Mathematical programming formulations using AMPL

Darin England

September 4, 2011

Introduction

This article is a supplement to the course IE 5112, "Introduction to Operations Research", in the Industrial and Systems Engineering Program at the University of Minnesota, Twin Cities. We formulate four different types of mathematical programming problems using AMPL (A Modeling Language for Mathematical Programming). The best way to learn AMPL is to use it, and the purpose of this article is simply to provide more examples with which you may experiment. Along the way we will illustrate various features of the language; however, the complete reference for the AMPL language is the AMPL book [2]. This highly recommended book contains clear explanations of several different classes of problem formulations in mathematical programming.

Why use AMPL? Well, it's not AMPL in particular, but rather mathematical modeling languages in general that are extremely useful when *formulating* mathematical programming problems. An algebraic modeling language allows one to easily specify and understand objective functions, constraints, and logical relationships among variables. Of all the modeling languages for mathematical programming that I have seen, I like AMPL best. The syntax is simple, yet expressive enough to enable many different types of problems to be directly modeled. Although the language contains constructs for conditional execution of statements and for iterative looping, AMPL is primarily a *modeling* language, not a programming language. Its purpose is to facilitate translation from a problem description into a form that is suitable for a solver to process.

Resource allocation

This example is taken from [1]. A forester has 100 acres of hardwood timber. She also has $4000 cash on-hand to use for the forestry business. There are two possible courses of action to take with the 100 acres of available hardwood:

1. Harvest the hardwood and let the area naturally regenerate. This would cost $10 per acre now and return $50 per acre later, yielding a profit of $40 per acre.

2. Harvest the hardwood and plant the area with pine. This would cost $50 per acre now and return $120 per acre later, yielding a profit of $70 per acre.

Option 2 is clearly more profitable; however, the forester must respect her budget of $4000. The mathematical programming problem is to maximize total profit subject to the resource constraints, which are the acres of available hardwood and the budget.

The complete problem formulation in AMPL is shown in figure 1. Two decision variables, `x1` and `x2`, represent the number of acres that the forester should allocate to each of the two possible courses of action. It would make no sense to allow these decision variables to take on negative values. The non-negativity requirements are more cleanly specified directly in the variable declarations as opposed to creating separate constraints. A graphical depiction of the feasible solution space is shown in figure 2.

The constraints labeled `acres` and `budget` represent the limitations on resource availabilities. It's unnecessary to explicitly name constraints in AMPL; however, providing short, descriptive names aids interpretation of the model. Moreover, in a resource allocation problem such as this, the constraint names refer to the values of the associated dual variables, which have an economic interpretation as the marginal value of an additional unit of the resource.

```
var x1 >= 0;  # acres to fell and let regenerate
var x2 >= 0;  # acres to fell and plant with pine

maximize profit: 40*x1 + 70*x2;

s.t. acres: x1 + x2 <= 100;
s.t. budget: 10*x1 + 50*x2 <= 4000;
```

Figure 1: Forestry problem (`forester.mod`)

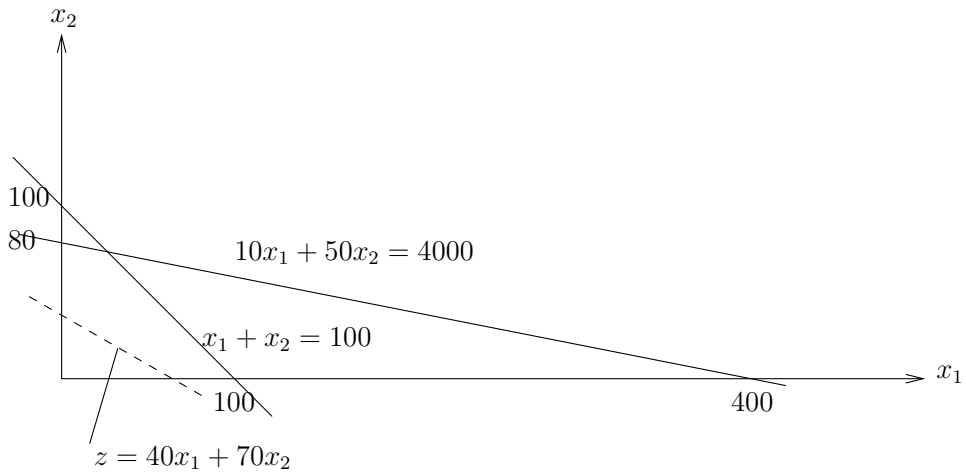

Figure 2: Feasible solution space

Since this problem is small, we include all data parameters directly in the model specification. However, it is usually good practice to separate the data from the model, and AMPL encourages this separation by providing `param` declarations and the ability to have a separate `data` section. We will illustrate these features of AMPL in the other problem formulations in this article. AMPL doesn't actually solve the mathematical programing problem, but rather passes a description of the problem to a solver, which returns information about the solution (if any solution was found) to AMPL. An AMPL session for the forestry problem follows:

```
ampl: model forester.mod;
ampl: solve;
MINOS 5.5: optimal solution found.
2 iterations, objective 6250
ampl: display x1,x2;
x1 = 25
x2 = 75
```

The optimal solution indicates that the forester should harvest 100 acres, but let 25 acres regenerate naturally and plant 75 acres with pine, for a profit of $6250. Notice that the entire budget of $4000 is exhausted because $10 \times 25 + 50 \times 75 = 4000$. When a resource capacity constraint such as `acres` or `budget` holds with equality in the optimal solution (as in this example), then the constraint is *binding*; all of the resource associated with the constraint is being consumed. If more/less of the resource were available, then profit would be increased/decreased. The value of the associated dual variable indicates the amount by which profit would be affected for small changes in the supply of the resource. This is called the marginal value of a resource, or the shadow price.

```
ampl: display acres, budget;
acres = 32.5
budget = 0.75
```

Displaying the shadow prices in AMPL indicates that an additional acre of hardwood timber would increase profit by $32.50, given the same budget of $4000. Modifying the right-hand side of `acres` to 101 and re-solving shows that this is indeed the case.

```
ampl: reset;
ampl: model forester.mod;
ampl: expand acres;
subject to acres:
x1 + x2 <= 101;

ampl: solve;
MINOS 5.5: optimal solution found.
2 iterations, objective 6282.5
ampl: display x1,x2;
x1 = 26.25
x2 = 74.75
```

The `expand` command displays the full form of a set of constraints. (This will be useful for indexed expressions.) Notice that the values of the decision variables have necessarily changed and that the solution is no longer integer-valued. This is typical of resource allocation problems. We are in fact assuming that the forester is able to execute the decisions on partial acres. For the `budget` constraint, an additional one dollar increase/decrease in the right-hand side would increase/decrease profit by $0.75, given the same 100 acres of hardwood. It stands to reason, then, that the forester could take out a loan and apply the funds to her operation. As long as the interest rate is less than .75, profit will increase.

Shadow prices are valid for *limited* increases/decreases in the right-hand sides of the constraints. The ranges over which the these values are valid is a topic of sensitivity analysis. Here, we will show how to extract this information from AMPL. First we need to tell AMPL to use a solver that is able to return sensitivity information along with the optimal solution. We will use the solver `cplex`. Then we set a solver-specific option to make the sensitivity range information available.

```
ampl: option solver cplex;
ampl: option cplex_options 'sensitivity';
ampl: solve;
CPLEX 11.2.0: sensitivity
CPLEX 11.2.0: optimal solution; objective 6250
2 dual simplex iterations (1 in phase I)

suffix up OUT;
suffix down OUT;
suffix current OUT;
```

We are now able to display the ranges over which the current shadow prices of \$32.5 per acre and \$0.75 per dollar of budget are valid. To do this we simply add a suffix to the name of the constraint, separated by a ".". The suffix `.current` displays the current value of a constraint's right-hand side, while `.down` and `.up` display the lower and upper limits, respectively. In the AMPL session follows, the upper limit of 5000 on the right-hand side of `budget` indicates the forester should only consider loans of \$1000 or less to be valued at an incremental marginal value of 0.75.

```
ampl: display acres.down, acres.current, acres.up;
acres.down = 80
acres.current = 100
acres.up = 400

ampl: display budget.down, budget.current, budget.up;
budget.down = 1000
budget.current = 4000
budget.up = 5000
```

Sensitivity ranges may also be obtained for the decision variables. In this case the lower and upper limits represent the valid ranges on the objective function coefficients over which the current solution remains optimal. The suffix `.current` refers to the current value of the coefficient. Displaying this information for the forestry problem reveals that the current solution value of `x1 = 25` remains optimal as long as its coefficient in the objective function is in the range 14 to 70, holding all other parameters at their current values.

```
ampl: display x1.down, x1.current, x1.up;
x1.down = 14
x1.current = 40
x1.up = 70

ampl: display x2.down, x2.current, x2.up;
x2.down = 40
x2.current = 70
x2.up = 200
```

Perfect matching

There are 10 students to be assigned to 5 dorm rooms. Each room holds exactly two students. For each pair of students, a value that indicates the desirability of placing that pair in the same room has been determined. Higher values correspond to better pairings. We would like to pair the students in such a way that the sum total of the values of each assigned pair is maximized. More generally, the perfect matching problem is to assign n pairs for $2n$ objects (with or without an objective function). The model and data are presented in figure 3.

`Pair` is a compound set with dimension two. Each member of this set is an ordered pair of students. We use the term "ordered" because the member (3,7) is distinct from the member (7,3). The `within` phrase

```
set Student;
set Pair within Student cross Student;

param value {Pair};

var x {Pair} binary;

maximize total_value: sum {(i,j) in Pair} value[i,j] * x[i,j];

s.t. perfect_match {i in Student}:
        sum {(i,j) in Pair} x[i,j] + sum {(j,i) in Pair} x[j,i] = 1;

data;
set Student := 1 2 3 4 5 6 7 8 9 10;

param: Pair: value:
   1  2  3  4  5  6  7  8  9 10 :=
1  .  .  .  .  .  .  .  .  .  .
2  3  .  .  .  .  .  .  .  .  .
3  5  8  .  .  .  .  .  .  .  .
4  1 -4  7  .  .  .  .  .  .  .
5  2 -1  9  2  .  .  .  .  .  .
6  2  5  3  2  9  .  .  .  .  .
7  8  2  1  1  3 -2  .  .  .  .
8  2  3  3  4  5  1  1  .  .  .
9 13 -1  3  4  4 -5  2  2  .  .
10 1  2  6  6  7 -4  5  6  1  .;
```

Figure 3: Model and data for perfect matching problem (`roommates.mod`)

tells us that the only allowed members in `Pair` are ordered pairs from the set `Student`. Such restrictions in the declaration of sets are encouraged in order to help detect errors in the data. For example, the following alternative declaration is perfectly acceptable, but using it would not allow the AMPL translator to recognize invalid pairs of students in the data section.

```
set Pair dimen 2;
```

The declaration of `Pair` is not strictly necessary. We could formulate the problem using only the set `Student`. However, using the compound set `Pair` makes the model easier to read and to maintain, particularly the indexing expressions. We specify a parameter `value` that is indexed over `Pair` to represent the desirability of matching up a particular pair of students. Our decision variables `x` are binary variables that are also indexed over `Pair`. `x[i,j]` will equal one if student `i` is matched with student `j`, and will equal zero otherwise. Notice the `binary` modifier in the declaration of `x`. This means that our decision variables may *only* take on the values zero or one and turns our formulation into one of pure integer programming [4].

The problem formulation itself consists of the objective function and a single set of constraints, one for each student. Our objective is to maximize the overall value of the assignments. It provides a good example of how to iterate over a compound set in AMPL. Notice that we must provide a pair of dummy variables `(i,j)` to index into `Pair`.

The constraints `perfect_match` state that each student must be matched with exactly one other student. To understand how this is accomplished note that the scope of the index `i` extends from its introduction in `i in Student` until the end of the statement marked by the semi-colon. In the expression `sum {(i,j) in Pair} x[i,j]`, `i` is held constant for a particular student (row of `value`) and the summation is over all room mates such that the pair of students represented by `(i,j)` exist in the set `Pair`, i.e. the summation is over columns. In the second expression with `i` and `j` interchanged, the summation is over rows. This is necessary due to the way that the data for `Pair` are structured. Notice that only the lower left portion of `value` is filled in.

This example demonstrates how to simultaneously define a set (`Pair`) and a parameter (`value`) that is indexed over that set. The "`.`" symbols indicate missing values, e.g. there exists no member `(1,1)` in the set `Pair`. An AMPL session for solving this problem follows. Although not strictly necessary for this particular problem, we will instruct AMPL to use a solver that can handle integer programming problems, such as the solver `gurobi`.

```
ampl: option solver gurobi;
ampl: model roommates.mod;
ampl: solve;
Gurobi 3.0.0: optimal solution; objective 39
14 simplex iterations
ampl: display x;
x [*,*]
:    1   2   3   4   5   6   7   8   9    :=
2    0   .   .   .   .   .   .   .   .
3    0   1   .   .   .   .   .   .   .
4    0   0   0   .   .   .   .   .   .
5    0   0   0   0   .   .   .   .   .
6    0   0   0   0   1   .   .   .   .
7    0   0   0   0   0   0   .   .   .
8    0   0   0   1   0   0   0   .   .
9    1   0   0   0   0   0   0   0   .
10   0   0   0   0   0   0   1   0   0
;
```

The large number of zeros makes the solution difficult to read. A useful option is to only display the decision variables with nonzero values. Then we can easily see that the pairs are (3,2),(6,5),(8,4),(9,1), and (10,7).

```
ampl: option omit_zero_rows 1;
```

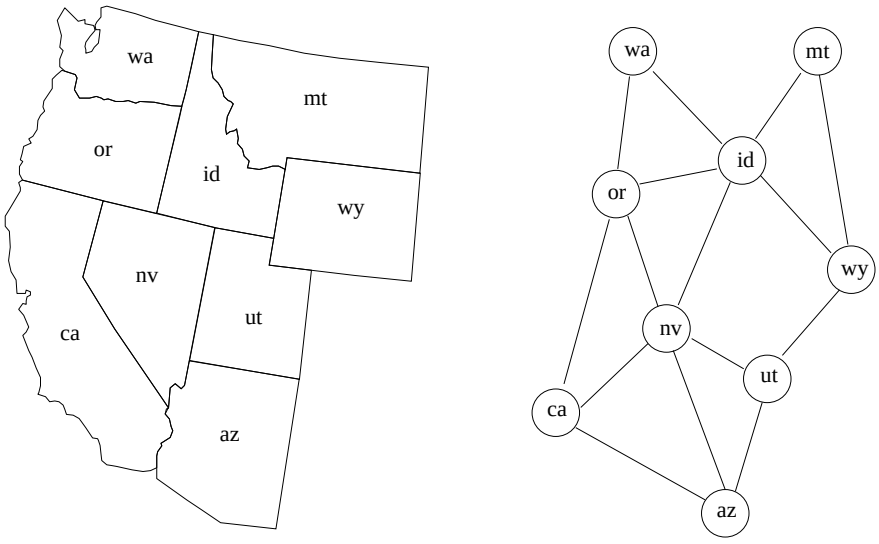

Figure 4: Western states and adjacency graph

```
ampl: display x;
x :=
3  2   1
6  5   1
8  4   1
9  1   1
10 7   1
;
```

Map coloring

The map coloring problem is to assign a color to each area on a map (e.g. state or county) in such a way that adjacent areas (i.e. those areas that share a border) are assigned different colors. Moreover, we would like to use as few colors as possible. The formulation for this problem was adapted from code written in GNU MathProg by Andrew Makhorin [3].

We may create a representation of the adjacency information by constructing a graph wherein each area on the map is represented by a node. An edge is drawn between two nodes if the respective areas on the map share a border. As an example, figure 4 shows the map and the corresponding adjacency graph for the western United States. The graph representation will better facilitate the requirement that adjacent areas must be assigned different colors.

Both the model and the data are presented in figure 5. In this model we must make the cardinality of the set `Color` large enough to effect a feasible solution. For a large problem the minimum number of colors needed will not be obvious (otherwise there is no need to solve the problem.) However, if the solver finds that the problem is infeasible we can always add more members to the set `Color`. See Andrew Makhorin's code [3] for an implementation of a heuristic to obtain an upper bound on the required number of colors.

Our binary decision variables `x` will indicate the particular color assigned to each node (area on the map). We also create binary modeling variables `u` to indicate that a color has been used in the solution. Now the objective is a simple summation over `u`. There are three dummy variables in the indexing expression for the `different` constraint. Notice that even though `x` is indexed over {`Node,Color`} the indexing expression for `different` will only create one constraint for each existing `Edge` and `Color` combination. Since `Edge` is declared to be `within (Node cross Node)` we may safely use the dummy indices `i` and `j` when indexing into `x`.

In the `data` section we see one way to specify the members of a two-dimensional set: the "`+`" symbols

```
set Node;
set Edge within (Node cross Node);
set Color;
# the cardinality of the set Color needs to be large enough
# to find a feasible solution

var x {Node,Color} binary; # x[i,c] = 1 means that node i is assigned color c
var u {Color} binary;      # u[c] = 1 means that color c is used

minimize num_colors: sum {c in Color} u[c];

s.t. assignment {i in Node}: sum {c in Color} x[i,c] = 1;
# each node is assigned exactly one color

s.t. different {(i,j) in Edge, c in Color}: x[i,c] + x[j,c] <= u[c];
# adjacent nodes must be assigned different colors

data;
set Color := red blue green yellow orange;

set Node := ca or wa mt id nv wy ut az;

set Edge:
   ca or wa mt id nv wy ut az :=
ca -  +  -  -  -  +  -  -  +
or -  -  +  -  +  +  -  -  -
wa -  -  -  -  +  -  -  -  -
mt -  -  -  -  +  -  +  -  -
id -  -  -  -  -  +  +  +  -
nv -  -  -  -  -  -  -  +  +
wy -  -  -  -  -  -  -  +  -
ut -  -  -  -  -  -  -  -  +
az -  -  -  -  -  -  -  -  -;
```

Figure 5: Model and data for map coloring problem (`kcolor.mod`)

indicate membership while the "-" symbols indicate non-membership. Alternatively, we could have specified `Edge` by providing only the ordered pairs.

```
set Edge :=
(ca,or)   (ca,az)   (or,id)   (wa,id)   (mt,wy)   (id,wy)   (nv,ut)   (wy,ut)
(ca,nv)   (or,wa)   (or,nv)   (mt,id)   (id,nv)   (id,ut)   (nv,az)   (ut,az);
```

Factory planning

This example is taken from [5]. A factory makes seven products that require various amounts of time on four different types of machines. The factory owns four grinders, two vertical drills, three horizontal drills, one boring machine, and one planing machine. For each product manufactured, the company can either sell the product (subject to market limitations) or hold the product in inventory at a cost of .5 per unit per month. We would like to develop a production and inventory plan for each of the next six months. We will not consider the sequence of machine operations; however, there is a fixed maintenance schedule that specifies when and how many of each machine type will be unavailable. The factory operates two eight-hour shifts each working day. There are 24 working days each month.

Refer to the data in figure 7 as well as the model in figure 6 while reading this description. Notice that the set `Month` is declared to be of type `ordered`, indicating a defined ordering among its (symbolic) members. This means that we can refer to the members of `Month` by their relative position. For example, in our data the expression `first(Month)` refers to the member `'jan'`. One common reason for declaring a set to be `ordered` is the need to refer to the previous and/or the next member in an indexed expression. A typical example of this use of an `ordered` set is illustrated by the `balance` constraints.

A defining feature of this model is use of three different variables to represent the quantities and timing for making, selling, and holding each product. The relationship among these variables is stated in the `balance` constraints. An upper bound of 100 units on the `hold` variable represents a storage limitation for each product in each month.

Our objective is to maximize the total profit of the factory over a period of six months. A unit profit is accrued for each item sold and a unit cost of .5 is incurred for each item held in inventory in a given month. Note that the parentheses surrounding the subtraction are necessary because the `sum` operator has higher precedence than `-`. As a side note, if we were to remove the holding cost from consideration in the objective then no parentheses would be required around the `profit[i]*sell[t,i]` term because the `sum` operator has lower precedence than `*` [2].

Each product requires a certain amount of processing time (possibly zero) on each type of machine. These requirements are specified in the data by the parameter `time_required` that is indexed over `Product` and `Machine` We need to specify constraints on the amount of machine time available each month. There are `work_hours` production hours available each month on each machine unless a machine is down for maintenance. Because the maintenance schedule is specific to each machine and month, the machine capacity constraints are indexed over `Month` and `Machine`. The total number of machine hours for all products must not exceed the time available in any particular month (respecting the `downtime` schedule.) There is an upper bound on the amount of each product that the market will absorb (i.e. that can be sold) each month.

We have three different variables to represent the decisions to make, sell, and hold product. The logical relationship among these variables is that in any particular month the amount sold plus the amount held in inventory must equal the amount produced plus any inventory from the previous month. When specifying these product balance constraints we must pay attention to the boundary conditions when indexing over `Month`. In our problem, we cannot refer to the previous month when the dummy index evaluates to `'jan'`. We can handle this in the main `balance` equations by placing a condition on the indexed set such that the order of the member is greater than 1 (and so `'jan'` is omitted.) This is possible because we declared the set `Month` to be of type `ordered`. We then need to specify a separate set of constraints for the first month (`balance0`). There is no beginning inventory and so the amount produced in `'jan'` equals the amount sold plus the amount held.

Instead of using the expression `make[first(Month),i]`, it would have been legitimate to refer to `make['jan',i]`; however, it's better practice to separate the model from the data. The model may then be applied to other problem instances with the same structure, but perhaps with a different starting month. Finally, we would

```
set Product;
set Machine;
set Month ordered;

param profit {Product} >= 0;
param time_required {Product,Machine} >= 0;
param num_available {Machine} integer, >= 0;
param downtime {Month,Machine} integer, >=0;
param market_limit {Month,Product} integer, >= 0;
param work_hours := 2*8*24;
# number of working hours in a month: 2 shifts of 8 hours each, 24 days/month

var make {Month,Product} >=0;  # how much of each product to make in each month
var sell {Month,Product} >=0;  # how much to sell
var hold {Month,Product} >=0, <=100;  # how much to hold

maximize total_profit:
  sum {t in Month, i in Product} (profit[i]*sell[t,i] - 0.5*hold[t,i]);

s.t. capacity {t in Month, m in Machine}:
sum {i in Product} time_required[i,m]*make[t,i]
  <= work_hours*(num_available[m]-downtime[t,m]);

s.t. marketing {t in Month, i in Product}: sell[t,i] <= market_limit[t,i];

s.t. balance {t in Month, i in Product : ord(t) > 1}:
  hold[prev(t),i] + make[t,i] = sell[t,i] + hold[t,i];
# on-hand inventory plus number produced must equal number
# sold plus number held in inventory for the next period

s.t. balance0 {i in Product}:
  make[first(Month),i] = sell[first(Month),i] + hold[first(Month),i];
# there is no inventory held over from december

s.t. end_inventory {i in Product}: hold[last(Month),i] = 50;
# stipulate that 50 of each product are to be held over from june
```

Figure 6: Model for factory planning problem (`factory_planning1.mod`)

```
data;
set Product := 1 2 3 4 5 6 7;
set Machine := grinder vdrill hdrill borer planer;
set Month := jan feb mar apr may jun;

param profit := 1 10 2 6 3 8 4 4 5 11 6 9 7 3;

param num_available :=
grinder 4 vdrill 2 hdrill 3 borer 1 planer 1;

param time_required (tr) :
         1   2   3   4   5   6   7 :=
grinder .5  .7   0   0  .3  .2  .5
vdrill  .1  .2   0  .3   0  .6   0
hdrill  .2   0  .8   0   0   0  .6
borer  .05 .03   0 .07  .1   0 .08
planer   0   0 .01   0 .05   0 .05;

param downtime :
     grinder vdrill hdrill borer planer :=
jan    1       0     0      0     0
feb    0       0     2      0     0
mar    0       0     0      1     0
apr    0       1     0      0     0
may    1       1     0      0     0
jun    0       0     1      0     1;

param market_limit :
       1     2     3     4     5     6     7 :=
jan  500  1000   300   300   800   200   100
feb  600   500   200     0   400   300   150
mar  300   600     0     0   500   400   100
apr  200   300   400   500   200     0   100
may    0   100   500   100  1000   300     0
jun  500   500   100   300  1100   500    60;
```

Figure 7: Data for factory planning problem (`factory_planning1.mod`)

like to end June with 50 of each product type in inventory. This is specified by the equality constraints named `end_inventory`.

References

[1] Vašek Chvátal. *Linear Programming*. W. H. Freeman and Company, 1983.

[2] Robert Fourer, David M. Gay, and Brian W. Kernighan. *AMPL: A Modeling Language for Mathematical Programming*. Brooks/Cole Publishing Company, second edition, 2003.

[3] Glpk (gnu linear programming kit). `www.gnu.org/software/glpk`.

[4] George L. Nemhauser and Laurence A. Wolsey. *Integer and Combinatorial Optimization*. John Wiley and Sons, 1988.

[5] Paul Williams. *Model Building in Mathematical Programming*. John Wiley and Sons, fourth edition, 1999.