becomes $\{b\}$, and b is removed from the root's free set, which becomes also the free set of the new child. We may also choose b from the free set of the first child, in which case a child at depth 2 is created:



The associated build set is $\{a, b\}$; object b is removed from the free set of the parent at depth 1, and the resulting set also becomes the free set of the new child.

Subsequent steps proceed similarly. An object is chosen from the free set of some existing node. A new child of that node is created, with a build set equal to the parent's build set plus the chosen object. The chosen object is then removed from the parent's free set, which becomes also the child's free set.

By continuing this process until all free sets are empty, we construct a tree such as the one depicted in Figure 2–3. Different priorities for creating new branches and nodes give rise to somewhat different trees, but all have the same essential properties. Each node is associated with one of the subsets of $\{a, b, c, d\}$, and the nodes at depth k represent all the subsets of k objects. The tree is asymmetric because the ordering of the objects within subsets is inessential. (We will observe in Section 3 how this search tree can be regarded as a special case of the standard binary branch-and-bound tree.)

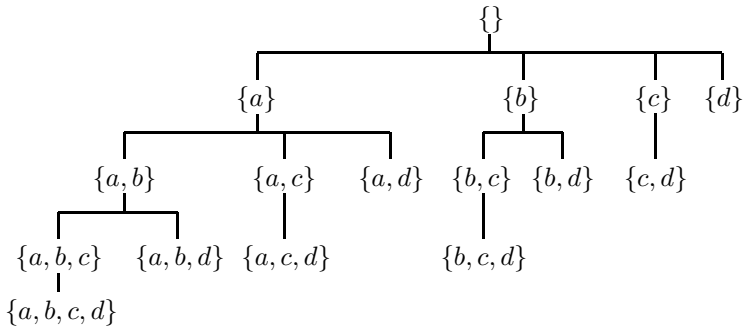


Figure 2–3. *A search tree for all subsets of* $\{a, b, c, d\}$. This particular tree is realized when a is given highest priority, b is given next-highest priority, and so forth.

Based on the construction of this sort of tree, we can design an *explicit* enumeration algorithm to solve any optimization problem over subsets. Upon the creation of each new node, the associated build set is evaluated for feasibility and (if feasible) for objective value, and it is saved as the incumbent if it gives a better objective than any subset previously seen. The branching priority is irrelevant, since we require only that every subset be built eventually.

A practical *implicit* enumeration scheme constructs the same subset tree, but with much greater care. Node and branch selection strategies, bounds on the optimal value, and other criteria are applied to greatly reduce the number of nodes actually created. As our previous remarks have suggested, however, we cannot expect to find useful general-purpose implicit enumeration criteria for combinatorial optimization problems formulated in terms of decision sets. To achieve acceptable results, the enumeration criteria must instead be designed specially for individual problem types.

We thus propose to introduce a collection of directives through which the modeler can specify problem-specific implicit enumeration criteria. These directives use many of the syntactic elements of the modeling language, but they are not part of the symbolic model or of the data. They do not change what is be solved, but rather determine how the implicit enumeration routines will search for a solution.

The remainder of this section introduces the most important directives, for the case of the knapsack problem. So that we may refer to the enumeration tree as it is built, a first directive must associate names with the build set and free set. Recall that the `var_set` we are enumerating is `Knapsack`. At any node, the build set is one of the subsets of objects that we might place in the knapsack; the free set contains the objects that are outside of the knapsack but are still eligible to be added. Thus our directive is:

```
enumerate Knapsack: build_set Inside,
                    free_set  Outside;
```

The identifiers `Inside` and `Outside` will serve as generic names for the build set and free set respectively. Within our directives, they will participate in expressions like other sets of the modeling language.

Fathoming

If the weight of the build set at a node exceeds the specified capacity, then no feasible solution is associated with the node. Moreover, no feasible solution is associated with any of the node's descendants, all of which correspond to even larger build sets. Thus the node and all of the search tree below it can be removed; in the usual branch-and-bound terminology, the node can be fathomed.

The above conditions are easily converted into a directive that uses the AMPL language's terminology:

```
fathom at child node: sum {i in Inside} weight[i] > capacity;
```

This is interpreted as stating that the specified feasibility condition is to be checked at each node upon its initial creation in the search tree. In the presence of such a directive, the enumeration can proceed much as before, but it will avoid generating potentially large sub-trees of infeasible solutions.

A more concise directive,

```
fathom at child node: Weight_Limit.slack < 0;
```

makes an equivalent statement, that any child node should be fathomed if its build set violates the `Weight_Limit` constraint.

Bounding

More powerful fathoming criteria often depend on bounding arguments. Thus it is desirable to have an alternative fathoming directive that simply gives a formula for a bound to be used.

In the case of a maximization, we seek an upper bound on the objective that may be achieved at a node or any of its descendants. For the knapsack problem, this means an upper bound on the total value of objects in a node's build set plus current free set. As an example, we may imagine that our knapsack is filled exactly to capacity by adding (to the build set) a fictitious object that has the greatest value-to-weight ratio of any in the free set. The resulting total value is an upper bound on the value that can actually be achieved by any feasible solution that adds zero or more objects from the free set to the build set.

To express this bound algebraically, we write the current total value of objects in the build set as `sum {i in Inside} value[i]`, and the remaining capacity of the knapsack as `capacity - sum {i in Inside} weight[i]`. The largest value-to-weight ratio in the free set can be written using AMPL's iterated maximization operator: `max {i in Outside} value[i] / weight[i]`. Putting these together, we arrive at the following directive:

```
bound at child node:
    sum {i in Inside} value[i] +
        (capacity - sum {i in Inside} weight[i]) *
             (max {i in Outside} value[i] / weight[i]);
```

This specifies an additional activity that is to be carried out whenever a new node is added to the tree. The bound expression is evaluated, and is compared with the objective achieved by the current incumbent solution. If the bound is less than the incumbent value, the node is fathomed.

We may prefer that the bound be computed every time a node is visited (just prior to becoming a parent) rather than only the first time. There are two reasons to favor such a strategy: the bound may decline as the free set shrinks, and the incumbent solution may increase as the enumeration proceeds. These phenomena may cause the node to be fathomed at a later visit, even though it could not be fathomed originally. To provide this option we can use the same directive, except with `parent` replacing `child`.

Branch selection

So far, we have allowed the enumeration scheme to create new nodes in any arbitrary order. We can hope to speed the implicit enumeration by providing directives to influence the ordering.

We can view branch selection as having two aspects: selection of the parent, and selection of the child. In the present example, we might want to specify a depth-first search, in which the selected parent should have the largest build set among all those available for branching. We could reasonably take the child as one that corresponds to adding an object with the largest available value-to-weight ratio.

To design directives for branch and node selection, we must make a few extensions to AMPL terminology. The language has a way to specify a maximum over a set (as in the `bound` directive above) but not over a collection of active nodes in a search tree. To distinguish the latter, we introduce a new keyword, `largest`, in the following directive:

```
select parent node: largest card {Inside};
```

This causes the expression `card {Inside}` — representing the cardinality of, or number of members in, the build set `Inside` — to be evaluated at each node where the free set is not yet empty. The node that has the largest such value is the one from which the search will branch.

Once a parent node is selected, it is a straightforward matter to specify the child that will be added. We need only indicate which member of the free set will be added to the build set. The following directive specifies an object having greatest ratio of value to weight:

```
select child node:
    arg max {i in Outside} value[i] / weight[i];
```

Here we have invented another AMPL construct, `arg max`, to return the set of objects for which the maximum is achieved. (The standard AMPL `max` operator returns only the value of the maximum.)

Neither of these directives necessarily specifies a node uniquely. If not, we assume that the system makes an arbitrary choice of node among the ones specified.

Updating

It can be inefficient to recompute all the quantities referenced by the directives at each node. Instead, certain quantities required at a new child node can be updated from their values at the parent.

As an example, consider the expression `sum {i in Inside} value[i]` in the `bound` directive above. We can call this `current_value`, and direct that it be initialized to zero at the root node (where the build set is empty):

```
param at root node: current_value := 0;
```

Now we must say that at each new node, our `sum` is increased by the value of the object newly added to the build set:

```
param at child node:
    current_value := current_value + value[last(Inside)];
```

Here we employ the function `last(Inside)`, which selects the element most recently added to the build set, `Inside`. In effect, we treat `Inside` as an AMPL ordered set, with its members being ordered in the sequence that objects were added to it.

We can similarly define `cap_avail` to represent the capacity not yet used up by the build set:

```
param at root node: cap_avail := capacity;
param at child node:
    cap_avail := cap_avail - weight[last(Inside)];
```

Then our previous fathoming and bounding directives can be rewritten more concisely like this:

```
fathom at child node: cap_avail < 0;
bound at child node: current_value +
    cap_avail * (max {i in Outside} value[i] / weight[i]);
```

Besides being more efficient to carry out, these directives are easier to understand when written in this way.

3. Alternative Search Principles

In the preceding knapsack example, we presented one approach to enumerating a domain of unordered subsets. We now observe that this approach embodies most of the features of the more familiar binary branch-and-bound enumeration tree. We also describe a family of enumeration schemes that are similar in spirit, but that can build from subsets other than the empty set.

The branch-and-bound search

An example of the standard branch-and-bound search tree is depicted in Figure 3–1. There are two nodes below the root: the one labeled $\{a\}$ represents a commitment to put a in the chosen subset, while the one labeled $\{a'\}$ represents a commitment *not* to put a in the subset. If we were using zero-one variables (as in the initial formulation in the previous section) we would say that nodes $\{a\}$ and $\{a'\}$ represent the actions of fixing the corresponding variable to 0 and to 1.

Below $\{a\}$ are a pair of nodes, $\{a, b\}$ and $\{a, b'\}$, which represent a commitment to add or not add b to the subset; and similarly below $\{a'\}$ are nodes $\{a', b\}$ and $\{a', b'\}$. At the kth level there are 2^k subsets, representing all possibilities for choosing k of the set members, or equivalently for fixing k of the 0–1 variables.

It is easy to see how the *binary tree* of Figure 3–1 can be "collapsed" to a *subset tree* like that in Figure 2–3. First drop the a', b', etc. from the node labels, so that for example $\{a, b\}$, $\{a, b'\}$, $\{a', b\}$, $\{a', b'\}$ become $\{a, b\}$, $\{a\}$, $\{b\}$, and $\{\}$. Then combine adjacent nodes that have the same labels, deleting their connecting arcs.

Viewed in this way, the subset tree appears to contain less information. It shows only what is committed to be *in* the enumerated subset — we refer to its node labels as "build" sets — whereas the binary tree's nodes also record what is committed to be *not in* the subset. Nevertheless, our procedure for using the subset tree in enumeration can be seen, on closer examination, to incorporate most features of the enumeration of the full binary tree.

Consider first the root node of the subset tree. When we begin the enumeration described in Section 2, the root node is associated with a free set containing all the possible members — in our example, $\{a, b, c, d\}$. When the child $\{a\}$ is formed, however, the member a is deleted from the root's free set. By this deletion, we insure that a will not be a member of any of the subsets represented by other children of the root; in other words, the root now plays the role of node $\{a'\}$ in Figure 3–1. Subsequently, when node $\{b\}$ is formed under the root, the root's free set becomes just $\{c, d\}$, and the root plays the role of $\{a', b'\}$. Continuing in this way, we see that the root node of this tree will eventually play the role of all the nodes from the binary tree that are collapsed into it. The same is true of the other subset tree nodes; for example, after $\{a, b\}$ is created, the node $\{a\}$ plays the role of $\{a, b'\}$.

To say this more generally, let $\mathcal{S}$ be the entire set whose subsets are being enumerated, and let $\mathcal{B}$ and $\mathcal{F}$ be the build set and current free set of some node in the subset tree. Then

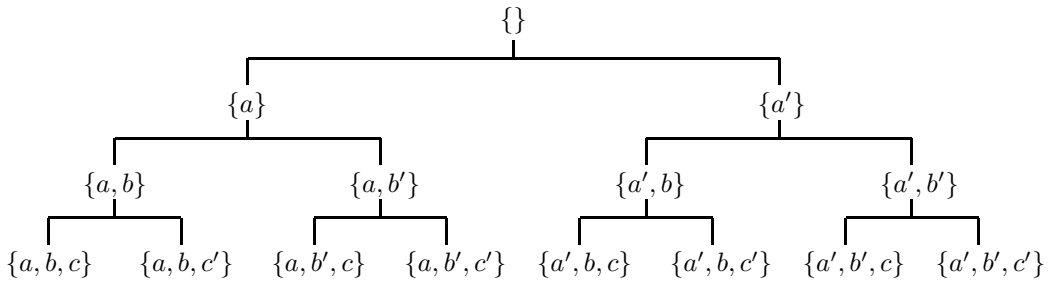


Figure 3–1. *Binary search tree* for all subsets of $\{a, b, c\}$.

we can regard that node as playing the role of

$$\mathcal{B} \cup [\mathcal{S} \backslash (\mathcal{B} \cup \mathcal{F})]'$$

in the binary tree, in the sense that it represents a commitment to include all the members of $\mathcal{B}$ and exclude all those of $\mathcal{S} \backslash (\mathcal{B} \cup \mathcal{F})$.

If only directives such as `bound at child node` and `fathom at child node` are employed, then our implicit enumeration recognizes only the role played by each node when it is first created (as a child of some higher node). To incorporate rules that recognize a node's subsequent roles, when it is revisited as a parent, we must use the previously described `bound at parent node`, or its analogue `fathom at parent node`. Even then, the commitment to include a certain object will always be tested before the commitment to exclude it. For example, our enumeration process branches explicitly from the parent $\{a\}$ to the child $\{a, b\}$, but can only consider $\{a, b'\}$ when it returns to the parent at a later step.

In conclusion, we can use the subset tree of Figure 2–3 tree in almost the same way that the binary tree of Figure 3–1 tree would customarily be used. The latter might be preferable because it offers a little more flexibility in committing to exclude a set member, or because it offers a more attractive way to think about some problems; if so, we could readily devise AMPL directives that work directly with it. We have chosen to instead use the subset tree in the knapsack example because it is more compact, and because it generalizes to the search tree needed for ordered subsets — which we investigate in Section 4 below.

Searching from a given set

We have assumed so far that it is desirable to start from an empty set and to "build" toward larger subsets as the tree is searched at greater depths. In many cases, however, a reasonably good choice of subset is already known, or can be cheaply computed. One can thus imagine that it might be desirable to be able to "start" the implicit enumeration from a given subset. We next suggest how this might be done within the framework that we have established.

Suppose that we are given a feasible subset for the knapsack problem. It might be declared in AMPL as a set named `GUESS`:

```
set GUESS within OBJECTS;
    check sum {i in GUESS} weight[i] <= capacity;
```

To create a search tree starting from this set, we need only take its symmetric difference with each of the node labels in the subset tree defined by Section 2. (The symmetric difference of sets $\mathcal{A}$ and $\mathcal{B}$ is $(\mathcal{A} \cup \mathcal{B}) \backslash (\mathcal{A} \cap \mathcal{B})$.) For the example of Figure 2–3, with `GUESS` taken as $\{b, c\}$, the result is as shown in Figure 3–2. Every possible subset still labels exactly one of the nodes, but the arrangement is different; at depth k, all subsets differ by k members from the root subset.

This generalized search tree can be implemented through AMPL directives by allowing an initial value to be assigned to the build set:

```
enumerate knapsack: build_set Inside := GUESS,
                    free_set  Free_to_Change;
```

The initial free set at the root is still the set of all objects, and free sets are modified and propagated as before. The free set at a node also still determines which build sets may be generated by branching from that node, but in a more general way. As before, an object in the free set may be *added* to the build set, provided that it was not in the root build set. But also, an object in the free set may be *dropped* from the build set, if it was in the root build set. These two different kinds of objects in the free set can be distinguished by defining:

```
set at child node: Free_to_Add  := Free_to_Change diff GUESS;
set at child node: Free_to_Drop := Free_to_Change int GUESS;
```

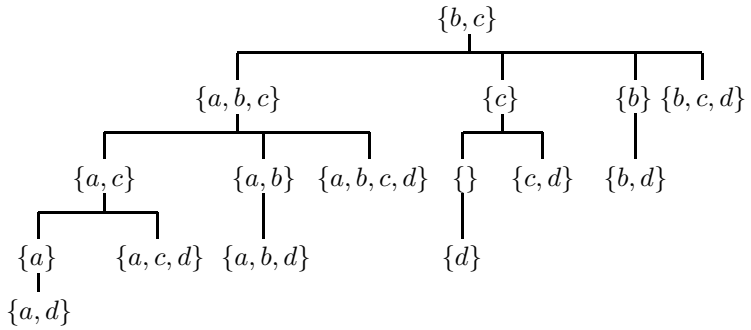


Figure 3–2. *Search tree for all subsets of* $\{a, b, c, d\}$, starting from $\{b, c\}$.

(The operators `diff` and `int` are AMPL's abbreviations for set difference and intersection.) To save the user the trouble of setting this up, we could provide an option to define these two additional sets in the `enumerate` directive.

To accommodate this generalization, we must also generalize Section 2's fathoming and bounding conditions. A node may be fathomed if its build set exceeds the knapsack capacity *after* removal of all objects from the free set that are eligible to be dropped:

```
fathom at child node:
    sum {i in Inside diff Free_to_Drop} weight[i] > capacity;
```

When this condition is satisfied, the build set is infeasible for the knapsack, and so are all subsets associated with its descendant nodes.

To generalize Section 2's bound, we must similarly consider that some objects in a node's build set may be removed at descendant nodes. In particular, we must account for the possibility of dropping those objects in `Free_to_Drop` that have a lower value-to-weight ratio than the highest-ratio object in `Free_to_Add`. This leads to:

```
param at child node: best_ratio :=
    if card {Free_to_Add} = 0 then 0
    else max {i in Free_to_Add} value[i] / weight[i];

set at child node: Keep_Inside := Inside diff
    {i in Free_to_Drop: value[i] / weight[i] < best_ratio};

bound at child node:
    sum {i in Keep_Inside} value[i] + best_ratio *
          (capacity less sum {i in Keep_Inside} weight[i]);
```

A few complications here are necessary to assure that a valid bound is computed at nodes where `Free_to_Add` is empty, or where the knapsack remains overfilled after dropping the lower-ratio objects. (In AMPL, `a less b` means the greater of `a - b` and zero.)

Finally, we can consider a variety of branching directives. If it is expected that most objects in the initial set will indeed be found in the optimal set, then we may prefer to branch from parents where fewer of the initial objects have been dropped:

```
select parent node: smallest card {GUESS diff Inside};
```

Or, we may want to branch from parents that are closer to the initial subset:

```
select parent node: smallest card {GUESS symdiff Inside};
```

This latter rule is just a breadth-first search for our generalized subset tree. (AMPL's `symdiff` operator gives the previously defined symmetric difference.)

As before, we can specify branching to the child node that adds an object having the largest available value-to-weight ratio:

```
select child node:
    arg max {i in Free_to_Add} value[i] / weight[i];
```

If `Free_to_Add` is empty, then this directive is ignored, and the implicit enumeration procedure selects some arbitrary child from `Free_to_Drop`. As a refinement, we can specify a secondary preference for the child node that drops an object having the *smallest* available value-to-weight ratio:

```
select child node: if card {Free_to_Add} > 0
       then arg max {i in Free_to_Add} value[i] / weight[i]
       else arg min {i in Free_to_Drop} value[i] / weight[i];
```

Further refinements to the branching (as well as bounding and fathoming) criteria are possible, for example by distinguishing the cases of feasible and infeasible build sets.

4. The Budgeted Traveling Salesman Problem

We next consider a more difficult case, typical of routing problems. The solution consists of a choice of an *ordered* subset, or subsequence.

We begin by stating the problem and considering its formulation in terms of both decision variables and decision sets, with the latter transcribed to AMPL. We then discuss the principles underlying the enumeration of subsequences, and consider the various ways in which the previously described directives can be used to specify a good implicit enumeration.

Formulation

We are given a set $\mathcal{C}$ of cities, and a home city $h \in \mathcal{C}$. Travel between cities is restricted to a subset of links $\mathcal{L} \subseteq \mathcal{C} \times \mathcal{C}$: if $(i,j) \in \mathcal{L}$, then it is possible to travel from i to j, at a cost c_{ij} ($\neq c_{ji}$ in general). We want to find a tour through the home city that visits as many other cities as possible, subject to a budget B for travel costs.

We can formulate this problem as a zero-one integer program, but only through the use of some tricks. Naturally there is a variable x_{ij} for each $(i,j) \in \mathcal{L}$, equal to 1 if link (i,j) is used by the tour, and zero otherwise. A new city is visited with each link traversed, so the objective is

$$\text{Maximize} \sum_{(i,j)\in\mathcal{L}} x_{ij},$$

and the budget constraint is enforced by

$$\sum_{(i,j)\in\mathcal{L}} c_{ij} x_{ij} \le B.$$

We add constraints to assure that the tour leaves and re-enters the home city,

$$\sum_{(h,j)\in\mathcal{L}} x_{hj} = 1, \quad \sum_{(i,h)\in\mathcal{L}} x_{ih} = 1,$$

and that it leaves every other city the same number of times that it enters:

$$\sum_{(i,k)\in\mathcal{L}} x_{ik} = \sum_{(k,j)\in\mathcal{L}} x_{kj}, \quad \text{for all } k \in \mathcal{C}\backslash\{h\}.$$

Finally, we eliminate additional subtours that do not visit the home city, by specifying

$$\begin{aligned} &u_i - u_j + |\mathcal{C}|\, x_{ij} \le |\mathcal{C}| - 1, && \text{for all } (i,j) \in \mathcal{L} \text{ such that } j \ne h, \\ &0 \le u_i \le |\mathcal{C}|, && \text{for all } i \in \mathcal{C}. \end{aligned}$$

```
set CITIES;
param Home symbolic in CITIES;

set LINKS within {i in CITIES, j in CITIES: i <> j};
param cost {LINKS};

param budget > 0;
```

Figure 4–1. *AMPL data declarations for the budgeted traveling salesman model.*

```
var X {LINKS} logical;
var U {CITIES} >= 0, <= card {CITIES};

maximize Cities_Visited: sum {(i,j) in LINKS} X[i,j];

subject to Budget_Limit:
   sum {(i,j) in LINKS} cost[i,j] * X[i,j] <= budget;

subject to Leave_Home:
   sum {(Home,j) in LINKS} X[Home,j] = 1;

subject to Return_Home:
   sum {(i,Home) in LINKS} X[i,Home] = 1;

subject to Enter_equals_Leave {k in CITIES diff {Home}}:
   sum {(i,k) in LINKS} X[i,k] = sum {(k,j) in LINKS} X[k,j];

subject to Subtour_Elimination {(i,j) in LINKS: j <> Home}:
   U[i] - U[j] + card {CITIES} * X[i,j] <= card {CITIES} - 1;
```

Figure 4–2. *An integer linear programming formulation* of the budgeted traveling salesman problem, using standard AMPL declarations.

where $u_i, i \in \mathcal{C}$, are supplementary continuous variables.

Here, even more so than in the knapsack problem, we can make the formulation more concise and intuitive by introducing a decision set. In this case we seek a subset $\mathcal{T}$ of $\mathcal{C}$, representing the cities of the tour. Unlike in the knapsack case, however, $\mathcal{T}$ must be regarded as an ordered set, because the cost of the tour depends on the order in which the cities are visited. Indeed, it is convenient to regard $\mathcal{T}$ as being circularly ordered, so that for every $i \in \mathcal{T}$ there is a unique city $\text{next}(i) \in \mathcal{T}$. Then the optimization can be formulated as follows:

$$\begin{aligned} \underset{\mathcal{T} \subseteq \mathcal{C}}{\text{Maximize}} \quad & |\mathcal{T}| \\ \text{Subject to} \quad & \sum_{i \in \mathcal{T}} c_{i,\text{next}(i)} \leq B \\ & \text{first}(\mathcal{T}) = h \\ & (i, \text{next}(i)) \in \mathcal{L}, \quad \text{for all } i \in \mathcal{T} \end{aligned}$$

The first constraint imposes the budget as before, while the others say that $\mathcal{T}$ describes a tour out of the home city that uses only allowed links from $\mathcal{L}$. The circular ordering insures that, in the case of the last city l, we have $\text{next}(l) = \text{first}(\mathcal{T}) = h$, returning the tour to the home city as required

Both of our formulations of this problem are readily transcribed to AMPL. For either one, the data may be described as in Figure 4–1. In the case of the zero-one formulation, AMPL's standard features are also sufficient to describe the variables, objective and constraints, as shown in Figure 4–2. To represent the ordered set formulation, however, we must introduce a declaration of a circularly ordered subset that plays the role of the decision set $\mathcal{T}$:

```
var_set Tour circular within CITIES;

maximize Cities_Visited: card {Tour};

subject to Budget_Limit:
   sum {c in Tour} cost[c,next(c)] <= budget;

subject to Leave_Home: first(Tour) = Home;

subject to Link_Exists {c in Tour}: (c,next(c)) in LINKS;
```

Figure 4–3. *A decision-set formulation* of the budgeted traveling salesman problem, using the new `var_set` declaration to specify a circularly ordered subset.

```
var_set Tour circular within CITIES;
```

This is much the same as the `var_set` declaration used in our knapsack example, except for the addition of the keyword `circular` to specify that `Tour` is to be treated as a circularly ordered set in subsequent AMPL declarations and directives.

Using this `var_set`, the AMPL objective and constraints can be written in the much more concise form depicted by Figure 4–3. The simplicity of this alternative is appealing, but again it is not a useful formulation unless we can append some directives to say how the collection of possible tours is to be implicitly enumerated. Because different orderings of the cities represent distinct tours, having generally different costs, we must work with a somewhat larger search tree than before.

As in Section 2, we first describe the principles of the search, and then survey the directives that can be used to specify an implicit enumeration.

Search principles

We want to start at the home node, and build up a tour by adding successive cities. Thus at the root node of our search tree, the build set contains the home city, while the free set contains all other cities; that is, the build set is $\{h\}$, and the free set is $\mathcal{C}\backslash\{h\}$.

To create the first child of the root, we pick some city a from the free set. The build set of the resulting node is $\{h, a\}$, and represents the tour from home to a and back. City a is also removed from the root's free set, which becomes $\mathcal{C} \setminus \{h, a\}$. The child's free set consists of all cities not in its build set, so it is also $\mathcal{C} \setminus \{h, a\}$.

A second child of the root may next be created in an analogous way, by picking another city b from the root's free set. Then the build set of the resulting node is $\{h, b\}$; city b is removed from the root's free set, which becomes $\mathcal{C}\backslash\{h, a, b\}$. The child's free set, consisting of all cities not in its build set, is $\mathcal{C}\backslash\{h, b\}$.

We can also create a new node by choosing the root's first child to be a parent, and selecting b from its free set. The build set of the resulting child node is $\{h, a, b\}$, representing the tour from home to a to b and back. City b is removed from the parent's free set, which becomes $\mathcal{C}\backslash\{h, a, b\}$. The child's free set consists of all cities not in its build set, so it is also $\mathcal{C}\backslash\{h, a, b\}$.

In general, a branching operation consists of choosing a parent node, and choosing a city from the current free set at that node. A child is created, whose build set is the parent's with the chosen city added at the end, and whose free set contains all cities not in its build set. The chosen city is also removed from the parent's free set. The final search tree is essentially the same regardless of the order in which parents and children are chosen; an example is shown in Figure 4–4.

This *ordered subset* search tree is significantly different from the *unordered subset* tree that we used for the knapsack problem (Figure 2–3). Because differently ordered subsets are distinguished, this tree is symmetric, and its number of nodes grows much more quickly with the number of members of the initial free set. Also, in the particular example of the

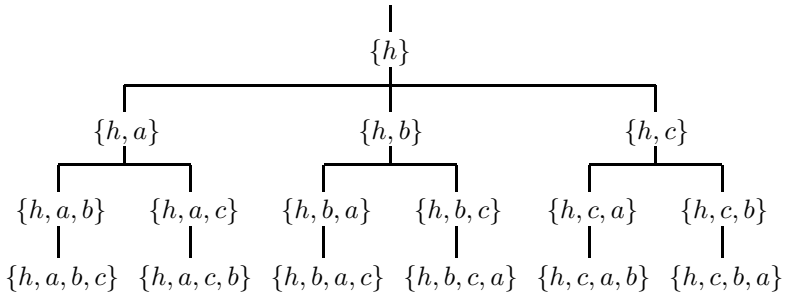


Figure 4–4. *Search tree for orderings of* $\{h, a, b, c\}$, starting from h.

budgeted traveling salesman, the root node is associated with $\{h\}$ rather than $\{\}$, because there is no meaning to a tour that does not include the home city.

As in previous examples, our first directive must give names to the build and free sets:

```
enumerate Tour: build_set SubTour := {Home},
                free_set  Free;
```

In this case we explicitly assign `{Home}` as the initial build set, on the assumption that the enumeration procedure will not know to do this automatically.

Fathoming

A node and all its descendants can be dropped if there is obviously no way within the budget to get from the last city in the build set back to the home city.

To develop a directive for this purpose, we first consider the cost of traveling from the home city to the last city in the build set, through the other cities in the build set in the given order. This cost, which we denote `path_cost`, is easily updated from node to node by use of the updating directives that we previously introduced:

```
param at first node: path_cost := 0;

param at child node:
    path_cost := path_cost +
         cost[prev(last(SubTour)),last(SubTour)];
```

Because `SubTour` is an ordered set, we can use AMPL's built-in functions `prev` and `last` to write `prev(last(SubTour))` for the city just previous to the last city.

There are two ways in which we might get from the end of the build set back to `Home`:

- Travel directly from `last(SubTour)` to `Home`, if a link between them exists.
- Travel from `last(SubTour)` to a city in the free set, and later from a city in the free set to `Home`.

We want to say that the node is fathomed if the cost of either of these possibilities, when added to the above-defined `path_cost`, must cause the budget to be exceeded. For the first, clearly the cost of travel is just `cost[last(SubTour),Home]`, if a link from `last(SubTour)` to `Home` exists. For the second, we can say that the travel back to `Home` will equal at least the sum of the following two costs:

- The cheapest cost of traveling from the last city in the build set to some city in the free set.
- The cheapest cost of traveling to the home city from some city in the free set.

The following directives define these as `out_cost` and `in_cost`:

```
param at new node:
    out_cost := min {j in Free:
        (last(SubTour),j) in Links} cost[last(SubTour),j];

param at new node:
    in_cost := min {i in Free: (i,Home) in Links} cost[i,Home];
```

Using these, we can write out the fathoming directive:

```
fathom at child node:
  (last(SubTour),Home) notin Links or
      path_cost + cost[last(SubTour),Home] > budget) and
  (path_cost + in_cost + out_cost > budget);
```

This directive may be ineffective at early stages of the search, where the build sets are still small and the `path_cost` values low. Thus we next look at other ways to keep the tree to manageable size.

Pruning

When a new node is created, many of the members of its free set correspond to obviously impossible branches. We want to specify that these members be removed from the free set immediately — or equivalently, that the branches be pruned immediately — so that search time is not wasted.

Most obviously, there is no sense branching to a city that is not connected by a link from the last city of the build set. To communicate this information to the enumeration procedure, we require a new kind of directive:

```
prune at child node:
    {c in Free: (last(SubTour),c) notin Links};
```

Here we have specified a set of cities, all of which are to be removed from the free set immediately after any node is created.

We can go further, to remove cities whose addition to the tour would be obviously too expensive. Suppose that we are considering the addition of city `c` to the tour represented by a certain build set. The cost of the resulting tour has three components:

- The cost of traveling from the home city to the last city in the build set.
- The cost of traveling from the last city in the build set to city `c`.
- The cost of traveling from city `c` back to the home city.

The first of these is what we have already denoted `path_cost`, and the second is just `cost[last(SubTour),c]`. It is tempting to write the third cost similarly as `cost[c,Home]`. However, we cannot be sure that the pair `(c,Home)` is in the set of allowed links; even if it is, there may be a shorter indirect route (unless the costs satisfy a triangle inequality). Instead we simply add a lower bound, equal to the cheapest path from `c` to either the home city, or another city in the free set. In AMPL notation, this bound can be written as `min {(c,j) in Links: j = Home or j in Free} cost[c,j]`.

Motivated by this analysis, we can write the following pruning directive to be applied after our first one:

```
prune at child node:
    {c in Free: path_cost + cost[last(SubTour),c] +
        min {(c,j) in Links: j = Home or j in Free} cost[c,j]
             > budget};
```

This says that we prune cities when a lower bound on the cost of adding them is already greater than the remaining budget. This lower bound can be improved in various straightforward ways, particularly by more closely considering the case in which `(c,Home)` is not an allowed link.

Bounding

We can try to further generalize the above strategy by calculating an upper bound on the number of cities that can be visited within the budget, given that we start with the sequence specified by the build set.

First, we note that if two cities from the free set are added to the build set, the cost of the resulting tour is at least `path_cost + out_cost + in_cost`. We then observe that, to insert k more cities from the free set, we must use $k+1$ more links that are entirely between cities in the free set. Thus $k+1$ times the cheapest link between two free-set cities cannot exceed `budget` minus `path_cost + out_cost + in_cost`. Solving for k, the maximum number of additional cities capable of insertion is defined by:

```
param at child node:
     max_insert := floor (
          (budget less (path_cost + out_cost + in_cost))
             / (min {(i,j) in Free} cost[i,j]) - 1 )
```

We take the `floor` of (greatest integer less than or equal to) the computed number, since we can only insert an integral number of cities.

The desired upper bound can now be expressed as the number of cities in the build set, plus at most two additional cities connected directly by links to the build set, plus the quantity `max_insert` defined above:

```
bound at child node: count {SubTour} + 2 + max_insert;
```

An analysis of this formula shows that it gives a correct bound even when `max_insert` comes out to be 0 or -1.

Branch selection

One appealing implicit enumeration strategy is to always extend the search tree by the cheapest branch. In other words we select a city, within the free set of some node, that can be reached from the end of the node's build set at lowest cost. We can hope that, by proceeding in this way, the search will quickly find good tours through large subsets of cities.

Such a selection strategy actually involves two choices: selection of a node that admits a cheapest branch, and selection of a city that corresponds to such a branch. The situation may be seen more clearly by examining the requisite directives:

```
select parent node: lowest min {c in Free} cost[last(SubTour),c];

select child node: arg min {c in Free} cost[last(SubTour),c];
```

The first directive selects the parent node at which the new child will be created; it is the node with the "lowest" minimum-cost branch available. Then the second directive picks a child to be created by branching from the selected parent.

5. A Loading Model

We consider finally a model motivated by the loading of bulk materials onto vehicles. The set from which a subset must be chosen is a set of pairs. The constraints exhibit complications that are typical of combinatorial optimization problems.

Formulation

Our vehicles have a fixed number of compartments, of various volumes. Compartments must be loaded in a given order — not necessarily their physical order — so as to maintain balance and other desirable properties. A particular compartment can hold only certain of the products that are to be shipped, and it must be filled completely with one product.

The relevant data for this problem can be described by the AMPL `set` and `param` declarations in Figure 5–1. For each compartment `c` in `1..number_compart`, there is a volume `vol[c]`. For each product `p` in set `PROD`, there are both a minimum required shipment `min_ship[p]` and a maximum acceptable shipment `max_ship[p]`; the difference between them provides us with some leeway to to find a feasible solution under the requirement that all compartments be filled completely and with one product.

One further piece of data, the set of pairs `OK` declared in Figure 5–1, indicates which compartments may hold which products. A compartment `c` is allowed to contain a product `p` if and only if `(c,p)` is a member of `OK`.

We can view each feasible loading order as corresponding to a subset of the `(c,p)` pairs from `OK`. Since the choice of this subset is the decision to be made, it can be declared as a `var_set` in our previous terminology:

```
var_set Loading within OK;
```

Figure 5–2 shows the model that results. The objective `sum {(c,p) in Loading} vol[c]` is simply to ship as much as possible. The constraints `Demand_Met` ensure that the amount of each product shipped is between the specified limits.

The remaining constraints enforce our operational requirements. For a compartment `c`, the expression `card {(c,p) in Loading}` equals the number of products loaded into `c`. The constraint `Fill_First_Compartment` says that one product is loaded into the first compartment, while the constraints `Load_in_Order` ensure that a contiguous sequence of compartments is filled with one product each, while all other compartments are left unfilled.

```
param number_compart integer > 0;

set PROD 'products';

param vol {1..number_compart} 'volume' > 0;

param min_ship {PROD} 'required' > 0;
param max_ship {PROD} 'acceptable' > 0;

set OK within {1..number_compart,PROD} 'permitted combinations';
```

Figure 5–1. *AMPL data declarations for the bulk material loading model.*

```
var_set Loading within OK;

maximize Total_Shipped: sum {(c,p) in Loading} vol[c];

subject to Demand_Met {p in PROD}:
     min_ship[p] <= sum {(c,p) in Loading} vol[c] <= max_ship[p];

subject to Fill_First_Compartment: card {(1,p) in Loading} = 1;

subject to Load_in_Order {c in 2..number_compart}:
     card {(c,p) in Loading} <= card {(c-1,p) in Loading};
```

Figure 5–2. *AMPL declarations for the objective and constraints of the bulk material loading model,* employing `var_set` to specify the selection of an optimal subset from a set of ordered pairs.

Search principles

If we treat `Loading` as merely a collection of objects, we can use the same kind of search tree as in the knapsack example. Most of the subsets enumerated by such a tree are in violation of the constraints, however, because they do not fill consecutive compartments with one product each. To avoid enumerating all these infeasible subsets, we prefer to restrict the search so that it considers filling the compartments in order. The search tree then looks like the example in Figure 5–3.

We can specify such a tree by simply restricting the free set at each node. A `free` directive could be introduced into AMPL for this purpose:

```
enumerate Loading: build_set Filled,
                   free_set  Unfilled;

param at root node: nextC := 1;
free at root node: {(nextC,p) in OK};

update at child node: nextC := nextC + 1;
free at child node: {(nextC,p) in OK};
```

The parameter `nextC` simply keeps track of the next compartment to be filled; it is equal to the depth of the node in the tree. The free set at a node is thus initially the set of all pairs `(nextC,p)` where `p` is a product that can be accommodated in the next compartment. This option of specifying the free set offers a great deal of flexibility in designing the search tree to take advantage of particular constraints. Indeed, its main weakness is that it may offer too much flexibility. It does nothing to prevent the modeler from accidentally specifying an incomplete tree, which may fail to include any of the optimal feasible subsets.

As an alternative, rather than expand the variety of search directives, we might enhance the `var_set` declaration so that it could directly describe our problem as one of choosing a sequence of products to load. Just as we declared the decision set `Tour` in the budgeted traveling salesman example to be `ordered`, we could declare `Loading` in the current example to be a `sequence`:

```
var_set Loading sequence within PROD;
```

The structure of the search tree in Figure 5–3 would follow directly from this declaration. As a bonus, we would be able to dispense with the constraints `Fill_First_Compartment` and `Load_in_Order`, which are arguably as unnatural as many of the constraints found in integer programming formulations. In their place we could declare simply:

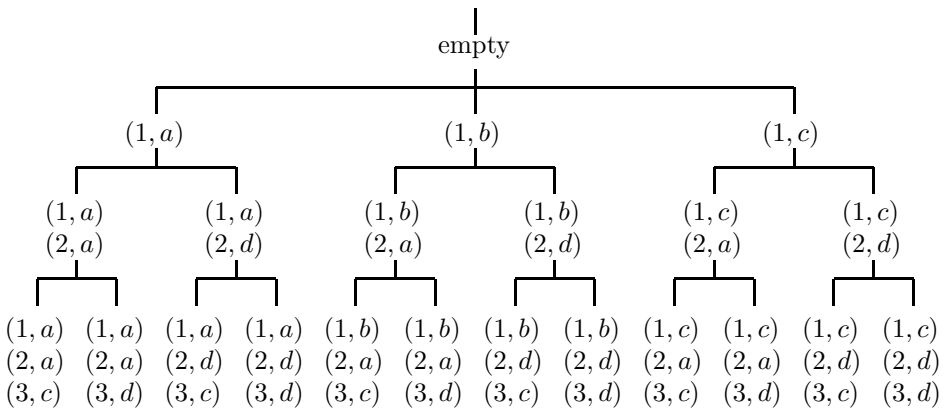


Figure 5–3. *Search tree for an instance of the loading problem,* where compartments 1, 2, and 3 can accommodate products from the sets $\{a, b, c\}$, $\{a, d\}$, and $\{c, d\}$, respectively.

```
subject to Loading_Allowed {p in Loading}: (ord(p),p) in OK;
```

The disadvantage to this approach is that `Loading` is no longer a set in the usual sense. It could be, say, $\langle a, b, a, a, c, b, d, b\rangle$, with some members appearing more than once because some products are loaded into more than one compartment. Sequences of this sort behave differently from sets when subjected to many of the usual indexing and set operations. Significant extensions to the syntax and semantics of the modeling language would be necessary to accommodate them.

Fathoming

We can define a parameter `loaded[p]` at each node, to represent the amount of product `p` that has already been loaded into compartments $1, \ldots,$ `nextC` $- 1$:

```
param at child node {p in PROD}:
    loaded[p] := sum {(c,p) in Filled} vol[c];
```

Then a node can be fathomed if neither it nor its descendants can possibly satisfy the demand constraints:

```
fathom at child node {p in PROD}: loaded[p] > max_ship[p];

fathom at child node:
    sum {p in PROD} (min_ship[p] less loaded[p])
        > sum {c in nextC .. number_of_compartments} vol[c];
```

The first directive detects any product of which too much has already been loaded. The second is for the case in which the volume of all remaining compartments is insufficient to permit the total minimum shipments of all products.

We can construct a stronger condition by observing that, because all filled compartments must be filled completely, there might be no combination of remaining compartments that will permit the total shipment of product `p` to lie between `min_ship[p]` and `max_ship[p]`. If this is true of even one product, then none of the descendant nodes is feasible, and the node under consideration can be fathomed.

To specify this condition we imagine that we are given, for each compartment `c`, a set `FEAS[c]` that has the following property: a number `s` is in `FEAS[c]` if and only if a shipment of volume `s` can be made using some combination of compartments `c` and later. We then specify the fathoming condition as follows:

```
set FEAS {1..number_compart} 'possible shipment sizes';

set at child node: Can_Ship {p in PROD} :=
    {s in FEAS[nextC]:
       min_ship[p]-loaded[p] <= s <= max_ship[p]-loaded[p]};

fathom at child node: exists {p in PROD} card {Can_Ship[p]} = 0;
```

For each product `p`, the set `Can_Ship[p]` contains the different total amounts that could be shipped in compartments `nextC` and later so as to achieve a feasible solution, given that an amount `loaded[p]` has already been committed in compartments prior to `nextC`. Thus a new child node can be fathomed if there exists any `p` for which this set is empty.

It is undesirable to have to supply each set `FEAS[c]` as independent data, since its membership is fully determined by the compartment sizes `vol[c]` up to `vol[number_compart]`. In this case, a short script of AMPL commands could build the needed sets:

```
let FEAS[number_compart] := {0, vol[number_compart]};

for {c in number_compart-1 .. 1 by -1}
   let FEAS[c] := FEAS[c+1] union setof {f in FEAS[c+1]} (f+vol[c]);
```

Our fathoming criterion could be refined further, at the cost of some additional complication, to take account of the fact that only certain products can go into certain compartments. The sets `FEAS[c,p]` needed for this purpose could still be computed in AMPL. For yet more complicated cases, problem-specific data manipulation routines could be written in a language such as C, then linked to a function in the modeling language; AMPL and other languages provide for user-defined functions of this kind.

Bounding

The sum of `loaded[p]` and the largest member of `Can_Ship[p]` gives the most of product `p` that can be feasibly loaded. Thus an upper bound on the objective for a new node and all its descendants is given by

```
bound at child node:
    sum {p in PROD} (loaded[p] + max {s in Can_Ship[p]} s);
```

The maximum is guaranteed to be over a nonempty set, since otherwise the node would already have been fathomed as explained above. If `p` has already been loaded as much as possible, then the corresponding set `Can_Ship[p]` will be `{0}`.

Pruning

The sets `Can_Ship[p]` are computed independently of the free set. Thus we could compute these sets first, and then use them to further restrict the free set:

```
free at child node:
    {(nextC,p) in OK: max {s in Can_Ship[p]} s > 0};
```

Alternatively, if it is clearer or more convenient, we could use the simpler `free` directive introduced previously, together with a pruning directive:

```
free at child node: {(nextC,p) in OK};

prune at child node:
    {(nextC,p) in Unfilled: max {s in Can_Ship[p]} s = 0};
```

Recall that `Unfilled` is the name given to the free set. We prune branches to products for which `Can_Ship[p]` has only the member zero, since there is no feasible way to ship any more of these products.

We can alternatively refine the free set by dropping products that would exceed their maximum if loaded into the next compartment:

```
prune at child node:
     {(nextC,p) in Unfilled: loaded[p] + vol[nextC] > max_ship[p]};
```

All nodes pruned by the preceding directive are in fact pruned by this one. This directive may prune additional nodes, however, representing products that cannot be loaded into compartment `nextC` but that remain eligible to be loaded into later, smaller compartments.

Branch selection

Criteria for selecting the next parent node for branching may reasonably be based on the amount loaded so far. For instance, we could specify either

```
select parent node: smallest sum {p in PROD} loaded[p];
```

or

```
select parent node: largest sum {p in PROD} loaded[p];
```

The former gives a kind of breadth-first search, and the latter a depth-first search.

In selecting branches to child nodes, we might want to first consider those that load products that are highly constrained in the total that can be shipped. Then the selection directive would be

```
select child node:
     arg min {(c,p) in Unfilled} (max_ship[p] - min_ship[p]);
```

Another possibility is to favor products that still have a large shipment quantity unassigned:

```
select child node:
     arg max {(c,p) in Unfilled} (min_ship[p] - loaded[p]);
```

Or, precedence might be given to products that have fewest shipment-size options remaining:

```
select child node:
     arg min {(c,p) in Unfilled} card {Can_Ship[p]};
```

Several of these could be used in sequence, each applying to the potential branches that are tied under the previous criteria. Here the modeling language shows its power in allowing many criteria to be formulated quickly and simply for testing.

6. Concluding Remarks

There is no single best way of representing combinatorial optimization problems to computer systems. We begin this section by contrasting several approaches currently under investigation or development. We first address enhancements to algebraic modeling languages that may be achieved through translation to integer programs and through generalization of branch-and-bound procedures, and then consider alternative logic-based and network-based approaches.

To conclude, we summarize the advantages and disadvantages of the decision-set approach proposed in this paper, and indicate some directions for extension of this work.

Translation to integer programs

As we have remarked at several points in this paper, certain classes of combinatorial optimization problems have long been addressed by formulating them as integer linear programs. Algebraic modeling languages are ideal for representing these formulations, as illustrated in Figures 2–1 and 4–2.

One logical next step would be to automate the integer programming formulation process. The modeling language would first be extended to better represent combinatorial problems, perhaps by adding constructs such as the `var_set` that we have proposed, and perhaps by extending the variety of expressions allowed in objectives and constraints. As an example of the latter, AMPL's existing `or`, `card` (set cardinality) and `if ... then ... else` operators can be used to write meaningful expressions for fixed costs,

```
minimize Total_Cost:
   sum {i in ORIG, j in DEST}
      (if sum {p in PROD} Trans[i,j,p] > 0 then fcost[i,j]) + ... ;
```

zero-or-minimum constraints,

```
subject to Truckload {i in ORIG, j in DEST}:
   sum {p in PROD} Trans[i,j,p] = 0 or
   sum {p in PROD} Trans[i,j,p] >= minload;
```

and uniqueness restrictions:

```
subject to Unique_Supplier {j in DEST, p in PROD}:
   card {i in ORIG: Trans[i,j,p] > 0} <= 1;
```

The language translator would then be extended to convert problems using these kinds of expressions into problems in terms of integer decision variables. In effect, the translator would apply a variety of integer programming formulation "tricks" (as discussed for example in Chapter 24 of [2]) without the modeler having to know about them. Following the translation, any general-purpose branch-and-bound code could be used as a solver. (Some provision would also have to be made for translating the results back into the terms of the original model.)

This approach has been investigated by Greenberg [17, 18], Hürlimann [24], and Mitra *et al.* [37], and continues to be of considerable interest. It is not yet available within many of the most popular modeling languages, however, and appears to face at least two significant hurdles to wider adoption.

First, when any new type of combinatorial expression is allowed in objectives and constraints, the variety of new problems that can be expressed is likely to exceed the variety of problems that can be translated to integer programs. Some of the new problems will contain expressions too complex for the translator to sort out, while others will be found to admit no concise integer programming formulation at all. As a result, certain restrictions will have to be placed on modeling language extensions for combinatorial optimization. Greenberg's MODLER system, for example, permits logical logical constraints of the forms

`IF` *binary condition* `THEN` *binary condition*
`IF` *binary condition* `THEN` *activity bound condition*

where the *binary condition* and *activity bound condition* are carefully restricted to ensure that a translation to algebraic constraints is always possible. Restrictions of these kinds have the advantage of ensuring that anything properly written in the language can be successfully translated, but also have the disadvantage of tending to appear arbitrary from the modeler's point of view. Especially for modelers unfamiliar with the the mathematics of the translation, these restrictions can make the language seem more complicated and less suitable as a general-purpose tool.

Second, any difficult combinatorial optimization problem has a range of different equivalent integer programming formulations, some of which are much better than others from the standpoint of solution time. The ability of a branch-and-bound code to return an optimal (or even near-optimal) solution may depend critically on the ability of the modeler to provide a "tight" formulation that gives good bounds at the nodes of the search tree. The ability of a modeling language translator to similarly provide a tight formulation is unproven. Certain principles can be seen clearly in individual cases, but their generalization for purposes of automatic translation remains a challenge.

Generalization of branch-and-bound procedures

As an alternative to translating combinatorial optimization problems into a form that standard branch-and-bound solvers require, one can consider generalizing the solvers to handle a broader variety of discrete constraints. The solvers can then directly support a more natural and general variety of expressions in algebraic modeling languages.

A few kinds of combinatorial restrictions on groups of variables, known as *special ordered sets* [1], have a long history of being accommodated by large-scale branch-and-bound codes [45]. Special ordered sets "of type 2" are employed to represent separable piecewise-linear functions, and hence can be supported by AMPL's piecewise-linear function notation [11]. Special ordered sets "of type 1" specify that at most one variable in a group may be nonzero. This restriction is harder to express in current algebraic languages, whose syntax does not naturally support statements about "at least one" or "exactly one" variable from a specified group. The AIMMS language [2] circumvents this difficulty by allowing a constraint in

nonnegative variables to be designated a `SOS1 CONSTRAINT`, in which case at most one variable appearing in that constraint is allowed to be positive.

The usefulness of special ordered sets suggests that they should be supported in algebraic modeling languages by a constraint syntax that expresses their intent directly. For example, the constraints `Unique_Supplier[j,p]` above, employing the `card` function, represent special ordered sets of type 1. They could be made clearer, both to the modeler and to the solver, by introducing a new construct analogous to the `exists` and `forall` operators currently in AMPL:

```
subject to Unique_Supplier {j in DEST, p in PROD}:
   atmost1 {i in ORIG} (Trans[i,j,p] > 0);
```

A language translator might automatically identify this constraint as a special ordered set, requiring no special designation by the user, and involving no translation into algebraic constraints in terms of integer variables.

Such an approach need not stop at special ordered sets. Branch-and-bound procedures can be extended to directly handle of variety of other logical conditions in objectives and constraints. A generalization to zero-or-range constraints such as `Truckload[i,j]` above is described by Hansen and Hugé [19]. More broadly, many search procedures developed for constraint satisfaction and logic programming are known to have powerful analogues in the context of branch-and-bound, as elucidated by recent work of Hooker and others [20, 22, 21]. These generalizations can help to speed the branch-and-bound process, by enabling it to deal directly with a problem's essential combinatorial restrictions, rather than limiting it to constraints that the user is able to construct in terms of integer variables.

As long as modeling languages (or other optimization problem formats) tend to force the modeler to work in terms of integer variables, however, there is limited motivation to generalize branch-and-bound solvers to handle combinatorial constraints. At the same time, since few such generalizations are implemented in solvers, there is limited motivation to extend the constraint syntax of modeling languages. Newer solvers, constructed modularly to invite generalizations, may help to break this deadlock. MINTO [38] is designed specifically to encourage users to write their own branching and bounding routines; to a lesser degree, similar "user exit" or "callback" features can be found in other widely used packages such as OSL [23] and CPLEX [7].

A generalized branch-and-bound approach has the great advantage of bypassing difficulties associated with automatic translation to an integer program. There remains the other kind of difficulty discussed previously, that in making the corresponding generalizations to a modeling language, we can easily cause the variety of language expressions allowed to exceed the variety of expressions that a solver can handle. Using the `atmost1` operator introduced above, for example, the modeler would expect to be able to formulate a related constraint,

```
subject to Unique_Supplier_All {j in DEST}:
   atmost1 {i in ORIG} (sum {p in PROD} Trans[i,j,p] > 0);
```

which however does not describe a special ordered set in the usual sense. In such situations, we can at least anticipate that the job of recognizing supported expressions may be pushed down from the language translation level to the solver level. As an example, independently of any solver, the AMPL translator could process the above constraint into its standard parsed output format. The driver linking any solver to AMPL would subsequently examine the parsed constraint to determine whether it was of a form that could be accommodated in the solver's branch-and-bound routines — much as current drivers for quadratic programming solvers check the parsed nonlinear objective to determine whether it represents a quadratic function. The modeling language itself would not have to be arbitrarily restricted in any way, and so would remain ready to support any more general forms that a solver might later be extended to handle.

Logic-based languages

Rather than continue to stretch algebraic modeling languages to better accommodate combinatorial optimization, one may consider alternative modeling languages that are designed specifically for the purpose of describing combinatorial or logical constraints. Indeed many such languages have been proposed, studied and implemented. Lauriere's ALICE [33], the most notable early work in this area, describes an optimization problem in terms of finding a best function of a certain kind, in a way that offers several parallels to our concept of a decision set; constraints are described through a variety of algebraic and logical forms. Work in this area has proceeded mainly within the field of artificial intelligence, by contrast to work on algebraic languages which his appeared mainly in the operations research literature.

Logic programming in particular has been extensively studied as an alternative to traditional mathematical programming systems for specifying and solving decision problems. In fact a logic programming language such as Prolog [44] is analogous, in several important respects, to an algebraic modeling language such as AMPL that has been extended to encompass decision sets. Both are declarative languages that can describe many combinatorial problems. Any problem expressed in either language can be solved, at least in principle, by a form of enumeration. When either is applied to problems of practical interest, however, no simple or explicit enumeration scheme can be relied upon to consistently find an answer in an acceptable amount of time.

In the context of the CHIP project, Van Hentenryck [46] has designed extensions to Prolog that are similar in general intent to our proposed AMPL enumeration directives. His extensions allow powerful search strategies to be introduced, and greatly simplify the description of search rules that are specific to individual problem classes. Additional extensions facilitate the description of constraints and objectives natural to combinatorial optimization problems.

An implementation of CHIP has been successfully applied to a variety of hard combinatorial optimization problems [8, 47]. Logic programming for combinatorial optimization continues to be an active area of research, with notable related work including Colmerauer's Prolog III [6], and McAloon and Tretkoff's 2LP [35].

For combinatorial optimization problems that are directly concerned with numerical decision variables, algebraic modeling languages offer a natural form of expression based on familiar mathematical notation, while logic programming languages often have the disadvantage of requiring a substantial re-thinking and translation. For other kinds of combinatorial optimization problems, however, the conversion to decision variables can be awkward, with the result that the logic programming approach has advantages both in naturalness of expression and in speed of solution. The ideas presented in this paper can be regarded as a way of generalizing algebraic modeling languages so that their advantages extend to a broader range of combinatorial optimization problems.

Network-based optimization

Network-based systems are motivated by the great variety of combinatorial optimization problems defined (like our budgeted traveling salesman example) in terms of some collection of nodes (cities) and arcs (links). These systems dispense with any constraint expression language, employing instead a collection of standard symbols — such as the *netforms* of Glover, Klingman and Philips [15, 16] — for describing network structure and attributes. Recent implementations such as Steiger, Sharda and Leclaire's GIN [42, 43], Ogryczak, Studziński and Zorychta's DINAS/EDINET [39, 40], Chesapeake Decision Sciences' MIMI/G [5], McBride's NETSYS [36], Jones's NETWORKS [28, 25, 26, 27], and Kendrick's PTS [30, 31] can thus offer intuitive graphical interfaces for the solution of combinatorial optimization problems on networks.

Typically, the graphical interface is employed to set up a display of the nodes and arcs, and to enter their fundamental attributes such as demands and requirements. The

modeler then selects from a menu of minimum-cost flow, optimal design, routing, and other combinatorial problems for which the system has built-in algorithms. The network display is automatically updated to represent the optimal solution, and in some cases also the progress of the algorithm.

These systems completely avoid the difficulties of describing combinatorial optimization problems in algebraic terms. In many respects their designs are analogous to those of successful statistical systems, with network diagrams being the counterpart of data series.

The main drawback of these systems is their limitation to whatever problem types have been built in. Current implementations have tended to concentrate on minimum-cost flow, shortest path, and maximum flow problems. These are relatively easy to describe in any algebraic modeling language, however, and are especially easy to describe in languages (like AMPL and AIMMS) that offer the option of defining variables and constraints through node and arc declarations [2, 11]. Possibly the best features of algebraic and network-based representations could be combined into a single design, as proposed for example by Jones and D'Souza [29].

Decision sets in algebraic modeling languages

Our examples from previous sections suggest that algebraic modeling languages have several strengths for the specification of combinatorial optimization problems. Most notably,

- A modeling language's facilities for describing sets, parameters and indexing are valuable in discrete optimization just as in continuous optimization.
- The set notation of a modeling language is well suited to describing a variety of discrete decisions that involve choosing some kind of subset.
- The algebraic expressions of a modeling language are well suited to describing sets and values that control bounding, fathoming and branching in an implicit enumeration scheme.

We have proposed to take advantage of these strengths by introducing the concept of *decision sets* into algebraic modeling languages. Our illustrations have shown — through introduction of the `var_set` construct — how decision sets can enable an algebraic language to express a broader range of combinatorial optimization problems, and to express such problems more naturally.

To the extent that this approach involves an extension of a modeling language's expressions, it has something in common with the algebraic language approaches described earlier in this section. Yet it also differs substantially, through its introduction of supplementary directives to guide an implicit enumeration of the decision set. By providing these directives, it relies less than other approaches on the ability of model translators or solvers to recognize and exploit combinatorial structures automatically. At the same time, it relies more on the modeler's ability to describe bounds, branching priorities and other rules relevant to a particular problem.

We have argued throughout this paper that the combination of decision sets with enumeration directives has the potential to successfully address a particularly wide variety of combinatorial optimization problems. This approach is also subject to some drawbacks that may limit its applicability, however, as our examples have also shown.

At the least, a significant degree of analysis will be needed to determine how a given discrete optimization problem can be accommodated using our ideas. The initial work of constructing a model is not at issue, since a formulation in terms of a decision set is often shorter and closer to the modeler's original idea than one in terms of decision variables. Further study will usually be necessary, however, to determine what enumeration directives are appropriate, and how they can be written in the modeling language. The work of constructing and testing directives will require considerably more talent on the part of the modeler than, say, the formulation of typical linear constraints.

There are also limits to the variety of implicit enumeration criteria that can be expressed through algebraic directives. Tighter bounds and stronger fathoming rules, in particular, are likely to involve subsidiary computations that cannot be characterized by purely declarative statements such as we have used in our examples for this paper. These computations will require the use of procedural statements, such as `repeat`, `if` and `break` in the AMPL command language [12] or their counterparts in the AIMMS [2] and GAMS [4] languages. Because these are interpreted languages that have been designed and implemented with an emphasis on generality and flexibility of the set and parameter expressions — rather than on speed of execution — computations are likely to be much slower than for identical bounds and rules implemented in C or another compiled language. AMPL's user-defined functions may help to circumvent this problem, by allowing statements in AMPL to call the user's compiled procedures, but at the cost of requiring the user to work in C as well as in the modeling language. Alternatively, following the lead of many database management systems, mathematical programming systems may introduce optional compilers for their languages.

In light of these observations, the approach proposed in this paper can be seen as offering both the advantages and disadvantages of using general-purpose rather than custom-designed optimization tools. A language custom-designed for budgeted traveling salesman or for bulk material loading problems would be easier to work with, but would require a greater initial investment relative to the range of problems covered. Similarly, an implicit enumeration algorithm custom-developed for one of these problems would be more efficient, but would require that a considerable expenditure in programming, debugging and maintenance be devoted to a restricted problem class.

An algebraic modeling language is thus likely to be most attractive for combinatorial optimization where a large investment is not immediately desirable. Examples include the solution of discrete problems whose size and complexity are modest, and the prototyping of branch-and-bound schemes for applications that cannot yet justify a commitment to more specialized software. The teaching of computational discrete optimization is another attractive prospective use.

We conclude by observing that our analysis has concentrated on the case of a single decision set (or `var_set`), and on certain ways of enumerating subsets. A variety of issues thus remain for study, including:

- Enumeration of more than one decision set in the same model.
- Formulation in terms of a decision set together with numerical-valued variables, either continuous or discrete.
- Specification of continuous linear programs or other subproblems to be solved for bounding information at each node.
- Extension of the decision set concept to sequences that may include several copies of a member.

Resolution of these matters will require further study of the principles underlying implicit enumeration, as well as further extensions to the proposed modeling language directives.

Acknowledgements

This work has been supported by the Koninklijke/Shell-Laboratorium, Amsterdam, The Netherlands. Additional support has been provided through National Science Foundation grants DDM-8908818 and DMI-9414487.

References

[1] E.M.L. Beale and J.A. Tomlin, 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 (1970) 447–454.

[2] J.J. Bisschop and R. Entriken, *AIMMS: The Modeling System*. Paragon Decision Technology, Haarlem, The Netherlands (1993).

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

[4] A. Brooke, D. Kendrick and A. Meeraus, *GAMS: A User's Guide, Release 2.25*. Boyd & Fraser/The Scientific Press, Danvers, MA (1992).

[5] Chesapeake Decision Sciences, *MIMI/LP User's Manual*. New Providence, NJ (1992).

[6] A. Colmerauer, An Introduction to Prolog III. *Communications of the ACM* **33** (1990) 69–90.

[7] CPLEX Optimization, Inc., *Using the CPLEX Callable Library*, version 3.0. Incline Village, NV (1994).

[8] M. Dincbas, H. Simonis and P. Van Hentenryck, Solving Large Combinatorial Problems in Logic Programming. *Journal of Logic Programming* **8** (1990) 75–93.

[9] S.P. Dirkse, Robust Solution of Mixed Complementarity Problems. Mathematical Programming Technical Report 94-12, University of Wisconsin, Madison (1994).

[10] S.P. Dirkse, M.C. Ferris, P.V. Preckel and T. Rutherford, The GAMS Callable Program Library for Variational and Complementarity Solvers. Mathematical Programming Technical Report 94-07, University of Wisconsin, Madison (1994).

[11] R. Fourer and D.M. Gay, Expressing Special Structures in an Algebraic Modeling Language for Mathematical Programming. Technical Report 91-01, Department of Industrial Engineering and Management Sciences, Northwestern University, Evanston, IL (1991); forthcoming in *ORSA Journal on Computing*.

[12] R. Fourer and D.M. Gay, Integrating Programming Features with an Algebraic Modeling Language for Optimization. RD17.3, 15th International Symposium on Mathematical Programming, Ann Arbor (1994); also MC27.1, ORSA/TIMS Joint National Meeting, Detroit (1994).

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

[14] R. Fourer, D.M. Gay and B.W. Kernighan, *AMPL: A Modeling Language for Mathematical Programming*. Boyd & Fraser/The Scientific Press, Danvers, MA (1992).

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

[16] F. Glover, D. Klingman and N.V. Phillips, *Network Models in Optimization and Their Applications in Practice*. John Wiley & Sons, New York (1992).

[17] H.J. Greenberg, MODLER: Modeling by Object-Driven Linear Elemental Relations. *Annals of Operations Research* **38** (1992) 239–280.

[18] H.J. Greenberg, *Modeling by Object-Driven Linear Elemental Relations: A User's Guide for MODLER.* Kluwer Academic Publishers, Boston, MA (1993).

[19] P. Hansen and J. Hugé, Implicit Treatment of "Zero or Range" Constraints in a Model for Minimum Cost Foundry Alloys. *Management Science* **35** (1989) 367–371.

[20] F. Harche, J.N. Hooker and G.L. Thompson, A Computational Study of Satisfiability Algorithms for Propositional Logic. *ORSA Journal on Computing* **6** (1994) 423–435.

[21] J.N. Hooker, Logic-Based Methods for Optimization. *Operations Research Society of America Computer Science Technical Section Newsletter* **15:**2 (1994) 4–11.

[22] J.N. Hooker, H. Yan, I.E. Grossmann and R. Raman, Logic Cuts for Processing Networks with Fixed Costs. *Computers and Operations Research* **21** (1994) 265–279.

[23] M.S. Hung, W.O. Rom and A.D. Waren, *Optimization with IBM OSL.* Boyd & Fraser Publishing Company, Danvers, MA (1994).

[24] T. Hürlimann, IP, MIP and Logical Modeling using LPL. Working Paper No. 205, Institute of Informatics, University of Fribourg (March 1994, updated November 1994).

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

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

[27] C.V. Jones, Attributed Graphs, Graph-Grammars and Structured Modeling. *Annals of Operations Research* **38** (1992) 281–324.

[28] C.V. Jones, An Integrated Modeling Environment Based on Attributed Graphs and Graph-Grammars. *Decision Support Systems* **10** (1993) 255–275.

[29] C.V. Jones and K. D'Souza, Graph-Grammars for Minimum Cost Network Flow Modeling. Technical Report, Faculty of Business Administration, Simon Fraser University, Burnaby, BC (1992).

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

[31] D.A. Kendrick, A Graphical Interface for Production and Transportation System Modeling: PTS. *Computer Science in Economics and Management* **4** (1991) 229–236.

[32] B. Kristjansson, *MPL Modelling System User Manual,* Version 2.8. Maximal Software Inc., Arlington, VA (1993).

[33] Jean-Louis Lauriere, A Language and a Program for Stating and Solving Combinatorial Problems. *Artificial Intelligence* **10** (1978) 29–127.

[34] Lindo Systems Inc., *LINGO Optimization Modeling Language.* Chicago, IL (1994).

[35] K. McAloon and C. Tretkoff, 2LP: Linear Programming and Logic Programming. In V. Saraswat and P. Van Hentenryck, eds., *Principles and Practice of Constraint Programming,* The MIT Press, Cambridge, MA (1995) 99–114.

[36] R.D. McBride, NETSYS — A Generalized Network Modeling System. Technical Report, University of Southern California, Los Angeles, CA (1988).

[37] G. Mitra, C. Lucas, S. Moody and E. Hadjiconstantinou, Tools for Reformulating Logical Forms into Zero-One Mixed Integer Programs. *European Journal of Operational Research* **72** (1994) 262–276.

[38] G.L. Nemhauser, M.W.P. Savelsbergh and G.L. Sigismondi, MINTO, a Mixed INteger Optimizer. *Operations Research Letters* **15** (1994) 48–59.

[39] W. Ogryczak, K. Studziński and K. Zorychta, DINAS: A Computer-Assisted Analysis System for Multiobjective Transshipment Problems with Facility Location. *Computers and Operations Research* **19** (1992) 637–647.

[40] W. Ogryczak, K. Studziński and K. Zorychta, 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 (1992) 197–212.

[41] T.F. Rutherford, Extensions of GAMS for Complementarity Problems Arising in Applied Economic Analysis. Technical report, Department of Economics, University of Colorado, Boulder (1994); forthcoming in *Journal of Economic Dynamics and Control.*

[42] D. Steiger, R. Sharda and B. Leclaire, 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 (1992) 213–229.

[43] D. Steiger, R. Sharda and B. Leclaire, Graphical Interfaces for Network Modeling: A Model Management System Perspective. *ORSA Journal on Computing* **5** (1993) 275–291.

[44] L. Sterling and E. Shapiro, *The Art of Prolog: Advanced Programming Techniques,* 2nd ed. MIT Press, Cambridge, MA (1994).

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

[46] P. Van Hentenryck, *Constraint Satisfaction in Logic Programming.* MIT Press, Cambridge, MA (1989).

[47] P. Van Hentenryck, A Logic Language for Combinatorial Optimization. *Annals of Operations Research* **21** (1989) 247–273.