A short OPL tutorial

Truls Flatberg

January 17, 2009

1 Introduction

This note is a short tutorial to the modeling language OPL. The tutorial is by no means complete with regard to all features of OPL. More detailed information can be found in the Language User's Manual and the Language Reference Manual provided with the OPL installation. The motivation of this note is to show a few examples to get people quickly started using OPL.

OPL is a modeling language for describing (and solving) optimization problems. The motivation for using modeling languages to model optimization problems is primarily due to two reasons:

- It provides a syntax that is close to the mathematical formulation, thus making it easier to make the transition from the mathematical formulation to something that can be solved by the computer.
- It enables a clean separation between the model and the accompanying data. The same model can then be solved with different input data with little extra effort.

OPL is just one example of a modeling language. Other examples include AMPL, Mosel and GAMS. Modeling languages are typically used for linear and integer optimization problems, but it is also possible to formulate quadratic problems in OPL.

Before starting on the examples, we give a short overview of the structure of an OPL model. A model typically includes these four parts:

1. *Data declarations.* This part declares the data parameters used in the model, typically coefficients and index sets for the decision variables. Individual data parameters can be declared by giving their type and name, e.g.

   ```
   int a = 4;
   ```

   Sets of a specific type are declared by enclosing the type within curly braces

```
{int} I = {1, 2, 3, 4};
```

The sets can then be used to create arrays with an entry for each element in the set

```
float A[I] = [0, 1, 5, 2];
```

2. *Decision variables.* These are declared with the keyword `dvar`, e.g.

   ```
   dvar float+ x;
   ```

   declares `x` to be a non-negative real variable. Similarly to data, it is possible to create arrays of decision variables for a specific set.

3. *Objective function.* The objective function is declared with the keyword `minimize` or `maximize` depending on what you want to do. A simple example is

   ```
   maximize 4*x;
   ```

4. *Constraints.* The constraints are declared using the keyword `subject to`. The following declares two linear inequalities involving the decision variables `x` and `y`:

   ```
   subject to {
        4*x + 2*y <= 1;
       -2*x +   y >= 5;
   }
   ```

Note that all statements in the model file are terminated by a semicolon.

In addition to keywords used for describing optimization models, OPL provides a scripting language for non-modelling purposes. This scripting language can be used to display information about the solution, to interact with the model and alter it, or do multiple runs on the same model. The scripting language has a slightly different syntax from the modeling part and is encapsulated in an `execute` block. For example,

```
execute {
    writeln("x=", x);
}
```

will print the value of the variable `x`.

All model and data files discussed in this note can be downloaded from the author's website `http://home.ifi.uio.no/trulsf`. The easiest way to run the examples is to open the accompanying project file in OPL Studio.

2 A first model

We will start out with a minimal OPL model to solve the following linear optimization problem:

$$\begin{aligned} \text{minimize} \quad & 4x - 2y \\ \text{subject to} \quad & x - y \geq 1 \\ & x \geq 0 \end{aligned}$$

The OPL model is as follows:

Listing 1: simple.mod

```
1  dvar float x;
2  dvar float y;
3
4  minimize
5    4*x - 2*y;
6
7  subject to {
8    x - y >= 1;
9    x >= 0;
10 }
```

Note how similar the two formulations are. The constraint in line 9 can be removed and replaced with an explicit declaration of the variable `x` as type `float+`.

3 Production planning

We will continue with a production planning problem. A company produces two products: doors and windows. It has three production facilities with limited production time available:

- Factory 1 produces the metal frame
- Factory 2 produces the wooden frame
- Factory 3 produces glass and mounts the parts

The products are produced in series of 200 items and each series generates a revenue depending on the product. Each series require a given amount of time of each factory's capacity. The problem is to find the number of series of each product to maximize the revenue. This can be modelled as the following linear

program:

$$\begin{aligned} \text{maximize} \quad & r_d x_d + r_w x_w \\ \text{subject to} \quad & t_{d,1} x_d + t_{w,1} x_w \leq c_1 \\ & t_{d,2} x_d + t_{w,2} x_w \leq c_2 \\ & t_{d,3} x_d + t_{w,3} x_w \leq c_3 \\ & x \geq 0, y \geq 0 \end{aligned}$$

where r_p is the revenue for product p, $t_{p,i}$ is the time needed at factory i to produce a series of product p and c_i is the time available at factory i. The series produced is given by x_p for each product p.

The following is a formulation of the problem in OPL:

Listing 2: prod.mod

```
1  {string} Products = ...;
2  {string} Factories = ...;
3
4  float r[Products] = ...;
5  float t[Products][Factories] =  ...;
6  float c[Factories]  = ...;
7
8  dvar float+ x[Products];
9
10 maximize
11   sum(p in Products) r[p] * x[p];
12
13 subject to {
14   forall(i in Factories)
15     sum(p in Products) t[p][i] * x[p] <= c[i];
16 }
```

Lines 1 and 2 declare two arrays of strings defining the set of products and the set of factories. Note the use of '...' to indicate that the value will be provided in a separate data file. Lines 4-6 define the input data as real numbers: revenues, time required for each product and the capacity of each factory. Line 8 declares a non-negative decision variable for each product, lines 10-11 declare the objective and lines 13-16 declare the constraints, one for each factory by using the `forall` keyword.

Suppose now that we want to solve the problem with the following data:

	Hours/series		Hours at disposal
	Door	Window	
Factory 1	1	0	4
Factory 2	0	3	12
Factory 3	3	2	18
Revenue/series	3000	5000	

This can be translated to the following OPL data file:

Listing 3: prod.dat

```
Products = {"Door", "Window"};
Factories = {"Factory 1", "Factory 2", "Factory 3"};

r = [3000 5000];
t = [[1 0 3] [0 3 2]];
c = [4 12 18];
```

Note that the separation between model and data facilitates easy updating if part of the problem is changed. For instance, adding a new product to the problem, will only affect the data file and the model can be left unchanged.

4 Linear programming

This example shows how to solve a general linear programming problem, and illustrates how scripting can be combined with the model to display results and obtain details on the solution. The problem we want to solve can be written in matrix form as

$$\begin{aligned} \max \quad & c^T x \\ \text{s.t.} \quad & Ax \leq b, \\ & x \geq 0 \end{aligned}$$

The OPL model is as follows

Listing 4: lp.mod

```
int m = ...;
int n = ...;

range Rows = 1..m;
range Cols = 1..n;

float A[Rows][Cols] = ...;
float b[Rows] = ...;
float c[Cols] = ...;

dvar float+ x[Cols];

maximize
    sum(j in Cols) c[j] * x[j];

subject to
{
    forall(i in Rows)
    ct:
```

```
20          sum(j in Cols) A[i][j] * x[j] <= b[i];
21  }
```

In lines 4-5 we use a special kind of set, `range`, that can be used to represent a sequence of integer values. Note also that we give a name to the constraint in line 19. This name will be used as an identifier when we later want to obtain information about the constraint.

The problem instance can be specified in a separate data file. The following is the input for an example with four constraints and four variables.

Listing 5: lp.dat

```
1   m = 4;
2   n = 4;
3
4   A = [[3  2  2  1]
5        [2  3  2  1]
6        [0  1  4  3]
7        [1  3  2  4]
8       ];
9
10  b = [3  4  5  7];
11
12  c = [2  2  2  1];
```

In addition to the modeling features, OPL has a scripting language that can be used for several purposes. In the following example we show how to print both primal and dual solution information for the LP problem.

Listing 6: lp.mod

```
22  execute {
23      writeln("Optimal value: ", cplex.getObjValue());
24      writeln("Primal solution:");
25      write("x=[");
26      for(var j in Cols)
27      {
28          write(x[j], " ");
29      }
30      writeln("]");
31
32      writeln("Dual solution:");
33      write("y=[");
34      for(var i in Rows)
35      {
36          write(ct[i].dual, " ");
37      }
38      writeln("]");
39  }
```

Note the slightly different syntax used in the scripting language, e.g. variables are declared using the `var` keyword while iteration over a set is done using the `for` keyword. If an `execute` block is placed after the objective and the constraints, it will be performed as a postprocessing step. If placed beforehand, it can also be used as a preprocessing step.

5 Integer programming

In this section we show how to model and solve a problem with integer variables, in this case 0-1 variables. Consider the following problem: We want to construct a sequence of n numbers. The i'th position should give the number of times the number i occurs in the sequence, where the indexing starts at zero. The following is an example of a legal sequence for $n = 4$:

pos	0	1	2	3
value	1	2	1	0

The problem of constructing a feasible sequence for a given n can be solved as an integer problem. Let x_{ij} be a 0-1 variable that is one if position i has the value j and zero otherwise, and consider the following integer program

$$
\begin{array}{rlll}
\min & 0 & & \\
\text{subject to} & \text{(i)} & \sum_{j=0}^{n-1} x_{ij} = 1, & \text{for } i = 0, \ldots, n-1 \\
& \text{(ii)} & \sum_{k=0}^{n-1} x_{ki} - j \leq n(1 - x_{ij}), & \text{for } i, j = 0, \ldots, n-1 \\
& \text{(iii)} & \sum_{k=0}^{n-1} x_{ki} - j \geq n(x_{ij} - 1), & \text{for } i, j = 0, \ldots, n-1 \\
& \text{(iv)} & x_{ij} \in \{0, 1\} & \text{for } i, j = 0, \ldots, n-1
\end{array}
$$

Note that we do not need an objective function as we are only interested in a feasible solution. Constraint (i) ensures that each position has a value. While constraints (ii) and (iii) ensure that $x_{ij} = 1$ implies that $\sum_{k=0}^{n-1} x_{ki} = j$.

The following is the corresponding OPL model:

Listing 7: ip.mod

```
int n = 10;
range N = 0..n-1;

dvar boolean x[N][N];

minimize 0;

subject to
{
  forall(i in N)
    sum(j in N) x[i][j] == 1;
```

```
12
13    forall(i in N, j in N)
14      sum(k in N) x[k][i]-j <= n*(1-x[i][j]);
15
16    forall(i in N, j in N)
17      sum(k in N) x[k][i]-j >= n*(x[i][j] -1);
18  }
19
20  execute {
21    for(var i in N)
22    {
23        for(var j in N)
24        {
25            if (x[i][j]==1)
26            {
27                write(j, " ");
28            }
29        }
30    }
31    writeln();
32  }
```

Note that 0-1 variables are declared using the keyword `boolean`, while general integer variables are declared using the `int` keyword.

6 Minimum spanning tree

We end this tutorial with a slightly more complex model. In the minimum spanning tree problem we want to find a spanning tree of minimum weight in an undirected graph $G = (V, E)$ with weight function $w : E \to \mathbb{R}$. This problem can be solved efficiently as a combinatorial problem using e.g. Kruskal's algorithm. Here we will show how to formulate and solve the problem as an integer problem.

The following is a valid formulation of the problem as an integer problem:

$$
\begin{array}{lllll}
\min & \sum_{e \in E} w_e x_e & & & \\
\text{s.t.} & \text{(i)} & \sum_{e \in E} x_e = |V| - 1 & & \\
& \text{(ii)} & \sum_{e \in \delta(U)} x_e \geq 1, & \text{for all } U \subset V & \\
& \text{(iii)} & x_e \in \{0, 1\}, & \text{for all } e \in E &
\end{array}
$$

Constraint (ii) is a cut constraint that ensures that we do not get a solution with subcycles. Note that there is an exponential number of these constraints, clearly limiting the size of the problems solvable in OPL.

This can be implemented as an OPL model using some additional features of OPL:

Listing 8: mst.mod

```
int n = ...;
range Nodes = 1..n;
{int} NodeSet = asSet(Nodes);

tuple Edge
{
    int u;
    int v;
}
{Edge} Edges with u in Nodes, v in Nodes = ...;
{int} Nbs[i in Nodes] = { j | <i,j> in Edges};

range S = 1..ftoi(round(2^n));
{int} Sub[s in S] = {i | i in Nodes:
  (s div ftoi(round(2^(i-1)))) mod 2 == 1 };
{int} Compl[s in S] = NodeSet diff Sub[s];

float Cost[Edges] = ...;

dvar boolean x[Edges];

constraint ctCut[S];

minimize
    sum(e in Edges) Cost[e] * x[e];

subject to {
  forall(s in S : 0 < card(Subset[s]) < n)
  ctCut[s]:
    sum(i in Sub[s], j in Nbs[i] inter Compl[s]) x[<i,j>]
    + sum(i in Compl[s], j in Nbs[i] inter Sub[s]) x[<i,j>]
        >= 1;

  ctAll:
    sum(e in Edges) x[e] == n-1;
}

{Edge} Used = {e| e in Edges : x[e] == 1};

execute
{
    writeln("Used edges=", Used);
}
```

The graph is represented as a list of edges where each edge is an ordered tuple

of two nodes. These are declared in line 1-10. For each node we record the set of neighbouring nodes in line 11. Lines 13-16 declares all possible subsets of nodes and the complement of these subsets. Since we have an ordered tuple for each edge we need to check for cutting edges in both directions when implmenting the cut constraints in lines 30-31.

The following is data for an example with 12 nodes:

Listing 9: mst.mod

```
1  n = 12;
2
3  Edges = {<1,3>, <2,3>, <3,4>, <1,5>, <2,6>, <1,8>, <7,8>,
4           <3,5>, <9,10>, <3,9>, <10,12>, <11,12>, <3,12>,
5                    <4,5>, <5,11>, <1,2>, <5,8>,<3,11>};
6
7  Cost = [5  1  3  1  2  4  2  3  2  5  1  6  2  2  1  1  2  1];
```