Ukulelelala

Author: Aster Santana
*July 2020*

This is the documentation of the Ukulelelala use case, which is part of the Learning Mip project maintained by Mip Master.

Concepts

- ➤ Modeling with integer decision variables
- ➤ Summation notation
- ➤ If-then constraints
- ➤ Big-M constraints
- ➤ Complement of a binary variable
- ➤ LP files

Problem statement

Ms. Mip's friend, Ted, is running a business on the music industry, a ukulele factory. Ted's brand, *Ukulelelala* (which we can pronounce singing: Ukulele-la♫-la♫), is becoming a big success because of the great quality of the ukulele he produces and also because of a recent marketing investment that Ted made.

Every week, Ted ships all his production to seven major partner retailers. But, recently, the demand has been increasing tremendously to the point that Ted has not been able to keep up with the production. Since Ted is not allowed to increase prices due to contractual reasons, every week, he must decide how much to fulfil of each order he receives.

The table shows the data from last week, when the factory was able to produce and ship 650 ukuleles, 510 below the total demand that was 1,160.

Retailer ID	Wholesale Unit Price ($)	Order Qty.	Shipped Qty.
R01	47.00	230	50
R02	65.00	150	135
R03	70.00	270	270
R04	68.00	90	90
R05	46.00	190	0
R06	78.00	55	55
R07	55.00	120	50

Mr. Mip happily stepped in to help his friend by writing a program that used MIP to arrive at the numbers you see on the Shipped Qty. column.

At first, these number might sound unreasonable. You would probably expect to see the order from R02 being met in whole and the order from R01 not being fulfilled at all. That would be the case if the only goal were to maximize profit. In which case, Mr. Mip could have simply sorted the retailers by selling price and fulfilled first the orders with higher prices.

However, Ted has a contract signed with each retailer. And one of the agreements in the contract is that Ted incurs a fee of 20 times the wholesale price of a ukulele if he can't ship at least 50 ukuleles to a retailer that placed an order for that week.

Let's see how Mr. Mip solved Ted's problem using MIP.

Formulation

Let's go through the three main steps that Mr. Mip typically take to write a formulation.

Decision variables

Mr. Mip's trick to define decision variables is to think about the *unknowns* of the problem. For example, the number of ukuleles available for shipment is known, 650 in this case. Likewise, the price by which he will sell to each retailer is also known, it's given in the Wholesale Unit Price column.

Which retailers should Ted ship to? That's unknown, it needs to be decided. Once Ted decide to ship to a retailer, how many units should he ship? This is also to be determined. Following this reasoning, Mr. Mip defined two set of decision variables.

> *Decision variables:*
>
> x_i equal 1 if Ted ships to retailer i, 0 otherwise, for all $i = R01, R02, ..., R07$.
>
> y_i the number of ukuleles shipped to retailer i, for all $i = R01, R02, ..., R07$.

Notice that x are binary variables (they can only take 0-1 values) while y are integer variables. It wouldn't make sense to define y as continuous since it doesn't make sense to ship 50.3 ukuleles, for instance.

To simplify notation, let's write $x_1, x_2, ..., x_7$ instead of $x_{R01}, x_{R02}, ..., x_{R07}$ when writing constraints and the objective function. Similarly, let's write $y_1, y_2, ..., y_7$ instead of $y_{R01}, y_{R02}, ..., y_{R07}$.

Constraints

Ideally, Ted would meet all order in full—because the more units he ships, the more profit he makes (we will model this with the objective function). This means that the optimization should try to increase each variable y_i as much as possible.

However, Ted doesn't have enough supply. Therefore, Mr. Mip needs to model the fact that the total shipped quantity cannot exceed 650 units. He accomplished this with the constraint.

> Constraints – Max availability:

$$y_1 + y_2 + y_3 + y_4 + y_5 + y_6 + y_7 \leq 650.$$

> Using summation notation, this is the same as the following:

$$\sum_{i=1}^{7} y_i \leq 650.$$

Next, there is the fact that, if Ted ships less than 50 ukuleles to a retailer, then he must pay a penalty worthy of 20 units' price. Mr. Mip asked Ted whether he would consider shipping less than 50 units, but Ted said no.

So, for example, either Ted ships at least 50 units to retailer R01, or he doesn't ship any unit and pay a penalty of $20 * 47 = \$940$. Mr. Mip has a smart and very useful trick to model this type of requirement.

Constraints – If ship, then ship at least 50:

$$50x_1 \leq y_1.$$

This inequality says that, **if** $x_1 = 1$, meaning that Ted ships at least one unit to retailer $R01$, **then** y_1 must be at least 50, meaning that Ted must ship at least 50 units to that retailer. Of course, we need one such constraint for every retailer. We see them all in the final formulation.

This is an **"if then"** type of constraint, which you should keep in your notes. Mr. Mip uses this trick very often and I'm sure that you will use it too. Here is a more generic form for reference:

Statement: If $x > 0$, then $y \geq c$.

Formulation: $cx \leq y$.

Assumptions: x is a binary and c is a constant.

But what if $x_1 = 0$? Well, in that case, y_1 can take any nonnegative value. But that's not what Mr. Mip wants. If $x_1 = 0$, meaning that Ted doesn't ship any unit to retailer $R01$, then y_1 <u>must</u> be zero as well. Though this is obvious from the definition of x and y, the optimization (or the computer, if you want to think this way) may not obey this rule unless Mr. Mip explicitly state it in the form of a constraint. So how would you model "if $x_1 = 0$, then y_1 must be zero"?

The first thing you might think is the following constraint:

$$y_1 \leq x_1.$$

But there is a problem with this constraint. It does what we want but it also does something we don't want. Do you see what that is?

When $x_1 = 0$, this constraint forces y_1 to be zero, which is what we want. But when $x_1 = 1$, this constraint restricts y_1 to be less than or equal to one, which is inacceptable. Here is how Mr. Mip models this.

Constraints – If doesn't ship, then doesn't ship:

$$y_1 \leq 230 * x_1.$$

Notice that 230 is the order placed by retailer $R01$. When $x_1 = 0$, this constraint still forces y_1 to be zero. And when $x_1 = 1$, this constraint restricts y_1 to be less than or equal to 230, which is also something we want. After all, Ted is not supposed to ship more than the order quantity. So, this constraint models two conditions at once. Again, we will need one such constraint for each retailer. You will see them all in the final formulation.

This type of constraint where a binary variable, x_1 in this case, is used to "deactivate" another variable, y_1 in this case, is itself a "if then" constraint. But it's a very common one and typically called **big-M constraint**.

Here's a more generic version for reference:

> Statement: If $x = 0$, then $y = 0$.
>
> Formulation: $y \leq Mx$.
>
> Assumptions: x is a binary, y is non-negative, and M is a large enough constant.

In most cases, an arbitrarily large M yields a correct formulation. However, it's strongly recommended to choose M as small as possible, since too large M can significantly compromise the performance of the solver.

Objective

The objective of this problem is to maximize total profit, which is total revenue minus penalty.

To formulate the revenue from retailer $R01$, simply multiplied y_1, the shipment quantity, by the unit selling price of retailer $R01$:

$$rvn_1 = 47\, y_1.$$

Formulating the total penalty is a bit trickier though. Mr. Mip formulated the penalty paid to retailer $R01$ as follows:

$$pnt_1 = (20 * 47)(1 - x_1).$$

Notice that it does exactly what you would expect. If $x_1 = 1$, meaning that Ted ships to retailer $R01$, then the quantity in the second brackets is zero and, therefore, $pnt_1 = 0$.

On the other hand, if $x_1 = 0$, meaning that Ted does not ship to retailer $R01$, then the quantity in the second brackets is one, in which case $pnt_1 = 20 * 47$, precisely the penalty that Ted is supposed to pay in this case.

To understand Mr. Mip's trick, you can think of the binary variable x_1 as an on/off switch. Then you think of $(1 - x_1)$ as a switch that does exactly the opposite of what x_1 does.
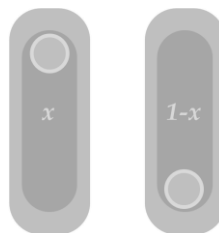
In other words, $(1 - x_1)$ is the complement of x_1.

Since the total revenue is the sum of each retailer revenue and the total penalty is the sum of each retailer penalty, Mr. Mip formulated the objective as follows.

Objective:

$$\max(revenue - penatlty),$$

where

$$revenue = \sum_{i=1}^{7} p_i\, y_i$$

$$penalty = \sum_{i=1}^{7} (25 * p_i)\, (1 - x_i),$$

where p_i is the wholesale price by which Ted sells to retailer $i = 1, 2, \dots, 7$.

Final formulation

Putting all together, Mr. Mip arrived at the following formulation.

Final formulation:

$$\max \sum_{i=1}^{7} p_i\, y_i - \sum_{i=1}^{7} (20 * p_i)\, (1 - x_i)$$

$$s.t. \sum_{i=1}^{7} y_i \leq 650$$

$$50x_i \leq y_i, \qquad\qquad i = 1, 2, \dots, 7$$
$$y_i \leq d_i\, x_i, \qquad\qquad i = 1, 2, \dots, 7$$
$$x_i \in \{0, 1\}, \;\; y_i \in \mathbb{Z}_{+}, \qquad i = 1, 2, \dots, 7,$$

where p_i is the wholesale price and d_i is the demand of retailer $i = 1, 2, \dots, 7$.

Implementation and optimization

Below is the implementation of the formulation using *gurobipy*. The code is available in the Mip Master repository on GitHub [link].

```
1  import gurobipy as gp
2
3  # Input Data
4  # retailers
5  I = [1, 2, 3, 4, 5, 6, 7]
6  # price
7  p = {1: 47, 2: 65, 3: 70, 4: 68, 5: 46, 6: 78, 7: 55}
8  # demand
9  d = {1: 230, 2: 150, 3: 270, 4: 90, 5: 190, 6: 55, 7: 120}
10 # production upper bound
11 PU = 650
12 # shipment lower bound
13 SL = 50
14 # penalty (num. of units)
15 PN = 20
16
17 # Define the model
18 mdl = gp.Model('Ukulele-la-la')
19
20 # Add variables
21 x = mdl.addVars(I, vtype=gp.GRB.BINARY, name='x')
22 y = mdl.addVars(I, vtype=gp.GRB.INTEGER, name='y')
23
24 # Add Constraints
25 mdl.addConstr(sum(y[i] for i in I) <= PU, name='max_avty')
26 mdl.addConstrs((SL * x[i] <= y[i] for i in I), name='at_least')
27 mdl.addConstrs((y[i] <= d[i] * x[i] for i in I), name='at_most')
28
29 # Set the objective function
30 revenue = sum(p[i] * y[i] for i in I)
31 penalty = sum((PN * p[i]) * (1 - x[i]) for i in I)
32 mdl.setObjective(revenue - penalty, sense=gp.GRB.MAXIMIZE)
33
34 # Optimize
35 mdl.optimize()
36
37 # Retrieve the solution
38 y_sol = {i: int(y[i].X) for i in I}
39 print(f'y = {y_sol}')
40 print(f'revenue {revenue.getValue()}')
41 print(f'penalty {penalty.getValue()}')
```

Notice that all the data is define outside of the optimization model. This way Mr. Mip will not have to change the model when Ted send him the data for next week. In fact, once Ted has validated the solution, Mr. Mip will modify the code to read the data from a file that Ted can update without touching the code.

The output of this code is the following:

```
y = {1: 50, 2: 135, 3: 270, 4: 90, 5: 0, 6: 55, 7: 50}
revenue  43185.0
penalty  920.0
```

From this Mr. Mip populated the Shipped Qty. column of the table above.

Challenge yourself

1) When Mr. Mip showed the solution from his program, Ted didn't like that $R07$ was going to receive only 50 ukuleles because $R07$ was a new client and he would fulfil at least 80% of their order. How can you modify/extend Mr. Mip's formulation to address this new requirement? Implement the new model to see that the total revenue drops to $42,725.

2) Ted later realized that he should also consider transportation cost. Specifically, there is a fixed cost of shipping to each retailer as follows:

   $R01: 85.00, R02: 125.00, R03: 92.00, R04: 137.00, R05: 100.00, R06: 60.00, R07: 77.00.$

   How can you further modify/extend the formulation to address this new requirement? Implement and solve the new model to see that the total revenue increases to $43,625 and the total penalty increases to $1,860.

3) Add the command `mdl.write('ukulelelala.lp')` to your code, anywhere after `mdl.optimize()`, to generate a text file that has a human readable version of your model. The file will be saved to the same directory of your script. This file is called the *LP File* of the model, and it's very useful to debug the implementation. In addition, solvers can read and solve the model from an LP file. Let say you have two MIP solvers and you want to see which one solves you model faster. Then you can implement the model with one solver, generate the LP file, and solve the model with the other solver without having to implement the model again.

Takeaways

1. Real-world problems can have unexpected requirements, such as shipping at least 50 of nothing.
2. To define decision variables, we need to identify the unknowns of the problem. Ask yourself what are the decisions I would have to make to solve this problem? In this example, you would have to ask: What retailers should Ted ship to? If he decides to ship to a retailer, how many units should he ship?
3. A MIP formulation can have multiple set of decision variables, and of different types. This example was formulated with a set of binary and a set integer variables.
4. If-then constraints are useful and can assume a variety of formats.
5. When using Big-M constraint, which is especial case of if-then constraints, don't get fooled by the name "big". In fact, the smallest the M the better.
6. Every binary variable, let say $x \in \{0, 1\}$, has a complement, $x - 1$. So, when x is "on", $x - 1$ is "off". And when x is "on", $x - 1$ is "off".
7. When a requirement needs to be formulated for several items of the same type, like minimum shipments quantity for seven retailers in this example, it's usually easier to do so for one item first and then generalize to all the others.
8. The LP file is a good tool to check if your implementation is correct and to export the model in a format that other solvers can understand.
9. When implementing the formulation, it's a good practice to keep all the data separated from the optimization model.
10. It's easy to address new requirements by extending an existing formulation. This is a great feature of MIP that you won't find in other approaches such as heuristics and dynamic programming.