From the model to Python/Gurobi

Outline

The basic gurobipy instructions

The four classes of variables

The model, one family per block

Reading the results

Tolerances

The relaxation with `relax()`

Three problems the course reuses

The course protocol, from start to finish

Running the code

Learning objectives (1/2)

At the end of this chapter you will be able to:

- write a model in `gurobipy`, one family of variables and constraints per block;
- declare the four classes of variables, remembering the default bounds;
- read the results in the right order: `Status`, `SolCount`, `ObjVal` and `ObjBound`;

Learning objectives (2/2)

At the end of this chapter you will be able to:

- use tolerances correctly when reading integer values;
- solve the LP relaxation with `relax()` and read its duals.

Prerequisites

- From this module, Chapter 1 «What is a MIP model»: the classes of variables and the structure of a model
- From this module, Chapter 2 «Relaxations, duality and bounds»: the LP relaxation, its dual and the solver attributes `ObjVal` and `ObjBound`

One solver only

- The course uses **one solver only**, Gurobi from Python
- This chapter shows how a model is written: one family of constraints per block, with the names of the mathematical model
- And above all how the results are *read*, including the case where the solver has not finished

The basic gurobipy instructions

Introduction: the basic gurobipy instructions

- A model is written in gurobipy with a handful of instructions, always the same ones
- From here on the line of code that writes it appears next to every model
- It pays to see them once on a small, complete model: the **binary knapsack**

The binary knapsack

- Out of n items of value p_j and weight w_j one chooses a subset of weight at most c worth as much as possible

$$\max \quad \sum_{j=1}^{n} p_j \, x_j$$

$$\text{subject to} \quad \sum_{j=1}^{n} w_j \, x_j \leq c,$$

$$x_j \in \{0,1\}, \quad \forall j \in \{1,2,\ldots,n\}.$$

The knapsack in gurobipy (1/2)

The modules and the data:

```
import gurobipy as gp       # the module, always abbreviated gp
from gurobipy import GRB    # the constants: GRB.BINARY, ...

p = [10, 7, 6, 4]           # values
w = [5, 4, 3, 3]            # weights
c = 9                       # capacity
n = len(p)
```

The knapsack in gurobipy (2/2)

The model, one instruction per line of the written model:

```
m = gp.Model("knapsack")                          # an empty model
x = m.addVars(n, vtype=GRB.BINARY, name="x") # n binaries
m.setObjective(gp.quicksum(p[j] * x[j] for j in range(n)),
               GRB.MAXIMIZE)
m.addConstr(gp.quicksum(w[j] * x[j] for j in range(n)) <= c,
            name="capacity")
m.optimize()
```

The instructions (1/3)

- `Model` creates the model; everything else hangs off it
- `addVars` adds a whole indexed family:
  - `addVars(n)` gives $x_0, \ldots, x_{n-1}$, indexed as `x[j]`
  - `addVars(n, k)` gives the matrix x_{ij}, indexed as `x[i, j]`
  - for a single variable there is `addVar`
- Python indices start at zero, written models start at one: that is the only difference to keep in mind

The instructions (2/3)

- `vtype` declares the domain:
  - `GRB.BINARY` for $x \in \{0, 1\}$
  - `GRB.INTEGER` for $x \in \mathbb{Z}_{\geq 0}$
  - without `vtype` the variable is continuous and non-negative
- The domain is a constraint: declaring it wrongly is declaring a different model
- `quicksum` builds a sum $\sum_j c_j x_j$: it does what `sum` does but without building intermediate expressions, and it is the one always used

The instructions (3/3)

- `setObjective` takes the expression and the direction, `GRB.MINIMIZE` or `GRB.MAXIMIZE`
- `addConstr` adds *one* constraint, `addConstrs` a whole family, and wants a generator in round brackets
  - one line of code per family of constraints, just as it is one line in the written model
- `optimize` solves

The `name`

- The `name` is of no use to the solver
- It is of use to whoever reads the exported model or an error message
- It is worth always matching it to the name of the constraint family in the written model

Bin packing with the same instructions

- n items of size w_j into bins of capacity c, using as few as possible
- Binaries x_{ji} ("item j goes into bin i") and y_i ("bin i is used")

```
x = m.addVars(n, k, vtype=GRB.BINARY, name="x")
y = m.addVars(k, vtype=GRB.BINARY, name="y")
m.setObjective(y.sum(), GRB.MINIMIZE)
m.addConstrs((x.sum(j, "*") == 1 for j in range(n)),
             name="assign")
m.addConstrs((gp.quicksum(w[j] * x[j, i] for j in range(n))
              <= c * y[i] for i in range(k)), name="capacity")
```

Scheduling and the travelling salesman

- **Scheduling on identical machines** $(P||z)$ changes only the objective and one constraint:
  - a continuous T for the makespan, min T
  - $\sum_j t_j x_{ji} \leq T$ for every machine
- The **travelling salesman** has one binary per arc, x_{ij}, and two families of equalities:
  - every city is left once and entered once
  - plus the constraints forbidding subtours, which are the interesting part

Summary

- A gurobipy model always uses the same instructions: `Model`, `addVars`, `setObjective`, `addConstr(s)`, `optimize`
- `vtype` declares the domain, which is part of the model
- `quicksum` for sums, one `addConstrs` per family
- Indices from zero in Python, from one in written models

The four classes of variables

Introduction: the four classes of variables

- Every variable of a model has a domain, and in gurobipy the domain is declared
- The classes needed are four: binary, integer, continuous, free
- Two default values are often forgotten, and they produce silently wrong models

The four classes of variables

What	`vtype`	Domain	Typical use
yes/no decision	`GRB.BINARY`	$\{0, 1\}$	selection, activation, assignment
count	`GRB.INTEGER`	$\mathbb{Z}$ between `lb` and `ub`	boxes, shifts, workers
measurable quantity	(default)	$[\texttt{lb}, \texttt{ub}] \subseteq \mathbb{R}$	time, money, flow
free variable	(default) with `lb=-GRB.INFINITY`	$\mathbb{R}$	duals of equalities

Observation 1: The two defaults people forget

`GRB.BINARY` already implies `lb` $= 0$ and `ub` $= 1$: there is no need to repeat them.

A continuous variable has `lb = 0` by default: a variable that must be allowed to go negative — typically the dual of an equality, or a signed deviation — must be declared with `lb=-GRB.INFINITY`, otherwise the model is silently wrong.

Summary

- Binary, integer, continuous and free: four classes, declared with `vtype` and the bounds
- Binaries already have bounds 0 and 1
- Continuous variables have `lb = 0` by default: a free variable must be declared with `lb=-GRB.INFINITY`

The model, one family per block

Introduction: the model, one family per block

- We see how the course writes a complete model in a Python function
- Three writing rules make the code match the mathematical model, line by line
- Finally we see how to check that the written model is the one designed on paper

An assignment in a function

```
def model(t, c, a):
    m = gp.Model("assignment");  m.Params.OutputFlag = 0
    x = m.addVars(n, k, vtype=GRB.BINARY, name="x")
    m.setObjective(gp.quicksum(c[j][h] * x[j, h]
                   for j in range(n) for h in range(k)),
                   GRB.MINIMIZE)
    m.addConstrs((x.sum(j, "*") == 1 for j in range(n)),
                 name="assign")
    m.addConstrs((gp.quicksum(t[j][h] * x[j, h]
                  for j in range(n)) <= a[h] for h in range(k)),
                 name="availability")
    return m, x
```

The three writing rules of the course (1/2)

1. **One** `addConstrs` **per family**, in the order of the mathematical model
   - the `name` is the label of the family (`"assign"`, `"availability"`)
   - the generated names are `assign[0]`, `assign[1]`, ...: they reappear in the duals and in the solver messages

The three writing rules of the course (2/2)

2. **Variables are declared all at once**, with `addVars` and the indices of the model
   - `x.sum(j, "*")` is the sum over all the indices of the second dimension: it is the writing of the model's *dummy index*
3. **Data come in as arguments**, not as globals
   - the same function serves the base instance and all the variants of the modelling questions

Checking the model

- The model as written has 9 variables, 6 constraints and 18 nonzero coefficients
- To check that it is what was written on paper, `m.write("model.lp")` produces the readable LP format
- It is the quickest way to spot a wrong coefficient: the instance table in the notes and this output must agree line by line

The file model.lp (1/2)

```
Minimize
  5 x[0,0] + 10 x[0,1] + 2 x[0,2] + 5 x[1,0]
   + 4 x[1,1] + 6 x[1,2] + 5 x[2,0] + 4 x[2,1]
   + 6 x[2,2]
Subject To
 assign[0]: x[0,0] + x[0,1] + x[0,2] = 1
 assign[1]: x[1,0] + x[1,1] + x[1,2] = 1
 assign[2]: x[2,0] + x[2,1] + x[2,2] = 1
```

The file model.lp (2/2)

```
 availability[0]: 2 x[0,0] + 3 x[1,0] + 4 x[2,0] <= 5
 availability[1]: x[0,1] + 4 x[1,1] + 5 x[2,1] <= 6
 availability[2]: 3 x[0,2] + 2 x[1,2] + 3 x[2,2] <= 7
Bounds
Binaries
 x[0,0] x[0,1] x[0,2] x[1,0] x[1,1] x[1,2]
 x[2,0] x[2,1] x[2,2]
End
```

Summary

- One `addConstrs` per family, with the `name` equal to the label in the model
- Variables declared all at once with `addVars`; `x.sum(j, "*")` is the dummy index
- Data as arguments of the function
- `m.write("model.lp")` to compare line by line with the written model

Reading the results

Introduction: reading the results

- After `optimize()` the solver does not always give the optimum: it may stop earlier, or find that there are no solutions
- The results are read in a precise order, which never changes
- We see what can be said — and what must *not* be said — in each case

The reading order, which never changes

1. `Status`. Always first.
2. `SolCount`: if it is 0, *no* solution exists and `ObjVal` cannot be read.
3. `ObjVal` and `ObjBound`: the best solution found and the best bound proved.
4. `MIPGap`, `NodeCount`, `Runtime`: how much remains uncertain, and how much it cost.

What can be said, case by case

Status	value	SolCount	What can be said
OPTIMAL	2	≥ 1	$z(MILP) =$ ObjVal, proved
INFEASIBLE	3	0	the model has no feasible solution
UNBOUNDED	5	0	a constraint, or a bound on a variable, is missing
TIME_LIMIT	9	0	nothing: neither a solution nor, in general, a useful bound
TIME_LIMIT	9	≥ 1	ObjBound $\leq z(MILP) \leq$ ObjVal
SOLUTION_LIMIT	10	≥ 1	as above

Example: the four cases on the assignment instance (1/2)

- **Normal solve.**
  - `Status` $= 2$, `SolCount` $= 2$, `ObjVal` $=$ `ObjBound` $= 11$
  - `MIPGap` $= 0$, `NodeCount` $= 0$: Gurobi closed everything at the root
- **Infeasible.**
  - with availability $a = (1, 1, 1)$ no assignment is possible
  - `Status` $= 3$, `SolCount` $= 0$

Example: the four cases on the assignment instance (2/2)

- **Stopped immediately.** With `TimeLimit = 0`:
  - `Status` $= 9$, `SolCount` $= 0$ and `ObjBound` $= -\infty$
  - there is nothing to report: neither a solution nor a bound
- **Stopped at the first solution.** With `SolutionLimit = 1`:
  - `Status` $= 10$, `SolCount` $= 1$, `ObjVal` $= 12$, `ObjBound` $= 10$, `MIPGap` $= 0.1667$
  - an *interval* is reported: the optimum lies between 10 and 12, and the 16.7% measures its width
  - saying "the optimum is 12" would be false

Observation 2: What is *not* reported

- *One does not write "the optimum is* `ObjVal` *" if* `Status` *is not* `OPTIMAL`*.*
- *One does not write a gap if* `SolCount` *is* 0*.*
- *One does not compare the* `Runtime` *of two models solved with different settings.*
- *One does not read* `ObjBound` *at the end of the solve thinking it is the root relaxation: for that,* `relax()` *is needed.*

Summary

- Reading order: `Status`, `SolCount`, `ObjVal` and `ObjBound`, then `MIPGap`, `NodeCount`, `Runtime`
- Only with `OPTIMAL` does one speak of the optimum
- Stopped with at least one solution: the interval $\texttt{ObjBound} \leq z(MILP) \leq \texttt{ObjVal}$ is reported
- With $\texttt{SolCount} = 0$ there is neither a solution nor a gap

Tolerances

Introduction: tolerances

- The solver works in floating-point arithmetic: nothing is exact
- Four parameters say how much the solver may be off on integrality, constraints, reduced costs and gap
- Knowing them avoids mistakes when reading and comparing the values of the variables

The tolerance parameters

Parameter	Default	Meaning
`IntFeasTol`	10^{-5}	how far an integer variable may be from an integer
`FeasibilityTol`	10^{-6}	violation allowed on a linear constraint
`OptimalityTol`	10^{-6}	tolerance on the reduced costs
`MIPGap`	10^{-4}	relative gap below which the solver stops

Observation 3: "Integer" means "integer within a tolerance" (1/2)

A binary may come back as 0.9999999997*. In the text one writes* 1*: values are rounded when they are reported, and comparisons always use a tolerance — in this course* 10^{-6}*, the constant* `TOL` *of* `python/mip.py`*.*

Writing `if x.X == 1` *is a mistake; one writes* `if x.X > 0.5`*.*

Observation 4: "Integer" means "integer within a tolerance" (2/2)

The default `MIPGap` $= 10^{-4}$ *also means the solver may stop before the exact optimum while declaring* `OPTIMAL`*: on instances with large values it is worth lowering it.*

Summary

- `IntFeasTol`, `FeasibilityTol`, `OptimalityTol`, `MIPGap`: the four tolerances
- Values are rounded when reported; comparisons use a tolerance (`x.X > 0.5`, never `x.X == 1`)
- `OPTIMAL` means optimal within `MIPGap`: on large values it is worth lowering it

The relaxation with `relax()`

Introduction: the relaxation with `relax()`

- The linear relaxation gives the bound $z(LP)$ and the dual variables
- In gurobipy it is obtained with `relax()`, which copies the model dropping integrality
- We see the code and the numbers on the assignment instance

The relaxation in gurobipy

```
m.update()      # relax() copies the model: pending
                # changes must be applied first
r = m.relax()   # binaries -> 0 <= x <= 1, integers -> x >= lb
r.Params.OutputFlag = 0
r.optimize()
zlp = r.ObjVal
duals = {c.ConstrName: c.Pi for c in r.getConstrs()}
```

The relaxation on the assignment instance

- On that instance $z(LP^+) = z(LP) = 53/5$
- The two relaxations coincide, because the assignment constraints already imply $x_{jm} \leq 1$
- The nonzero duals are

$$\tilde{\mu} = (2,\ 4.8,\ 5), \qquad \tilde{\pi}_2 = -0.2$$

- That is, the three dual variables of the `assign` constraints and the one of `availability[1]`: machine 2 is the only tight resource

The relaxation without the bounds, when it is needed

- The function `rilassamento(m, rafforzato=False)` of `python/mip.py` also drops the upper bound of the binaries
- It is the relaxation whose dual the notes write by hand
- Both versions are reported, with a note on whether they coincide and why

Summary

- `m.update()` and then `m.relax()`: a copy of the model without integrality
- `r.ObjVal` is $z(LP)$, `c.Pi` the dual of each constraint
- With or without the bounds $x \leq 1$: both relaxations are reported, saying whether they coincide and why

Three problems the course reuses

Introduction: three problems the course reuses

- So far the example model has always been the knapsack
- The three problems of this section — bin packing, makespan, travelling salesman — come back in the heuristics chapter
- There the solutions of next-fit, first-fit, best-fit, LPT and nearest neighbour are built by hand
- Here their models are written, so that those heuristics have an optimum to compare themselves with

Bin packing: how many containers are enough

Bin packing

- There are $n \in \mathbb{Z}_{\geq 1}$ items: item $j \in \{1, 2, \ldots, n\}$ weighs $w_j \in \mathbb{Q}_{>0}$
- The containers are all alike, of capacity $c \in \mathbb{Q}_{>0}$, and at most one per item may be used
- Use the smallest number of containers
- Two families of binary variables are needed:
  - $x_{jb} = 1$ if item j goes into container b
  - $y_b = 1$ if container b is used

Bin packing

$$\min \quad \sum_{b=1}^{k} y_b \tag{3.1a}$$

$$\text{subject to} \quad \sum_{b=1}^{k} x_{jb} = 1, \qquad \forall j \in \{1, 2, \ldots, n\}, \tag{3.1b}$$

$$\sum_{j=1}^{n} w_j \, x_{jb} - c \, y_b \leq 0, \qquad \forall b \in \{1, 2, \ldots, k\}, \tag{3.1c}$$

$$x_{jb} \in \{0, 1\}, \quad \forall j \in \{1, 2, \ldots, n\}, \ \forall b \in \{1, 2, \ldots, k\}, \tag{3.1d}$$

$$y_b \in \{0, 1\}, \qquad \forall b \in \{1, 2, \ldots, k\}. \tag{3.1e}$$

Bin packing: description of the model

- The objective counts the containers switched on
- The constraints $\sum_b x_{jb} = 1$ say that every item ends up in exactly one container
- The capacity constraints are written as an *activation*: while $y_b = 0$ container b can receive nothing, and as soon as $y_b = 1$ it takes up to c
- The domains are the two families of binaries, one per item–container pair and one per container

Bin packing: building it in gurobipy

```
def modello_bpp(w, c, k):
    n = len(w)
    m = nuovo_modello("bin_packing")
    x = m.addVars(n, k, vtype=GRB.BINARY, name="x")
    y = m.addVars(k, vtype=GRB.BINARY, name="y")
    m.setObjective(y.sum(), GRB.MINIMIZE)
    m.addConstrs((x.sum(j, "*") == 1 for j in R(n)),
                 name="item")
    m.addConstrs((gp.quicksum(w[j] * x[j, b] for j in R(n))
                  <= c * y[b] for b in R(k)), name="capacity")
    return m, x, y
```

Bin packing on the instance

Bin packing on the instance

$$
\begin{aligned}
\min \quad & y_1 + y_2 + y_3 \\
\text{subject to} \quad & x_{11}+x_{12}+x_{13} = 1 \\
& x_{21} + x_{22} + x_{23} = 1 \\
& x_{31} + x_{32} + x_{33} = 1 \\
& x_{41} + x_{42} + x_{43} = 1 \\
& 5x_{11} + 4x_{21} + 3x_{31} + 3x_{41} - 7y_1 \le 0 \\
& 5x_{12} + 4x_{22} + 3x_{32} + 3x_{42} - 7y_2 \le 0 \\
& 5x_{13} + 4x_{23} + 3x_{33} + 3x_{43} - 7y_3 \le 0 \\
& x_{11}, x_{12}, x_{13}, x_{21}, x_{22}, x_{23}, x_{31}, x_{32}, x_{33}, x_{41}, x_{42}, x_{43} \in \{0,1\} \\
& y_1, y_2, y_3 \in \{0,1\}
\end{aligned}
$$

Bin packing: the count on the instance

- The total weight is 15, so no solution can use fewer than $\lceil 15/7 \rceil = 3$ containers
- The optimum uses exactly 3: the count is therefore tight
- In the model the possible containers are three, the number actually needed
- With six containers the model would have 42 columns and the last three would stay empty in every optimal solution

Observation 5: The relaxation of bin packing is very weak

Relaxing y_b to $y_b \geq 0$ the model buys fractions of a container, and the LP optimum drops to $\sum_j w_j/c = 15/7 \approx 2.14$: the relaxation does not know that a container opens whole.

This is why on this problem the dual bound is of little use and the heuristics matter more.

Makespan: the load of the busiest machine

Makespan on identical machines

- There are $n \in \mathbb{Z}_{\geq 1}$ jobs, of duration $d_j \in \mathbb{Q}_{>0}$, and $k \in \mathbb{Z}_{\geq 1}$ identical machines
- Every job goes on a single machine and is not interrupted
- Minimise the load of the busiest machine
- Variables:
  - $x_{jm} = 1$ if job j goes on machine m
  - $z \geq 0$ the load of the busiest machine

Makespan on identical machines

$$\min \quad z \tag{3.2a}$$

$$\text{subject to} \quad \sum_{m=1}^{k} x_{jm} = 1, \qquad \forall j \in \{1, 2, \ldots, n\}, \tag{3.2b}$$

$$\sum_{j=1}^{n} d_j \, x_{jm} - z \leq 0, \qquad \forall m \in \{1, 2, \ldots, k\}, \tag{3.2c}$$

$$x_{jm} \in \{0, 1\}, \quad \forall j \in \{1, 2, \ldots, n\}, \ \forall m \in \{1, 2, \ldots, k\}, \tag{3.2d}$$

$$z \geq 0. \tag{3.2e}$$

Makespan: description of the model

- The objective is the single variable z: no datum appears in it
- It is the k load rows that give it meaning, saying that no machine works longer than z
- The minimisation then presses it down onto the load of the busiest machine: it is the *min-max* technique
- The constraints $\sum_m x_{jm} = 1$ assign every job to a single machine; the domains declare the assignments binary and the load non-negative

Makespan: building it in gurobipy

```
def modello_cmax(d, k):
    n = len(d)
    m = nuovo_modello("makespan")
    x = m.addVars(n, k, vtype=GRB.BINARY, name="x")
    z = m.addVar(name="z")
    m.setObjective(z, GRB.MINIMIZE)
    m.addConstrs((x.sum(j, "*") == 1 for j in R(n)),
                 name="job")
    m.addConstrs((gp.quicksum(d[j] * x[j, mm] for j in R(n))
                  <= z for mm in R(k)), name="load")
    return m, x, z
```

Makespan on the instance

Makespan on the instance

$$
\begin{aligned}
\min \quad & z \\
\text{subject to} \quad & x_{11} + x_{12} = 1 \\
& x_{21} + x_{22} = 1 \\
& x_{31} + x_{32} = 1 \\
& x_{41} + x_{42} = 1 \\
& 3x_{11} + 4x_{21} + 5x_{31} + 6x_{41} - z \leq 0 \\
& 3x_{12} + 4x_{22} + 5x_{32} + 6x_{42} - z \leq 0 \\
& x_{11}, x_{12}, x_{21}, x_{22}, x_{31}, x_{32}, x_{41}, x_{42} \in \{0, 1\} \\
& z \geq 0
\end{aligned}
$$

Makespan: the count on the instance

- The total load is 18 and the machines are two: no solution can go below $18/2 = 9$
- The optimum is exactly 9: the jobs split into $6 + 3$ and $5 + 4$
- Here the count settles the problem on its own

Travelling salesman: the MTZ formulation

Travelling salesman

- There are $n \in \mathbb{Z}_{\geq 2}$ cities and a distance $d_{ij} \in \mathbb{Q}_{>0}$ between every pair
- Find the shortest tour that visits every city exactly once and returns to the starting point

Travelling salesman: the variables

- $x_{ij} = 1$ if the tour goes from i to j
- The two families "one leaves once" and "one enters once" are not enough: they also admit solutions made of separate *subtours*
- The Miller–Tucker–Zemlin formulation adds a variable u_i for every city other than the first, recording its position along the tour

TSP, MTZ formulation

$$\min \quad \sum_{i=1}^{n} \sum_{j \neq i} d_{ij}\, x_{ij} \tag{3.3a}$$

$$\text{subject to} \quad \sum_{j \neq i} x_{ij} = 1, \qquad \forall i \in \{1, 2, \ldots, n\}, \tag{3.3b}$$

$$\sum_{i \neq j} x_{ij} = 1, \qquad \forall j \in \{1, 2, \ldots, n\}, \tag{3.3c}$$

$$u_i - u_j + n\, x_{ij} \leq n - 1, \qquad \forall i, j \in \{2, 3, \ldots, n\},\ i \neq j, \tag{3.3d}$$

$$x_{ij} \in \{0, 1\}, \qquad \forall i, j \in \{1, 2, \ldots, n\},\ i \neq j, \tag{3.3e}$$

$$u_i \in [1,\ n - 1], \qquad \forall i \in \{2, 3, \ldots, n\}. \tag{3.3f}$$

TSP: description of the model (1/2)

- The objective sums the distances of the arcs travelled
- The two families of equalities say that one leaves every city once and enters every city once
- The MTZ constraints are the heart of the formulation:
  - if $x_{ij} = 0$ the row becomes $u_i - u_j \leq n - 1$, always true because the u lie between 1 and $n - 1$: it forbids nothing
  - if $x_{ij} = 1$ it becomes $u_j \geq u_i + 1$, that is "if I go from i to j, the position of j is the next one"

TSP: description of the model (2/2)

- A subtour missing city 1 would need a chain of ever-growing positions closing on itself
- This is impossible
- City 1 has no u of its own precisely because it is where the tour closes

TSP: building it in gurobipy (1/2)

The variables and the objective:

```
def modello_tsp(D):
    n = len(D)
    m = nuovo_modello("tsp")
    x = m.addVars(((i, j) for i in R(n) for j in R(n)
                   if i != j), vtype=GRB.BINARY, name="x")
    u = m.addVars(R(1, n), lb=1, ub=n - 1, name="u")
    m.setObjective(gp.quicksum(D[i][j] * x[i, j]
                   for i, j in x), GRB.MINIMIZE)
```

TSP: building it in gurobipy (2/2)

The constraints, one family per block:

```
m.addConstrs((gp.quicksum(x[i, j] for j in R(n) if j != i)
              == 1 for i in R(n)), name="out")
m.addConstrs((gp.quicksum(x[i, j] for i in R(n) if i != j)
              == 1 for j in R(n)), name="in")
m.addConstrs((u[i] - u[j] + n * x[i, j] <= n - 1
              for i in R(1, n) for j in R(1, n) if i != j),
             name="mtz")
return m, x, u
```

TSP on the instance

- On the four-city instance of the heuristics chapter the optimal tour is $1 \to 2 \to 4 \to 3 \to 1$ and measures 22
- The model of the instance has fifteen columns — twelve arcs and three positions — and fourteen rows

TSP on the instance

TSP on the instance

$$
\begin{aligned}
\min\ & 4x_{12}+5x_{13}+9x_{14}+4x_{21}+9x_{23}+9x_{24}+5x_{31}+9x_{32}+4x_{34}+9x_{41}+9x_{42}+4x_{43} \\
\text{subject to}\ & x_{12}+x_{13}+x_{14} = 1 \\
& x_{21}+x_{23}+x_{24} = 1 \\
& x_{31}+x_{32}+x_{34} = 1 \\
& x_{41}+x_{42}+x_{43} = 1 \\
& x_{21}+x_{31}+x_{41} = 1 \\
& x_{12}+x_{32}+x_{42} = 1 \\
& x_{13}+x_{23}+x_{43} = 1 \\
& x_{14}+x_{24}+x_{34} = 1 \\
& 4x_{23}+u_2-u_3 \le 3 \\
& 4x_{24}+u_2-u_4 \le 3 \\
& 4x_{32}-u_2+u_3 \le 3 \\
& 4x_{34}+u_3-u_4 \le 3 \\
& 4x_{42}-u_2+u_4 \le 3 \\
& 4x_{43}-u_3+u_4 \le 3 \\
& x_{12},\ x_{13},\ x_{14},\ x_{21},\ x_{23},\ x_{24},\ x_{31},\ x_{32},\ x_{34},\ x_{41},\ x_{42},\ x_{43} \in \{0,1\} \\
& u_2,\ u_3,\ u_4 \ge 1
\end{aligned}
$$

Observation 6: MTZ is convenient, not the strongest

The MTZ constraints are $O(n^2)$ and take three lines of `gurobipy`*, but their linear relaxation is weak: the continuous u absorb almost everything and the LP stays far from the integer optimum.*

Formulations that remove subtours with connectivity cuts give much better bounds, at the price of an exponential number of constraints to be generated as they are needed. For the sizes of this course MTZ is enough.

Summary (1/2)

Bin packing:

- Binaries x_{jb} and y_b; capacity written as the activation $\sum_j w_j x_{jb} \leq c\, y_b$
- On the instance the optimum is $3 = \lceil 15/7 \rceil$; the relaxation (15/7) is very weak

Makespan:

- Min-max technique: $\min z$ with $\sum_j d_j x_{jm} \leq z$ for every machine
- On the instance the optimum is $9 = 18/2$: the count settles the problem

Summary (2/2)

Travelling salesman (MTZ):

- Equalities "leave once" and "enter once" plus the positions u_i
- $u_i - u_j + n\,x_{ij} \leq n - 1$ forbids the subtours that miss city 1
- Convenient to write, but with a weak relaxation
- On the instance the optimal tour measures 22

The course protocol, from start to finish

Introduction: the course protocol

- Every exercise of the course follows the same sequence, from the data to the table of bounds
- Every step produces a number that is checked automatically
- The result ends up in a single file, from which the notes, the website and the checks read it

The sequence of every exercise

$$\text{data} \rightarrow \text{model} \rightarrow \text{heuristic and check} \rightarrow \text{LP and dual}$$
$$\rightarrow \text{MIP} \rightarrow \text{table} \rightarrow \text{figures and notebook}$$

The protocol in code (1/2)

Data, model, heuristic and check:

```
m, x = model(t, c, a)                          # (1) data, (2) model

e = best_fit(t, a, lambda j, h, ra: c[j][h], "cost")
ub = sum(c[j][h] for (j, h) in e.x)     # (3) heuristic
# constraints, bounds AND integrality:
assert ammissibile(m, {f"x[{j},{h}]": 1 for (j, h) in e.x})
```

The protocol in code (2/2)

Dual, relaxations, MIP and table:

```
d = dual(t, c, a)                    # (4) the dual written by hand
by_hand = {f"mu[{j}]": min(c[j]) for j in range(n)}
lb, viol = valuta(d, by_hand);  assert viol <= 1e-9
zlp, zlp_str, pi = due_rilassamenti(m, d)  # strong duality

z = risolvi(m)                                        # (5) the MIP
row = registra_bound("assignment", ub, lb, zlp,
                     zlp_str, z)                      # (6) table
salva_dati(pd.DataFrame([row]), "fam07_1_bound")
```

The four numbers of the protocol

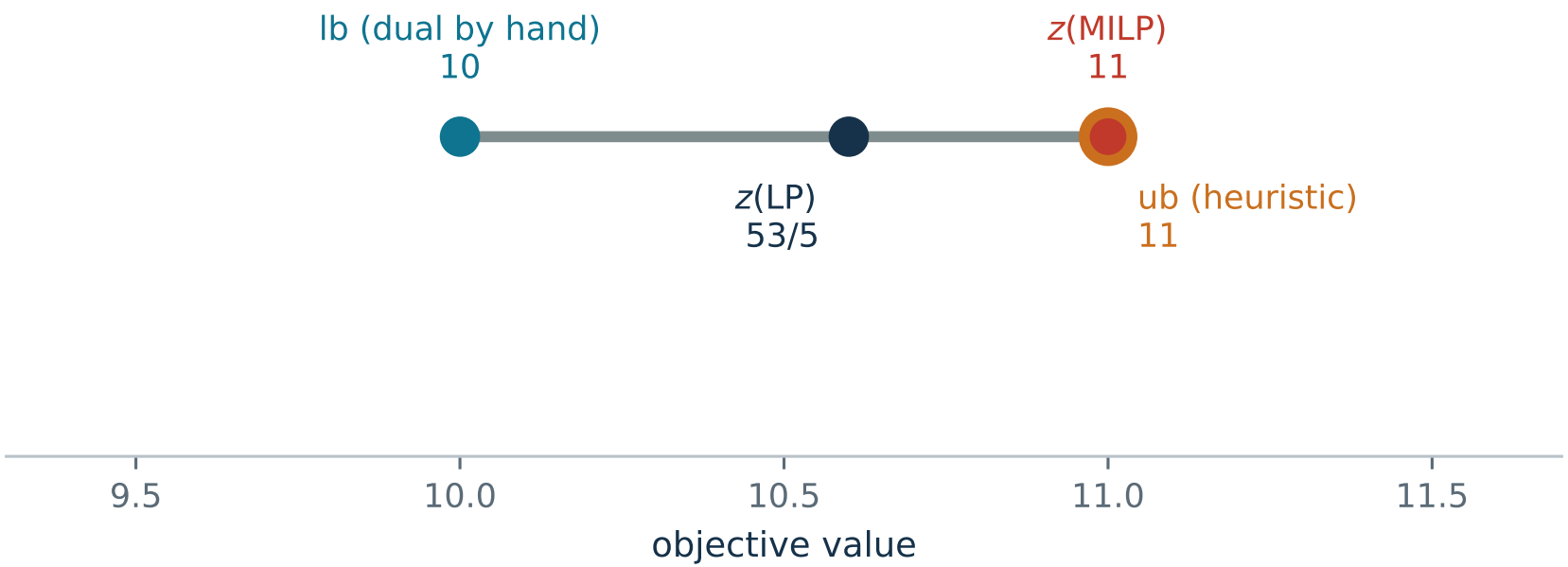
- On the assignment instance of this chapter the protocol produces

$$LB = 10, \qquad z(LP) = z(LP^{+}) = \tfrac{53}{5}, \qquad z(MILP) = 11, \qquad UB = 11$$

- The row ends up in `data/sched1_bound.csv`
- That is where the notes, the website page and `check_numbers.py` read it from
- **One single place where the number exists**

The four numbers on the assignment instance

The four numbers every exercise produces



Summary

- Data $\rightarrow$ model $\rightarrow$ heuristic and check $\rightarrow$ LP and dual $\rightarrow$ MIP $\rightarrow$ table $\rightarrow$ figures and notebook
- Every number is checked by an `assert`
- On the assignment instance:
  $LB = 10 \leq z(LP) = 53/5 \leq z(MILP) = 11 = UB$
- One single place where the number exists: the CSV file in `data/`

Running the code

Introduction: running the code

- The course code can be run in two ways: locally or on Colab
- In both cases the licence bundled with `gurobipy` is enough
- We see the commands and why the licence is sufficient

Locally

From the python/ folder:

```
python3 -m venv .venv && source .venv/bin/activate
pip install gurobipy pandas matplotlib
python3 fam07_1_assignment.py  # one chapter
python3 run_all.py             # all of them, plus notebooks
python3 check_numbers.py       # the asserts on the numbers
```

On Colab

- Every chapter has a notebook that opens from the badge on the website page
- The first cell installs `gurobipy` and downloads the three shared modules (`stile.py`, `mip.py`, `euristiche.py`) if they are not already there
- Nothing needs to be installed on the computer

Observation 7: The licence bundled with `gurobipy` is enough, and why

*The size-limited licence of the pip package allows models of up to **2000 variables and 2000 constraints**. The instances of this course are tiny — the largest model has a few dozen variables — and all fit with an enormous margin.*

For larger instances the free academic licence is activated from portal.gurobi.com. If a model exceeds the limit, Gurobi reports it with an explicit error at `optimize()`: it does not silently produce a wrong result.

Summary

- Locally: a virtual environment, `pip install gurobipy pandas matplotlib`, then the scripts
- On Colab: the notebook installs everything by itself
- The licence of the pip package (2000 variables, 2000 constraints) is enough for the whole course

Final summary (1/2)

Writing the model:

- A handful of instructions: `Model`, `addVars`, `setObjective`, `addConstrs`, `optimize`
- The domain (`vtype`, `lb`) is part of the model
- One `addConstrs` per family, with the name of the written model; check with `m.write`

Reading the results:

- `Status` first of all, then `SolCount`, `ObjVal`, `ObjBound`
- If the solver has not finished an interval is reported, not an optimum

Final summary (2/2)

Tolerances and relaxation:

- Integers within a tolerance: comparisons with `> 0.5`
- `relax()` gives $z(LP)$ and the duals

Problems and protocol:

- Bin packing, makespan and TSP (MTZ) are the reference models for the heuristics
- The protocol goes from the data to the table
  $LB \leq z(LP) \leq z(MILP) \leq UB$