

AMPL and CPLEX tutorial

Gábor Pataki

August 22, 2014

1 The steel production problem

1.1 The problem

2 products can be produced at a steel mill:

- We can make 200 tons of product 1 in an hour; the profit for each ton is 25 dollars; the demand is 6000 tons. We must make at least 1000 tons of this product.
- We can make 140 tons of product 2 in an hour; the profit for each ton is 30 dollars; the demand is 4000 tons. We must make at least 2000 tons of this product.

We have 40 hours of production time available.

The goal is to design a production plan to maximize total profit.

With x_i = tons of product i to be made, we get the following LP:

$$\begin{array}{ll} \max & 25x_1 \qquad \qquad \qquad +30x_2 \\ \text{st.} & x_1 \geq 0, \qquad \qquad \qquad x_2 \geq 0 \\ & \frac{1}{200}x_1 \qquad \qquad \qquad + \frac{1}{140}x_2 \leq 40 \\ & 1000 \leq x_1 \leq 6000 \\ & 2000 \leq x_2 \leq 4000 \end{array} \tag{1}$$

1.2 Writing and running a correct model

The simplest version of the steel problem's solution is below:

```
### File: steel-simple.mod

var x1 >=1000, <= 6000;
var x2 >=2000, <= 4000;

maximize profit: 25*x1 + 30*x2;

subject to time: (1/200)*x1 + (1/140)*x2 <= 40;

## here: 'profit' and 'time'
## are arbitrarily chosen names.
```

AMPL run (the output may look slightly different, depending on the version).

```

ampl: model steel-simple.mod;
ampl: option solver cplex;
ampl: solve;
CPLEX 12.6.0.0: optimal solution; objective 188571.4286
0 dual simplex iterations (0 in phase I)
ampl: display x1, x2;
x1 = 1000
x2 = 4000
ampl:

```

When we split the problem into model and data files, they look like this:

```

### File: steel.mod

param n :=2;

param a {j in 1..n};
param b;
param c {j in 1..n};
param u {j in 1..n}; # Upper bound on production
param l {j in 1..n}; # Lower bound on production

var x {j in 1..n} <= u[j], >= l[j];

maximize profit: sum {j in 1..n} c[j] * x[j];

subject to time: sum {j in 1..n} (1/a[j]) * x[j] <= b;

## here: 'param' and 'var' are reserved keywords; 'profit' and 'time'
## are arbitrarily chosen names.

-----

### File: steel.dat

param      a  1 200
           2 140;

param      c  1 25
           2 30;

param      u  1 6000
           2 4000;

param      l  1 1000
           2 2000;

param b := 40;

```

AMPL run

```
AMPL run
ampl: model steel.mod;
ampl: data steel.dat;
ampl: option solver cplex;
ampl: solve;
CPLEX 12.6.0.0: optimal solution; objective 188571.4286
0 dual simplex iterations (0 in phase I)
ampl: display x;
x [*] :=
1 5142.86
2 2000
;
```

```
ampl:
```

```
### Remark: the number of iterations can be 0,
### if the problem is very simple;
### in that case, a so-called 'LP preprocessor'
### already solves the problem.
```

```
### We can change some of the data, and resolve:
```

```
ampl: let u[1] := 5000;
ampl: display x.ub;
x.ub [*] :=
1 5000
2 4000
;
```

```
### x.ub is the current upper bound on x; the .ub extension works for
### any variable. I.e. we can write y.ub, if y is a variable vector.
### Similarly, x.lb gives the lower bounds.
```

```
ampl: solve;
CPLEX 12.6.0.0: optimal solution; objective 188000
0 simplex iterations (0 in phase I)
```

1.3 Inputting a general problem

This is how to input an LP of the form

$$\begin{array}{ll} \max & c^T x \\ \text{st.} & Ax \leq b \end{array}$$

where A is a matrix with m rows, and n columns.

The A, b, c for the problem will be

$$A = \begin{pmatrix} 5 & 6 \\ 7 & 8 \\ 9 & 10 \end{pmatrix}, b = \begin{pmatrix} 13 \\ 15 \\ 19 \end{pmatrix}, c = \begin{pmatrix} 5 \\ 6 \end{pmatrix}.$$

Model file:

```
param n;
param m;

param A {i in 1..m, j in 1..n};
param b {i in 1..m};
param c {j in 1..n};

var x {j in 1..n};

maximize whatever: sum {j in 1..n} c[j] * x[j];

subject to cons {i in 1..m}:
    sum {j in 1..n} A[i,j]*x[j] <= b[i];
```

Data file:

```
param n :=2;
param m :=3;

param A:   1 2 :=
           1 5 6
           2 7 8
           3 9 10;

param b :=
           1 13
           2 15
           3 19;

param c :=
           1 5
           2 6;
```

1.4 Debugging an incorrect model

Suppose we have set up the data in a way, so the LP is infeasible. Usually in an infeasible LP, there are only a few constraints which result in infeasibility. As an extreme example, in:

$$x_1 \leq 2, x_1 \geq 3, 0 \leq x_i \leq 1 (i = 2, \dots, 1000)$$

the only constraints that cause trouble, are the first two. They form a so called *irreducible infeasible system*; that is, a subset of all inequalities, which are infeasible, but dropping any one of them would make this system feasible. (i.e. $x_1 \leq 2, x_1 \geq 3$ is infeasible, but dropping any one of these gives just one inequality, which is of course feasible).

For instance, this data makes the steel problem infeasible:

```
### File: steel.dat
```

```
param a 1 200
      2 140;
```

```
param c 1 25
      2 30;
```

```
param u 1 6000
      2 4000;
```

```
param l 1 4000
      2 3000;
```

```
param b := 40;
```

```
AMPL run:
```

```
ampl: solve;
```

```
presolve: constraint time cannot hold:
```

```
      body <= 40 cannot be >= 41.4286; difference = -1.42857
```

```
### Not too useful info... We will find an IIS, to localize the problem.
```

```
ampl: option presolve 0;
```

```
### This tells the solver to turn the preprocessor off.
```

```
ampl: option cplex_options 'iisfind 1';
```

```
### This tells the solver to find an IIS.
```

```

ampl: solve;
CPLEX 12.6.0.0: iisfind 1
CPLEX 12.6.0.0: infeasible problem.
Returning an IIS of 2 variables and 1 constraints.
No basis.

```

```

suffix iis symbolic OUT;

```

```

option iis_table '\
0 non not in the iis\
1 low at lower bound\
2 fix fixed\
3 upp at upper bound\
4 mem member\
5 pmem possible member\
6 plow possibly at lower bound\
7 pupp possibly at upper bound\
8 bug\
';
ampl:

```

```

### Most of the above stuff is just technicalities
### that you can ignore...
### The important part comes below:

```

```

ampl: display x.iis;
x.iis [*] :=
1 low
2 low
;

```

```

ampl: display time.iis;
time.iis = upp

```

The meaning of the above lines is:

$$x_1 \geq 4000, x_2 \geq 3000, (1/200)x_1 + (1/140)x_2 \leq 40$$

is an IIS. (That is, the upper bounds on x have nothing to do with the infeasibility).

1.5 Another variant of the steel problem

The next model is the same, but it gives names to the products.

```

### File: steel2.mod

```

```

set P;

```

```

param a {j in P};
param b;
param c {j in P};
param u {j in P};
param l {j in P};

var x {j in P};

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

subject to time: sum {j in P} (1/a[j]) * x[j] <= b;

subject to limit {j in P}: l[j] <= x[j] <= u[j];

```

```

### File: steel2.dat

```

```

set P := bands coils;

```

```

param:  a :=  bands 200
        coils 140;

```

```

param:  c :=  bands 25
        coils 30;

```

```

param:  u :=  bands 6000
        coils 4000;

```

```

param:  l :=  bands 1000
        coils 2000;

```

```

param b := 40;

```

```

### AMPL run:

```

```

ampl: model steel2.mod;

```

```

ampl: data steel2.dat;
ampl: option solver cplex;

```

```

ampl: solve;
ampl: display x;

```

```

x [*] :=
bands 5142.86
coils 2000
;

```

Important!! If you make a mistake in a model, or data file, you will need to 1) fix it, 2) type “reset”, or “reset data” before you can reread those files. Example:

```
ampl: model steel.mod;
```

```
steel.mod, line 11 (offset 205):
```

```
    syntax error
```

```
context: m >>> aximize <<< profit: sum {j in 1..n} c[j] * x[j];
```

```
ampl?
```

```
### There was a mistake in the model file; we fix it, and reread it.
```

```
ampl: reset;
```

```
ampl: model steel.mod;
```

```
ampl: data steel.dat;
```

```
steel.dat, line 1 (offset 2):
```

```
    syntax error
```

```
context: p >>> aram <<<      a 1 200
```

```
ampl?
```

```
### Now the model file was OK, but there is a mistake in the data file;
```

```
### we fix the data file, and reread only the data file (the model file
```

```
### was OK to start with).
```

```
ampl? ;
```

```
ampl: reset data;
```

```
ampl: data steel.dat;
```

```
ampl:
```

2 The minimum cost flow problem

This problem is excellent to illustrate how to define variables x_{ij} , where the *existing* variables are just a small subset of the *possible* ones.

```
### File: mcf.mod
```

```
param n :=5;      # Number of nodes;
```

```
set ARCS within {1..n, 1..n};
```

```
param demand {1..n};
```

```
    check: sum {i in 1..n} demand[i] = 0;
```

```
### This statement will check that the sum of demands is zero, as one would
```

```

### expect for the problem to be feasible.

param cost {ARCS};
param u {ARCS};

var x {ARCS} >=0;

minimize total_cost:
    sum { (i,j) in ARCS } cost[i,j]*x[i,j];

subject to balance {i in 1..n}:
    sum { (j,i) in ARCS } x[j,i] - sum{ (i,j) in ARCS } x[i,j] = demand[i];

### File: mcf.dat

param demand := 1 1
2 3
3 5
4 -6
5 -3;

param: ARCS: cost := 1 2 10
    1 4 5
                1 5 7
    2 3 5
    2 4 6
    2 1 1
    3 1 5
    3 4 10
    3 5 1
    4 2 1
    4 5 6
    5 1 1
    5 2 3
    5 4 7;

```

3 A multiperiod problem

Suppose we have a multiperiod problem, with variables

- inv_i (for inventory at the end of period i), $prod_i$ (for production in period i , and
- parameters $demand_i$ (for demand in period i),
- constraints

$$inv_{i-1} + prod_i = demand_i + inv_i$$

We can write these constraints concisely as follows (only parts of the model and data files are written down):

```

### File: production.mod

param n;

param demand {1..n};

var inv {0..n}; # inventory;
var prod {1..n}; # production;

subject to balance {i in 1..n}:
    inv[i-1] + prod[i] = demand[i] + inv[i];

etc.

### File: production.dat

param n := 6;

param demand := 1 5
2 3
3 16
4 11
5 10
                6 7;

etc.

```

This has the following advantages, as opposed to writing out the 6 constraints individually:

- This is much cleaner, and easier to read.
- If the parameter n is not “hardwired” into the program, i.e. you do not write 6 in any place where n should be used, then the code is much more flexible. If you have 2 data files, one with say $n = 6$, the other with $n = 1000$, then you can use the same model file with both.

4 Some neat tricks

There are some useful internal variables in AMPL:

```

_nvar is the number of variables;
_var  is a vector containing the values of all variables;
_varname is a vector containing the names of all variables;

```

So in the steel problem, we can do:

```
AMPL: display _varname;
_varname [*] :=
1 'x[1]'
2 'x[2]'
;
```

```
AMPL: display _var;
_var [*] :=
1 5142.86
2 2000
;
```

```
AMPL: display {i in 1.._nvars: _var[i]>0} _varname[i], {i in 1.._nvars: _var[i]>0} _var[i];
: _varname[i] _var[i] :=
1 'x[1]' 5142.86
2 'x[2]' 2000
;
```

The last command displays the names and values of all variables with a positive value (in this instance, actually all variables have a positive value).