Notes on AMPL for 16.410 and 16.413

(Adapted from notes by Sommer Gentry)

User Guides for MS-Dos and UNIX:

`http://www.ampl.com/REFS/index.html#OS_Links`

About AMPL/CPLEX

AMPL stands for A Mathematical Programming Language, and it was invented at AT&T's Bell Labs by a group including Brian Kernighan, of C fame. AMPL is a scripting language that allows you to express mathematical programs in an intuitive manner. AMPL can translate those programs, if they are linear programs, integer programs, or quadratic programs, into a format that can be read by a number of dierent solvers including CPLEX and LP-SOLVE.

Gaining access to AMPL/CPLEX or AMPL/LPSOLVE

You have two options: download the student version of AMPL/CPLEX for your Windows or Unix computer at `http://www.ampl.com/DOWNLOADS/cplex71.html`, or run your AMPL programs on the web at `http://www.ampl.com/TRYAMPL/startup.html`. If you use the latter option, you must choose the `lpsolve` solver. The default solver will be `minos`, and `minos` does not solve integer programs such as the one on your homework. CPLEX is not available through the online version.

Writing Model and Data Files

AMPL separates the model, which describes the mathematical program to be solved; from the data, the numbers that specify one instance of the problem. The best way to gain experience is to look at some prewritten model and data files and try modifying them. You can find an entire directory full of sample model and data files listed for you at `http://www.ampl.com/BOOK/EXAMPLES/index_figs.html`

The model file generally consists of three sections: the declaration of types, the objective function, and constraints.

Declarations

There are three types of declarations you will need for this assignment: `set`, `param` and `var`. All that we do in the declaration section is declare what things are. For instance:

```
set NAMES;
```

declares that `NAMES` is a set, and should be treated as such wherever it appears. Sets are typically collections of names, places, numbers, etc. We don't say what's **in** the set in the model file: that is done in the data file.

A `param` is a fixed parameters of the problem. Again, we don't give the specific value here: we do that in the data file.

```
param time_limit;
```

declares that there is a parameter called `time_limit`. We can say that a param might be an array of values, such as in

```
param COSTS {INPUT};
```

which says that we will have a `COSTS` param, and in fact we will have a separate `COSTS` value for each different value of `INPUT`. We can do multi-dimensional arrays as well:

```
param COSTS {INPUT,OUTPUT};
```

We could also add constraints on the range of values a `param` can have, such as in

```
param COSTS {INPUT,OUTPUT} >=0;
```

which says that `COSTS` means an array of specific values, to be specified in the data file, which all have to be non-negative.

A `var` is a decision variable of the optimization. The form of its declaration is basically the same as a `param`. For instance,

```
var ASSIGN {ROBOTS,MISSIONS};
```

declares that the optimization has to choose appropriate values for `ASSIGN` for each value of `ROBOTS` and `MISSIONS`. A `var` can also be restricted by qualifiers following its definition, such as `>=0`, `integer`, or `binary`.

Objective function

The objective value is indicated by a line beginning `maximize ObjectiveName:` The format is usually

```
maximize ObjectiveName: sum {i in set} some function of variable
```

You can replace `maximize` with `minimize`, and you can do objectives involving multiple variables. For example,

```
minimize TOTAL_COST: sum {i in ROBOTS, j in MISSIONS} COST[i,j]
```

is the same thing as minimizing:

$$\sum_{i=1}^{n}\sum_{j=1}^{m} x_{i,j}, \tag{1}$$

where $x_{i,j}$ is the cost of assigning robot i to mission j.

Constraints

The constraints (in this file there is only one constraint) are named separately on lines beginning with `subject toConstraintName:` The format is very similar to that for the objective function, except that there is an inequality to be tested in the function. For example,

```
subject to TOTAL_COST: sum {i in ROBOTS, j in MISSIONS} COST[i,j] <= 100
```

would say that the `TOTAL_COST` is subject to the constraint that it sum to less than 100, that is

$$\sum_{i=1}^{n}\sum_{j=1}^{m} x_{i,j} \leq 100. \tag{2}$$

There are also constructs for all of the standard elements of mathematical programs, like upper and lower bounds on variables and sets of variables indexed by members of another set (like `Make`, indexed by members of a set like `ROBOTS`).

Data File

For each set you declared in the model, you provide a list of values. For instance:

```
set ROBOTS := HAL R2D2 C3P0;
```

now says that `ROBOTS` is a set of length 3. Notice the different equality (`:=`).

For each parameter you declared in the model, you also provide values. AMPL provides a very compact notation which can be hard to wrap your head around: you can actually create tables of values and set multiple parameters simultaneously.

If we had defined a `set PROD` and the `params rate profit commit market`, then we could set all the params in the following table:

```
set PROD := bands coils plate;

param:    rate  profit  commit  market :=
  bands    200    25     1000    6000
  coils    140    30      500    4000
  plate    160    29      750    3500 ;
```

Note that if you set multiple `params` simultaneously using a table like this, you have to make sure that all the `params` are over the same sets. You couldn't mix `rate` and `profit` if `rate` was defined in the model over the `PROD set`, but `profit` was defined in the model over something else like `TIME`.

AMPL is often finicky about transposing matrix data elements, so if you are having trouble with a two-dimensional data set, you might try adding `(tr)`, as in `param amount (tr): ROBOT1 ROBOT2 :=` when declaring numerical values in a matrix.

Solving

If you are running over the web, the web interface is pretty self-explanatory. (Don't forget to choose the appropriate solver.) But, if you are running locally, then you will need the following commands:

1. `option solver [solver];`
   If you have downloaded cplex, then you would choose that.
2. `model [filename];`
3. `data [filename];`
4. `solve;`
5. `display [variable];`

Example File

Choose the files `steel.mod` and `steel.dat`. These files implement a simple production decision model. The steel factory modelled has to decide which types of steel products to manufacture given profit, market limits, and production rates for each product. Notice that there are constructs for `sets`, `params` and `vars`The # mark denotes the rest of that line as a comment. The data file, then defines the elements included in a set and the numeric values of each `param`.

```
##################### STEEL.MOD ####################

## STARTING DECLARATIONS HERE

set PROD;  # PROD is a set that will have a specific
           # set of values

param rate {PROD} > 0;     # tons produced per hour
param avail >= 0;          # hours available in week
param profit {PROD};       # profit per ton
param market {PROD} >= 0;  # limit on tons sold in week

var Make {p in PROD} >= 0, <= market[p]; # tons produced

## END OF DECLARATIONS

## STARTING OBJECTIVE HERE

maximize total_profit: sum {p in PROD} profit[p] * Make[p];

               # Objective: total profits from all products

## STARTING CONSTRAINT HERE

subject to Time: sum {p in PROD} (1/rate[p]) * Make[p] <= avail;

               # Constraint: total of hours used by all
               # products may not exceed hours available

##################### STEEL.DAT ####################

set PROD := bands coils;

param:    rate  profit  market :=
  bands    200    25     6000
  coils    140    30     4000 ;

param avail := 40;
```

Example Execution

```
nickroy@oiler:[nickroy] 25>ampl
ampl: option solver cplex;
ampl: model steel.mod
ampl: data steel.dat;
ampl: solve;
CPLEX 8.0.0: optimal solution; objective 192000
0 dual simplex iterations (0 in phase I)
ampl: display Make;
Make [*] :=
bands  6000
coils  1400
;

ampl:
```

Running AMPL

The online version of AMPL shows the run commands to which the online version defaults in the `AMPL commands:` box. The default run commands will solve the optimization problem and then display the values of the decision variables. Remember to set the solver to be `lpsolve` instead of `minos`.

Explore

Play around with the steel model, adding a new product or a new constraint (like minimum production values for each product) to try to get a dierent optimal solution. To impose integrality constraints, just add the word `integer` to the variable declaration in the model file, like `var Buy {j in FOOD} integer >=0 ;` Another good example file to explore would be `multmip1.mod`, which implements a network shipping model with fixed costs for using each route.

More Info

For more information about AMPL, see www.ampl.com. The website for AMPL contains an online version of the first chapter of the user's manual and all of the sample files used in the book. You can glean many hints on AMPL syntax and shortcuts by studying the sample files. The FAQ is also helpful when you get stuck. The user's manual, [1], describes both AMPL and linear programming and is available in the MIT libraries.

References

[1] Robert Fourer, David M. Gay, and Brian W. Kernighan. AMPL: A Modeling Language for Mathematical Programming, 1993.