Lecture notes 3: Solving linear and integer programs using the GNU linear programming kit

Vincent Conitzer

In this set of lecture notes, we will study how to solve linear and integer programs using standard solvers. Specifically, we will use the GNU linear programming kit (GLPK), which is available free of charge. Other solvers (including commercial solvers) can be used similarly.

The key issue we will study here is how to *represent* a linear or integer program, that is, what *language* (or format) we use to express the programs. We will study two different languages. The first is straightforward. The second one is a modeling language that allows us to represent a linear or integer program in an *abstract* form first—that is, using abstract sets and symbols for the parameters, as we have done previously—after which we can instantiate the program by providing concrete sets and concrete values for the parameters. Using the modeling language feels much more like programming, and it has the advantage that it is very easy to modify the particular instance of the problem. An alternative approach to using the modeling language is to write some code in a programming language such as C or Java that automatically generates the linear or integer program (in the basic language) based on some input. This approach can be useful, especially in the context of a bigger project; but often the modeling language suffices, and it is much easier and faster to use. Linear and integer programs written in the modeling language, with concrete instantiations of the sets and parameters, can easily be converted to linear programs in the basic language.

1 The basic (`.lp`) language

In the basic language, the integer program for the painting problem instance can be represented as follows:

```
maximize
3x1 + 2x2
subject to
4x1 + 2x2 < 15
x1 + 2x2 < 8
x1 + x2 < 5
bounds
x1>0
x2>0
integer
x1
x2
end
```

Note the use of strict inequalities instead of weak inequalities, even though they are in fact still interpreted as weak inequalities. If integrality is not required, then the lines

```
integer
x1
x2
```

can be dropped.

If all of this is placed into a file called `painting.lp`, we can solve it by entering the command:
`glpsol --cpxlp painting.lp -o painting.out`
This places the output in a file called `painting.out`. The resulting output is the following:

```
Problem:
Rows:       3
Columns:    2 (2 integer, 0 binary)
Non-zeros:  6
Status:     INTEGER OPTIMAL
Objective:  obj = 12 (MAXimum) 12.5 (LP)

   No.   Row name        Activity     Lower bound   Upper bound
------ ------------    ------------- ------------- -------------
     1 r.4                        14                          15
     2 r.5                         8                           8
     3 r.6                         5                           5

   No. Column name       Activity     Lower bound   Upper bound
------ ------------    ------------- ------------- -------------
     1 x1           *              2             0
     2 x2           *              3             0

Integer feasibility conditions:

INT.PE: max.abs.err. = 0.00e+00 on row 0
        max.rel.err. = 0.00e+00 on row 0
        High quality

INT.PB: max.abs.err. = 0.00e+00 on row 0
        max.rel.err. = 0.00e+00 on row 0
        High quality

End of output
```

The optimal solution to the problem instance can be read off from the column activities (2 and 3), and the objective value is given higher up (12). From the row activities we can also see how much of each paint color is being used.

`Minimize` can also be used instead of `Maximize`.

2 The modeling (`.mod`) language

We now arrive at the more interesting modeling language. The easiest way to learn how to use this language is to look at some example programs. Here is the painting problem again:

```
set PAINTINGS;
set COLORS;

var quantity_produced{j in PAINTINGS}, >=0, integer;

param selling_price{j in PAINTINGS};
param paint_available{i in COLORS};
param paint_needed{i in COLORS, j in PAINTINGS};

maximize revenue: sum{j in PAINTINGS} selling_price[j]*quantity_produced[j];

s.t. enough_paint{i in COLORS}: sum{j in PAINTINGS}
paint_needed[i,j]*quantity_produced[j] <= paint_available[i];

data;

set PAINTINGS := p1 p2;
set COLORS := blue green red;

param selling_price := p1 3 p2 2;
param paint_available := blue 15 green 8 red 5;

param paint_needed :
       p1   p2 :=
blue    4    2
green   1    2
red     1    1;

end;
```

Note how we first give the abstract form of the program, then (preceded by `data;`) we give the specific instantiation. Naturally, one needs to be careful to use brackets, semicolons, *etc.* correctly.

We first specify two abstract sets, `PAINTINGS` and `COLORS`. These are later instantiated to `p1 p2` and `blue green red`, but we could have instantiated them in other ways as well, for example `night_watch starry_night` and `cyan magenta yellow black`. Such a change does not require any change above the `data;` line.

We then specify the variables of the problem in the line `var quantity_produced{j in PAINTINGS}, >=0, integer;` This means that there will be one nonnegative integer `quantity_produced` variable for every painting (indicating how many reproductions of that painting we make). If integrality is not required, then `, integer` can be dropped.

Next, we specify the parameters of the problem. For example, the line `param paint_needed{i in COLORS, j in PAINTINGS};` means that there will be one `paint_needed` parameter for every combination of a color and a painting. Note that the modeling language makes the distinction between variables and parameters explicit.

The objective is next. We have to give a name to the objective, in this case `revenue`. We then sum the product of `selling_price` and `quantity_produced` over all paintings. Of course, we are only allowed to take this product because `selling_price` is a parameter and not a variable; had it been a variable, then this would not have been linear.

The constraints are next. `s.t.` is short for `subject to`. We also have to give names to the constraints. `enough_paint{i in COLORS}` means that there is one `enough_paint` constraint for

every color (and `i` is that color in the constraint).

This completely specifies the problem in the abstract. To solve it, we need a concrete instantiation of this problem (that is, the sets and parameters). The line `data;` indicates that we will specify the data next. The data part is fairly self-explanatory. Note how we can nicely specify a 2-dimensional parameter such as `paint_needed` in matrix form.

If all of this is placed in a file called `painting.mod`, then we can solve it with
`glpsol --math painting.mod -o painting.out`
After this, `painting.out` will contain:

```
Problem:    painting
Rows:       4
Columns:    2 (2 integer, 0 binary)
Non-zeros:  8
Status:     INTEGER OPTIMAL
Objective:  revenue = 12 (MAXimum) 12.5 (LP)

   No.   Row name        Activity     Lower bound   Upper bound
------ ------------    ------------- ------------- -------------
     1 revenue                    12
     2 enough_paint[blue]
                                  14                           15
     3 enough_paint[green]
                                   8                            8
     4 enough_paint[red]
                                   5                            5

   No. Column name       Activity     Lower bound   Upper bound
------ ------------    ------------- ------------- -------------
     1 quantity_produced[p1]
                   *               2             0
     2 quantity_produced[p2]
                   *               3             0

Integer feasibility conditions:

INT.PE: max.abs.err. = 0.00e+00 on row 0
        max.rel.err. = 0.00e+00 on row 0
        High quality

INT.PB: max.abs.err. = 0.00e+00 on row 0
        max.rel.err. = 0.00e+00 on row 0
        High quality

End of output
```

Incidentally, a problem instance expressed in the modeling language can be converted to a linear program in the basic language using the command:
`glpsol --check --math painting.mod --wcpxlp painting.lp`
(The `--check` means that the solver is not actually made to solve the instance.) This produces the following `painting.lp` file:

```
\* Problem: painting *\

Maximize
 revenue: + 3 quantity_produced(p1) + 2 quantity_produced(p2)

Subject To
 enough_paint(blue): + 4 quantity_produced(p1) + 2 quantity_produced(p2)
 <= 15
 enough_paint(green): + quantity_produced(p1) + 2 quantity_produced(p2)
 <= 8
 enough_paint(red): + quantity_produced(p1) + quantity_produced(p2) <= 5

Generals
 quantity_produced(p1)
 quantity_produced(p2)

End
```

2.1 Markov decision processes

Below is the Markov decision process problem from before in the modeling language. (This also illustrates how to deal with a 3-dimensional parameter, in this case the transition probabilities.)

```
set STATES;
set ACTIONS;

var value{s in STATES};
param transition_probability{s in STATES, a in ACTIONS, s2 in STATES};
param reward{s in STATES, a in ACTIONS};
param discount_factor;

minimize total: sum{s in STATES} value[s];

s.t. bellman{s in STATES, a in ACTIONS}: value[s] >= reward[s,a] + sum{s2
in STATES} discount_factor*transition_probability[s,a,s2]*value[s2];

data;

set STATES := good deteriorating broken;
set ACTIONS := maintain ignore;

param transition_probability:=

[*,*,good]:
        maintain ignore :=
good           1     .5
deteriorating .9      0
broken        .2      0

[*,*,deteriorating]:
        maintain ignore :=
good           0     .5
deteriorating .1     .5
broken         0      0

[*,*,broken]:
        maintain ignore :=
good           0      0
deteriorating  0     .5
broken        .8      1;

param reward:
        maintain ignore :=
good           1      2
deteriorating  1      2
broken        -1      0;

param discount_factor := .9;

end;
```

The output produced is:

```
Problem:    mdp
Rows:       7
Columns:    3
Non-zeros:  13
Status:     OPTIMAL
Objective:  total = 39.80567227 (MINimum)
```

No.	Row name	St	Activity	Lower bound	Upper bound	Marginal
1	total	B	39.8057			
2	bellman[good,maintain]	B	1.66912	1		
3	bellman[good,ignore]	NL	2	2		16.9485
4	bellman[deteriorating,maintain]	NL	1	1		9.48004
5	bellman[deteriorating,ignore]	B	5.55436	2		
6	bellman[broken,maintain]	NL	-1	-1		3.57143
7	bellman[broken,ignore]	B	0.715861	-0		

No.	Column name	St	Activity	Lower bound	Upper bound	Marginal
1	value[good]	B	16.6912			
2	value[deteriorating]	B	15.9559			
3	value[broken]	B	7.15861			

```
Karush-Kuhn-Tucker optimality conditions:

KKT.PE: max.abs.err. = 3.55e-15 on row 1
        max.rel.err. = 9.50e-16 on row 4
        High quality

KKT.PB: max.abs.err. = 0.00e+00 on row 0
        max.rel.err. = 0.00e+00 on row 0
        High quality

KKT.DE: max.abs.err. = 1.47e-15 on column 2
        max.rel.err. = 7.33e-16 on column 2
        High quality

KKT.DB: max.abs.err. = 0.00e+00 on row 0
        max.rel.err. = 0.00e+00 on row 0
        High quality
```

```
End of output
```

From this output, we can read off the values of the states, which is already enough to determine the optimal policy. In fact, we can read off the optimal policy directly: the optimal action to take in each state is the action for which there is no slack in the constraint. The constraints for which this is true are `bellman[good,ignore]`, `bellman[deteriorating,maintain]`, `bellman[broken,maintain]`, so we should maintain the machine unless it is in good shape.