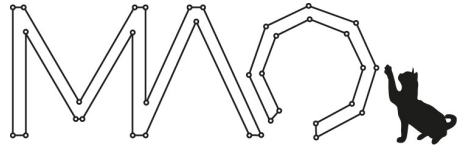
UNIVERSITÀ
DEGLI STUDI
DI BRESCIA

Introduction to Gurobi

Prof. **Renata Mansini**

Research Group MAO@DII



Models and Algorithms for Optimization at Department of Information Engineering

Website: `https://or-dii.unibs.it/`

INTRODUCTION

Overview

Gurobi Optimizer:

- Commercial software, written in C, dedicated to the optimization of mathematical problems of different types (LP, MILP, MIQP);
- Widely used in industry and academia;
- High performances and customization;
- Designed to exploit multi-core processors out of the box.

Installing Gurobi

- Go to `www.gurobi.com` and create an account (using your academic email);
- Download and install the Gurobi Optimizer (64-bit version);
- Go to your license page in the website and, from a command/terminal prompt, execute the command you'll find there

  grbgetkey "license-code"

  N.B. You have to be connected to the University network in order to activate your license.
- Include in the project the gurobi.jar library you'll find at

  ...\gurobi912\win64\lib

The example

We will use a simple Java example in order to present the main features of the Gurobi Java Interface.

In this example we will:

- build a model;
- optimize it;
- query some information about the model and the solution.

The example

We will use a simple Java example in order to present the main features of the Gurobi Java Interface.

In this example we will:

- build a model;
- optimize it;
- query some information about the model and the solution.

We will optimize the following model:

$$
\begin{aligned}
\textbf{Maximize} \quad & x + y + 2z \\
\textbf{subject to} \quad & x + 2y + 3z \leq 4 \\
& x + y \geq 1 \\
& x, y, z \quad \textit{binary}
\end{aligned}
$$

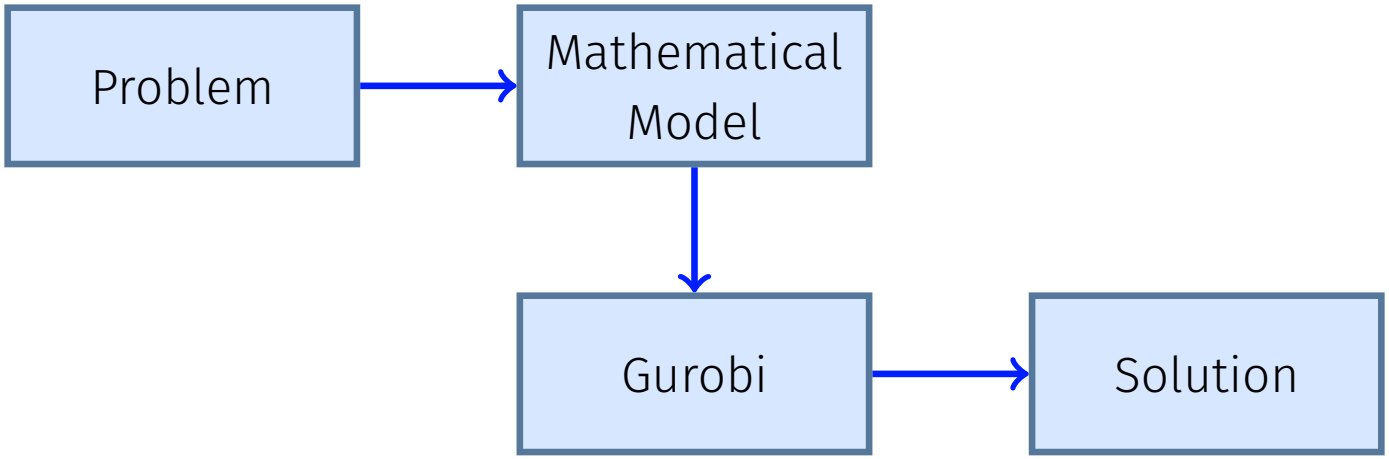
The example

You can find the full example in the elearning course pages:

mipExample.java

BUILDING THE MODEL

Gurobi as a black-box solver



Solution: A value is assigned to each one of the problem variables.

The solution might not be optimal:

- A solution does not exist;
- Gurobi requires too much time and\or memory.

Gurobi provides a lot more information about the solution itself and the solution process.

The environment

Our example begins by importing the Gurobi classes

```
import gurobi.*;
```

The environment

Our example begins by importing the Gurobi classes

```
import gurobi.*;
```

Then we instantiate a GRBEnv, the environment

```
GRBEnv env = new GRBEnv("mipExample.log");
```

The constructor argument specifies the name of the log file.

To create and optimize a model we will always need an environment.

The environment

Our example begins by importing the Gurobi classes

```
import gurobi.*;
```

Then we instantiate a GRBEnv, the environment

```
GRBEnv env = new GRBEnv("mipExample.log");
```

The constructor argument specifies the name of the log file.

To create and optimize a model we will always need an environment.

It is sometimes useful to change some environment parameters:

```
env.set(GRB.IntParam.Threads, 4);
env.set(GRB.IntParam.Presolve, 2);
env.set(GRB.DoubleParam.MIPGap, 1e-4);
env.set(GRB.DoubleParam.TimeLimit, 600);
```

Environment parameters

It is sometimes useful to change some environment parameters:

```
env.set(GRB.IntParam.Threads, 4);
```

The Threads parameter determine how many threads/cores Gurobi will use while optimizing a model in this environment.

Useful when multiple models from different environments are solved simultaneously.

Default value is 0 (all cores used).

Environment parameters

It is sometimes useful to change some environment parameters:

```
env.set(GRB.IntParam.Presolve, 2);
```

The Presolve parameter controls the presolve level. -1 corresponds to an automatic setting. Other options are off (0), conservative (1), or aggressive (2).

A more aggressive usage of presolve takes more time, but can sometimes lead to a model significantly faster to solve.

-1 is the default value.

Environment parameters

It is sometimes useful to change some environment parameters:

```
env.set(GRB.DoubleParam.MIPGap, 1e-4);
```

$$MIPGap = \frac{|Incumbent - BestBound|}{|Incumbent|}$$

If $MIPGap < 10^{-4}$, the optimization stops and returns the Incumbent solution as the optimal solution.

To avoid loosing potentially optimal solutions, it is better to set the MIPGap parameter to a sufficiently low value lower than

$$\frac{1}{LP\_Solution\_Value}$$

Environment parameters

It is sometimes useful to change some environment parameters:

```
env.set(GRB.DoubleParam.TimeLimit, 600);
```

The TimeLimit parameter limits the total time spent finding an optimal solution for a model.

It is expressed in seconds.

The model

Once we have an environment, we can create a model.

```
GRBModel model = new GRBModel(env);
```

A Gurobi Model holds a single optimization problem. It consists of a set of variables, a set of constraints, an objective function and all their attributes.

With the above code we create an empty model object and we will later add all the required information.

Adding variables

The next step is to add variables to the model.

```
GRBVar x = model.addVar(0.0, 1.0, 0.0, GRB.BINARY, "x");
GRBVar y = model.addVar(0.0, 1.0, 0.0, GRB.BINARY, "y");
GRBVar z = model.addVar(0.0, 1.0, 0.0, GRB.BINARY, "z");
```

Variables are added through the addVar() method on a model object.
A variable is always associated with a particular model.

Adding variables

The next step is to add variables to the model.

```
GRBVar x = model.addVar(0.0, 1.0, 0.0, GRB.BINARY, "x");
GRBVar y = model.addVar(0.0, 1.0, 0.0, GRB.BINARY, "y");
GRBVar z = model.addVar(0.0, 1.0, 0.0, GRB.BINARY, "z");
```

The first argument to the *addVar*() call is the lower bound of the variable.

Adding variables

The next step is to add variables to the model.

```
GRBVar x = model.addVar(0.0, 1.0, 0.0, GRB.BINARY, "x");
GRBVar y = model.addVar(0.0, 1.0, 0.0, GRB.BINARY, "y");
GRBVar z = model.addVar(0.0, 1.0, 0.0, GRB.BINARY, "z");
```

The first argument to the *addVar*() call is the lower bound of the variable.

The second argument is the upper bound of the variable.

Adding variables

The next step is to add variables to the model.

```
GRBVar x = model.addVar(0.0, 1.0, 0.0, GRB.BINARY, "x");
GRBVar y = model.addVar(0.0, 1.0, 0.0, GRB.BINARY, "y");
GRBVar z = model.addVar(0.0, 1.0, 0.0, GRB.BINARY, "z");
```

The first argument of the *addVar*() call is the lower bound of the variable.

The second argument is the upper bound of the variable.

The third argument is the linear objective coefficient of the variable. It is zero here since we will set the objective function later.

Adding variables

The next step is to add variables to the model.

```
GRBVar x = model.addVar(0.0, 1.0, 0.0, GRB.BINARY, "x");
GRBVar y = model.addVar(0.0, 1.0, 0.0, GRB.BINARY, "y");
GRBVar z = model.addVar(0.0, 1.0, 0.0, GRB.BINARY, "z");
```

The fourth argument is the variable type. There are three main possibilities:

- GRB.BINARY
- GRB.INTEGER
- GRB.CONTINUOUS

Adding variables

The next step is to add variables to the model.

```
GRBVar x = model.addVar(0.0, 1.0, 0.0, GRB.BINARY, "x");
GRBVar y = model.addVar(0.0, 1.0, 0.0, GRB.BINARY, "y");
GRBVar z = model.addVar(0.0, 1.0, 0.0, GRB.BINARY, "z");
```

The fourth argument is the variable type. There are three possibilities:

- GRB.BINARY
- GRB.INTEGER
- GRB.CONTINUOUS

The last argument is the name of the variable.

Lazy updates

Model modifications in the Gurobi optimizer are done in a *lazy* fashion.

Model changes are not seen immediately, since they have to be integrated manually.

Changes are integrated by the *update* method:

```
model.update();
```

N.B. Since the 7.0 version, the update call is no longer necessary in *most cases*.

Objective function

The next step is to set the optimization objective:

```
//Set objective: maximize x + y + 2 z

GRBLinExpr expr = new GRBLinExpr();
expr.addTerm(1.0, x);
expr.addTerm(1.0, y);
expr.addTerm(2.0, z);
model.setObjective(expr, GRB.MAXIMIZE);
```

Objective function

The next step is to set the optimization objective:

```
//Set objective: maximize x + y + 2 z

GRBLinExpr expr = new GRBLinExpr();
expr.addTerm(1.0, x);
expr.addTerm(1.0, y);
expr.addTerm(2.0, z);
model.setObjective(expr, GRB.MAXIMIZE);
```

Firstly, we construct an empty linear expression.

Then we add three terms to the expression using the *addTerm* method. The first argument is the coefficient that multiplies the variable, which is the second argument.

Objective function

The next step is to set the optimization objective:

```
//Set objective: maximize x + y + 2 z

GRBLinExpr expr = new GRBLinExpr();
expr.addTerm(1.0, x);
expr.addTerm(1.0, y);
expr.addTerm(2.0, z);
model.setObjective(expr, GRB.MAXIMIZE);
```

Finally we set the objective through the *setObjective* method.

The second argument of *setObjective* indicates that the optimization sense is maximization.

The constraints

We now add the constraints to the model.

```
// Add constraint: x + 2 y + 3 z <= 4
expr = new GRBLinExpr();
expr.addTerm(1.0, x);
expr.addTerm(2.0, y);
expr.addTerm(3.0, z);
model.addConstr(expr, GRB.LESS_EQUAL, 4.0, "c0");
```

The constraints

We now add the constraints to the model.

```
// Add constraint: x + 2 y + 3 z <= 4
expr = new GRBLinExpr();
expr.addTerm(1.0, x);
expr.addTerm(2.0, y);
expr.addTerm(3.0, z);
model.addConstr(expr, GRB.LESS_EQUAL, 4.0, "c0");
```

As we have seen for the variables, constraints are always associated with a model. Constraints are added to the model using the *addConstr* method.

The first argument of *addConstr()* is the left-hand side of the constraint, which we built by first creating an empty linear expression object, and then adding three terms to it.

The constraints

We now add the constraints to the model.

```
// Add constraint: x + 2 y + 3 z <= 4
expr = new GRBLinExpr();
expr.addTerm(1.0, x);
expr.addTerm(2.0, y);
expr.addTerm(3.0, z);
model.addConstr(expr, GRB.LESS_EQUAL, 4.0, "c0");
```

The second argument is the constraint sense (GRB_LESS_EQUAL, GRB_GREATER_EQUAL, or GRB_EQUAL)

The third argument is the right-hand side (a constant in our example).

The final argument is the constraint name.

The constraints

The second constraint is created in a similar manner.

```
// Add constraint: x + y >= 1
expr = new GRBLinExpr();
expr.addTerm(1.0, x);
expr.addTerm(1.0, y);
model.addConstr(expr, GRB.GREATER_EQUAL, 1.0, "c1");
```

Other methods

Some useful methods

```
expr.addConstant(5);
```

The method *addConstant* adds a constant term to the expression.

Other methods

Some useful methods

```
expr.addConstant(5);
```

The method *addConstant* adds a constant term to the expression.

```
model.getVarByName("x");
```

The method *getVarByName* returns the variable added to the model with the specified name.

OPTIMIZING & RESULTS

Optimization

Now that the model has been built, the next step is to optimize it:

```
model.optimize();
```

This method performs the optimization of the model and populates several internal model attributes.

The *Status* attribute of the model indicates the state of the optimization process when it ended.

Some of the most common values are:

- GRB.OPTIMAL
- GRB.INFEASIBLE
- GRB.TIME_LIMIT

Reporting results

Once the optimization is complete, we can query the values of the attributes.

```
x.get(GRB.StringAttr.VarName);
x.get(GRB.DoubleAttr.X);
model.get(GRB.DoubleAttr.ObjVal);
```

Attribute *VarName* contains the name of the variable.

Attribute *X* contains the solution value of the variable.

The model attribute *ObjVal* contains the objective value for the current solution.

Continuous relaxation

It is often useful to solve the continuous relaxation of a model:

```
GRBModel relaxedModel = model.relax();
relaxedModel.optimize();
x.get(GRB.DoubleAttr.RC);
```

To solve the continuous relaxation of the current MIP Model, we first create the relaxed model, by calling the *relax* method of the original model object.

After the relaxation is solved, other attributes can be queried, for example the *RC* variable attribute, which contains the reduced cost of the variable.

Cleaning up

You need to get rid of the model and the environment once you don't need them anymore:

```
model.dispose();
env.dispose();
```

These methods free the resources associated with the model and the environment. Garbage collection would claim these resources eventually, but if your program does not exit immediately after performing the optimization, it is best to free them explicitly.

Let's see the example ⟹

The log

```
Academic license - for non-commercial use only
Read MPS format model from file C:\Users\Roberto\workspace\Progetto2019\instances\rococoC10-001000.mps
Reading time = 0.02 seconds
rococoC10-001000: 1293 rows, 3117 columns, 11751 nonzeros
Optimize a model with 1293 rows, 3117 columns and 11751 nonzeros
Coefficient statistics:
  Matrix range     [1e+00, 3e+04]
  Objective range  [1e+00, 1e+00]
  Bounds range     [1e+00, 3e+05]
  RHS range        [1e+00, 1e+00]
Presolve removed 750 rows and 675 columns
Presolve time: 0.00s
Presolved: 543 rows, 2442 columns, 6806 nonzeros

Iteration    Objective       Primal Inf.    Dual Inf.      Time
       0    0.0000000e+00   2.022254e+03   0.000000e+00      0s
    1494    7.5152710e+03   0.000000e+00   0.000000e+00      0s

Solved in 1494 iterations and 0.04 seconds
Optimal objective  7.515271029e+03
```

CALLBACKS

Callbacks (I)

Using Gurobi only as a black-box solver is not always the best option.

The user is allowed to interact with the solver even during the optimization process.

Callback: a user function that is called periodically by the Gurobi optimizer in order to allow the user to query or modify the state of the optimization.

Callbacks (II)

A callback is implemented in its own class, that extends the abstract class GRBCallback.

It is added to the model using the *setCallback* method

```
model.setCallback(new Callback(vars));
```

The callback logic is implemented in the *callback* method.

```
protected void callback()
```

Callbacks (III)

The callback class contains one protected int member variable: *where*
This variable can be queried to determine where the callback was called from.

It can take several values:

- *GRB.CB_POLLING*, a periodic callback;
- *GRB.CB_PRESOLVE*, a presolve callback;
- *GRB.CB_MIPSOL*, a MIP solution callback;
- *GRB.CB_MIPNODE*, a MIP node callback.

Callbacks (IV)

Inside a callback, a few methods are available:

- *getNodeRel(var)*, to get the current value of *var* (only when *where = GRB.CB_MIPNODE*)
- *getSolution(var)*, to get the current value of *var* (only when *where = GRB.CB_MIPSOL*)
- *addCut(expr, GRB.GREATER_EQUAL, expr)*, to add a constraint to the model, that does not cut off any feasible solution. (only when *where = GRB.CB_MIPNODE*)
- *addLazy(expr, GRB.GREATER_EQUAL, expr)*, to add a constraint to the model, that can cut off a feasible solution. (only when *where = GRB.CB_MIPNODE or GRB.CB_MIPSOL*)