Masterclass

May 4, 2022

1 Masterclass on Combinatorial Optimization

Combinatorial Optimization course, FEE CTU in Prague. Created by Industrial Informatics Department.

In the first part of today's lab, we will explore vast possibilities offered by Gurobi solver, showing it is much more than an ILP solver. The second part will revisit some of the previously studied problems that we formalized using MILP. This time, we will apply Constraint Programming (CP) and evaluate which method is better for which type of the problem.

[1]:
```
# pip install -i https: // pypi.gurobi.com gurobipy
import gurobipy as g
import matplotlib.pyplot as plt
import numpy as np
import networkx as nx
import itertools as iter
import math
from collections import namedtuple
```

2 1) GUROBI Automated model tuning

Gurobi solver provides a wide variety of model parameters that can be tweaked to improve the solver's performance. While you can tinker with these yourself manually or by writing some sort of grid search, the Gurobi solver provides a way to tweak these parameters much more effortlessly. To demonstrate this, we will revisit the ILP model for Game of Fivers, which, as you might remember, took considerable time to solve for larger board sizes. We will use the auto-tuning tool to improve the solver's execution time (more information about it in video here). Let's start by the building of the model and executing it for some test instance:

[2]:
```
def game_of_fivers(n, params=None):
    m = g.Model()

    # "x" represent if the stone was selected to flip (we add extra border rows␣
↪and columns to avoid indexation problems when building constraint)
    x = m.addVars(n + 2, n + 2, vtype=g.GRB.BINARY, obj=1)
    # "k" is to help enforce oddness of the number of flips in the neighborhood␣
↪of each stone -> thus ensuring that it is flipped to black side at the end
    k = m.addVars(range(1, n + 1), range(1, n + 1), vtype=g.GRB.INTEGER)
```

```
    # Ensure that every stone (except the ones in border rows and columns which␣
↪were artificially added) was flipped odd number of times, so it ends up␣
↪flipped to black
    for i in range(1, n + 1):
        for j in range(1, n + 1):
            m.addConstr(x[i, j] + x[i + 1, j] + x[i - 1, j] + x[i, j + 1] +␣
↪x[i, j - 1] == 2 * k[i, j] + 1)

    # Stones in bordering rows and columns can't be the ones being flipped
    m.addConstr(x.sum(0, "*") + x.sum(n + 1, "*") + x.sum("*", 0) + x.sum("*",␣
↪n + 1) == 0)

    # We save the model for the current problem instance
    m.write('fivers_data/fivers_n{}.lp'.format(n))

    # If we have parameters, set them to the model
    if params is not None:
        for param, value in params.items():
            m.setParam(param, value)
        m.params.outputflag = 0  # disable the standard output of the solver

    m.optimize()

    X = [[int(round(x[i, j].X)) for j in range(1, n + 1)] for i in range(1, n +␣
↪1)]
    return m.runtime
```

[3]:
```
# Run with default parameters
def_time = game_of_fivers(21)  # Save the execution time, so we can later␣
↪compare it with the tuned version
print('Time with default settings: {}s'.format(def_time))
```

```
Academic license - for non-commercial use only - expires 2022-09-01
Using license file /Users/novakan9/gurobi.lic
Gurobi Optimizer version 9.1.2 build v9.1.2rc0 (mac64)
Thread count: 6 physical cores, 12 logical processors, using up to 12 threads
Optimize a model with 442 rows, 970 columns and 2734 nonzeros
Model fingerprint: 0xaa07108c
Variable types: 0 continuous, 970 integer (529 binary)
Coefficient statistics:
  Matrix range     [1e+00, 2e+00]
  Objective range  [1e+00, 1e+00]
  Bounds range     [1e+00, 1e+00]
  RHS range        [1e+00, 1e+00]
Presolve removed 1 rows and 96 columns
Presolve time: 0.01s
```

```
Presolved: 441 rows, 874 columns, 2533 nonzeros
Variable types: 0 continuous, 874 integer (518 binary)

Root relaxation: objective 9.585736e+01, 1238 iterations, 0.07 seconds

    Nodes    |    Current Node    |     Objective Bounds      |     Work
 Expl Unexpl |  Obj  Depth IntInf | Incumbent    BestBd   Gap | It/Node Time

     0     0   95.85736    0  440          -   95.85736      -     -    0s
     0     0   96.94696    0  468          -   96.94696      -     -    0s
     0     0   96.99182    0  469          -   96.99182      -     -    0s
     0     0   96.99375    0  470          -   96.99375      -     -    0s
     0     0   96.99408    0  470          -   96.99408      -     -    0s
     0     0   97.86566    0  477          -   97.86566      -     -    0s
     0     0   97.87560    0  475          -   97.87560      -     -    0s
     0     0   97.88237    0  478          -   97.88237      -     -    0s
     0     0   97.88258    0  477          -   97.88258      -     -    0s
     0     0   98.82660    0  490          -   98.82660      -     -    0s
     0     0   98.95163    0  494          -   98.95163      -     -    0s
     0     0   99.00039    0  498          -   99.00039      -     -    0s
     0     0   99.01761    0  499          -   99.01761      -     -    0s
     0     0   99.01909    0  504          -   99.01909      -     -    0s
     0     0   99.81988    0  501          -   99.81988      -     -    0s
     0     0   99.91494    0  506          -   99.91494      -     -    0s
     0     0   99.93230    0  509          -   99.93230      -     -    0s
     0     0   99.93331    0  510          -   99.93331      -     -    0s
     0     0  100.86086    0  512          -  100.86086      -     -    0s
     0     0  101.28866    0  523          -  101.28866      -     -    0s
     0     0  102.27016    0  521          -  102.27016      -     -    0s
     0     0  103.32723    0  521          -  103.32723      -     -    0s
     0     0  103.88010    0  544          -  103.88010      -     -    1s
     0     0  104.34467    0  539          -  104.34467      -     -    1s
     0     0  104.66254    0  554          -  104.66254      -     -    1s
     0     0  104.88282    0  549          -  104.88282      -     -    1s
H    0     0                    245.0000000  104.88282  57.2%     -    1s
     0     0  104.88282    0  548  245.00000  104.88282  57.2%     -    1s
     0     2  104.89753    0  548  245.00000  104.89753  57.2%     -    1s
  1294   867  109.37345   17  432  245.00000  105.90159  56.8%   116    5s
  1341   899  118.42853   30  658  245.00000  118.42853  51.7%   112   10s
  1378   923  122.15446   53  695  245.00000  122.15446  50.1%   109   15s

Cutting planes:
  Cover: 5
  MIR: 166
  Flow cover: 22
  Zero half: 183
  RLT: 3
```

```
Explored 1393 nodes (171387 simplex iterations) in 16.80 seconds
Thread count was 12 (of 12 available processors)

Solution count 1: 245

Optimal solution found (tolerance 1.00e-04)
Best objective 2.450000000000e+02, best bound 2.450000000000e+02, gap 0.0000%
Time with default settings: 16.80030608177185s
```

Now, let's try to tune parameters based on the saved model representing the instance from the previous execution.

```
[ ]: # First, we load the model for the problem instance
model = g.read('fivers_data/fivers_n21.lp')

model.params.tuneResults = 1  # How many sets of parameters we want to return␣
↪by the auto tuner
model.params.TuneTimeLimit = 30  # How much time to invest into the tuning

model.tune()  # Run the tuning
if model.tuneResultCount > 0:
    model.getTuneResult(0)  # Get the best (first) configuration
    model.write('fivers_data/fivers_tuned_params.prm')  # Save it into the file
```

In the output above, you can see information about the tuning process, which parameters were tested, if some parameter improved solver's results, etc. Note that the tuning process tries different seeds for the solver to be more robust against randomness in the solving process. When the tuning is done, let us run the same problem instance with the best parameter set found in the tuning process. *(Note that we hardcode found parameters based on a longer tuning search since it is possible that in a short time, you will not be able to find any better parameter settings)*:

```
[4]: print('Time before tuning: {}s'.format(def_time))
print('Time after tuning: {}s'.format(game_of_fivers(21, {'CutPasses': 10,␣
↪'PreDual': 1, "Presolve": 2})))
```

```
Time before tuning: 16.80030608177185s
Changed value of parameter CutPasses to 10
   Prev: -1  Min: -1  Max: 2000000000  Default: -1
Changed value of parameter PreDual to 1
   Prev: -1  Min: -1  Max: 2  Default: -1
Changed value of parameter Presolve to 2
   Prev: -1  Min: -1  Max: 2  Default: -1
Time after tuning: 10.88615107536316s
```

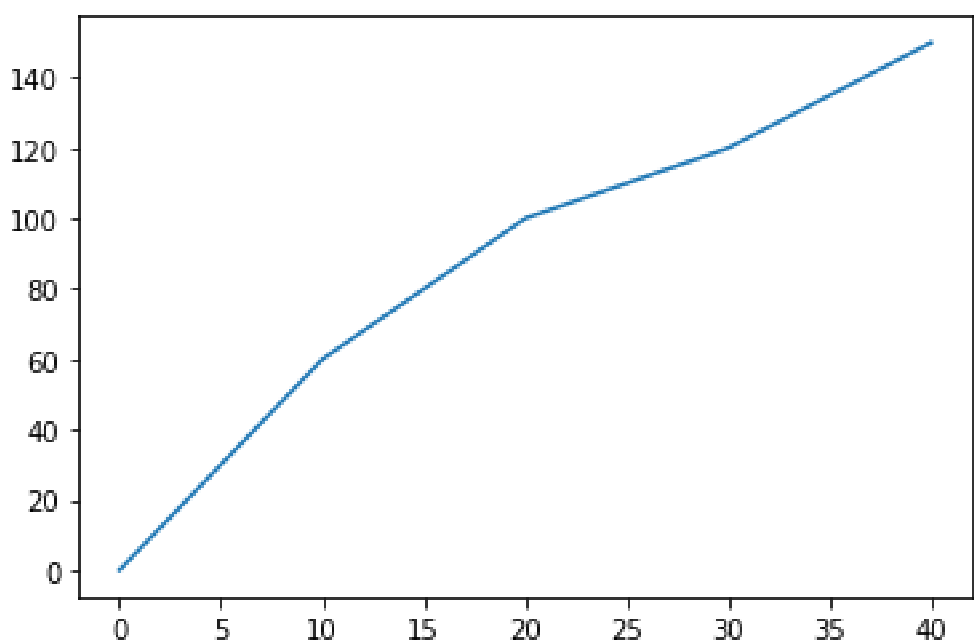
We can see that we cut down the execution time roughly to half by using the tuned parameters. However, the parameters were tweaked for one particular instance of the problem, so if we wanted to achieve a general improvement to the model, it would be wise to try multiple instances and choose what would work generally.

3 2) GUROBI Nonlinear (general) constraints

So far, we have focused on modelling problems where constraints over variables always represented one fixed linear relation regardless of the variable's value (x <= 10 or y <= x + 10). But imagine a case where we would need to change the function depending on the value of x. Of course, this could be achieved by using the big M trick, but it is not such a convenient and clean solution. Thus, let us consider the concept of the Piece-wise linear (PWL) function. As the name suggests, PWL allows us to change the form of the function/constraint depending on the value of variable x. Consider a simple problem where we are choosing how many products we will manufacture, but the price of the product is different based on how many of them we manufacture (wages, overtime, material discounts etc.):

[5]:
```
# We define lists containing steps (intervals) for number of products (x) and␣
↪their respective price sum (y)
price_x = [0, 10, 20, 30, 40]  # Minimum is 0 (thousands), maximum is 40␣
↪(thousands)
price_y = [0, 60, 100, 120, 150]  # Making 10 (thousands) products cost 60␣
↪(thousands) units, 20 products 100 units and so on, each interval is␣
↪linearly interpolated

# Let's plot the function to see how it looks
plt.plot(price_x, price_y)
```

[5]:
```
[<matplotlib.lines.Line2D at 0x1242b6e50>]
```



Imagine that when we produce too little, the non-scalable costs like rent payments for the factory space will make bigger portion of the overall cost making each product more expensive. On the other hand, if we produce too much, we will have to pay overtime, making products also more costly. So the production cost function's overall shape changes depending on the produced volume, which is illustrated by the graph above.

[6]:
```
# Now let's build model maximizing the profit (while adding constraint for␣
↪buying extra machines for increased volume)
m = g.Model()

x = m.addVar(vtype=g.GRB.CONTINUOUS, lb=0, ub=40)  # Number of products
y = m.addVar(vtype=g.GRB.CONTINUOUS, lb=0, ub=150)  # Cumulative price sum for␣
↪production

# The PWL constraint is defined by giving it the related variables and list of␣
↪intervals
m.addGenConstrPWL(x, y, price_x, price_y, "cost_constraint")

#For each 7 (thousand) products we need to buy extra machine to be able to␣
↪handle the manufacture process
machine_count = m.addVar(vtype=g.GRB.INTEGER)
m.addConstr(machine_count * 7 >= x)

# The selling price for each product is 10
m.setObjective(x * 10 - y - machine_count * 40, sense=g.GRB.MAXIMIZE)

m.optimize()

print(x.x, y.x)
```

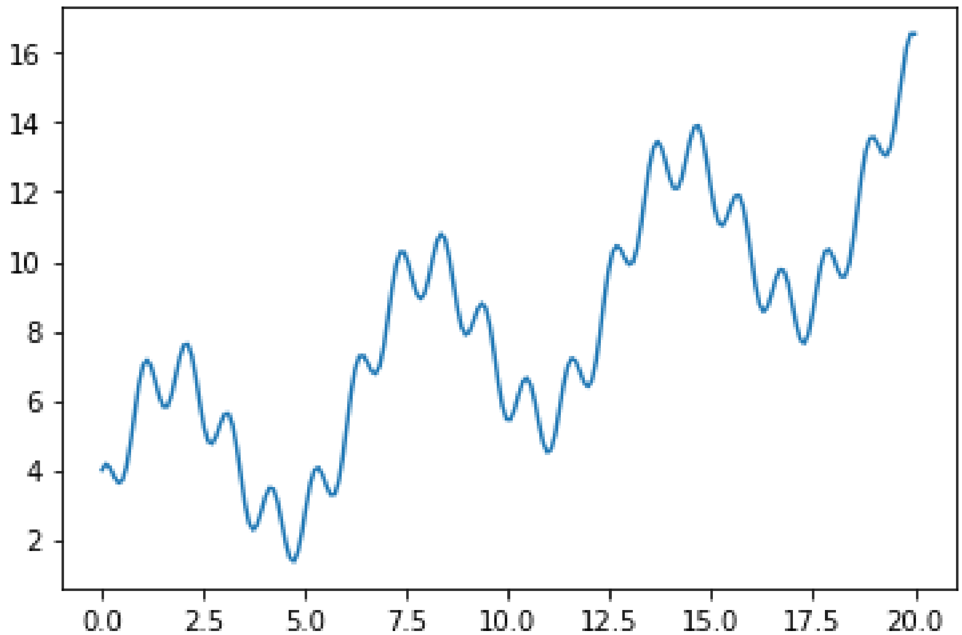
```
Gurobi Optimizer version 9.1.2 build v9.1.2rc0 (mac64)
Thread count: 6 physical cores, 12 logical processors, using up to 12 threads
Optimize a model with 1 rows, 3 columns and 2 nonzeros
Model fingerprint: 0x88382a9a
Model has 1 general constraint
Variable types: 2 continuous, 1 integer (0 binary)
Coefficient statistics:
  Matrix range     [1e+00, 7e+00]
  Objective range  [1e+00, 4e+01]
  Bounds range     [4e+01, 2e+02]
  RHS range        [0e+00, 0e+00]
Found heuristic solution: objective -0.0000000
Presolve added 3 rows and 5 columns
Presolve time: 0.00s
Presolved: 4 rows, 8 columns, 17 nonzeros
Presolved model has 1 SOS constraint(s)
Variable types: 7 continuous, 1 integer (0 binary)

Root relaxation: objective 2.142857e+01, 1 iterations, 0.00 seconds
```

```
    Nodes    |    Current Node    |     Objective Bounds      |     Work
 Expl Unexpl |  Obj  Depth IntInf | Incumbent    BestBd   Gap | It/Node Time

     0     0   21.42857    0    1   -0.00000   21.42857      -     -    0s
H    0     0                       10.0000000   21.42857   114%     -    0s
*    0     0               0       15.0000000   15.00000  0.00%     -    0s

Cutting planes:
  MIR: 1

Explored 1 nodes (3 simplex iterations) in 0.02 seconds
Thread count was 12 (of 12 available processors)

Solution count 3: 15 10 -0

Optimal solution found (tolerance 1.00e-04)
Best objective 1.500000000000e+01, best bound 1.500000000000e+01, gap 0.0000%
35.000000000000014 135.00000000000006
```

We already know that Gurobi can handle more than just linear constraints (for example, quadratic constraints), but what if we want to formulate even more obscure-looking functions? Gurobi has us covered. Let us illustrate an example where the function is some combination of sine and cosine plus some linear element:

[7]:
```python
n = 200
t = np.linspace(0, 20, n)
y = 3 * np.sin(t) + np.cos(6 * t) + 0.5 * t + 3

# Plot the final function
plt.plot(t, y)
```

[7]:
```
[<matplotlib.lines.Line2D at 0x1243f4250>]
```



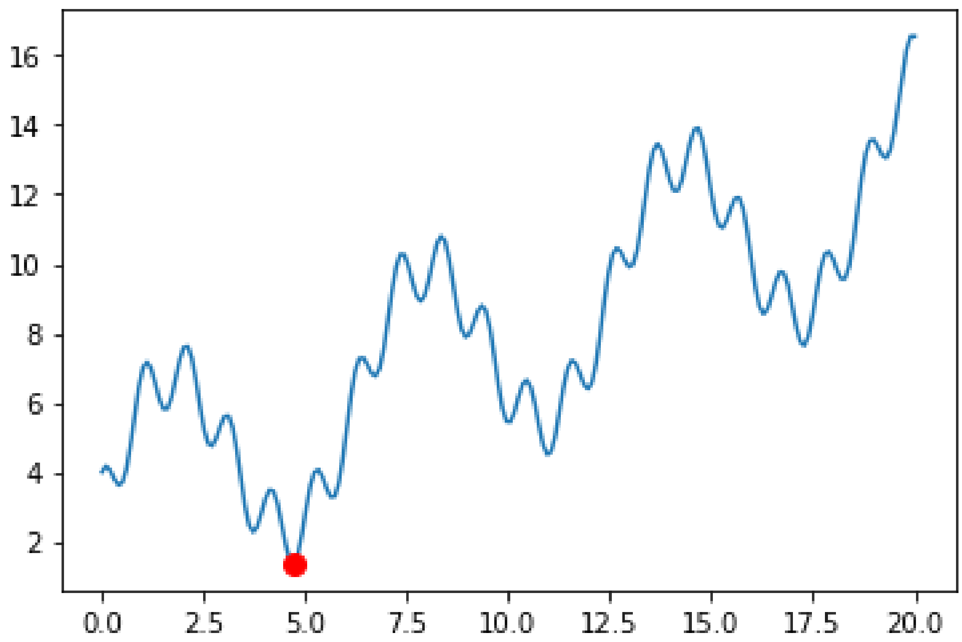
[8]:

```
m = g.Model()

u = m.addVar(vtype=g.GRB.CONTINUOUS)
v = m.addVar(vtype=g.GRB.CONTINUOUS)
m.addGenConstrPWL(u, v, t, y)  # Note that the intervals are sampled points in␣
↪"t" and "y"

m.setObjective(v, sense=g.GRB.MINIMIZE)

m.optimize()

plt.plot(t, y)
plt.plot(u.x, v.x, marker='o', markersize=8, color="red")
```

```
Gurobi Optimizer version 9.1.2 build v9.1.2rc0 (mac64)
Thread count: 6 physical cores, 12 logical processors, using up to 12 threads
Optimize a model with 0 rows, 2 columns and 0 nonzeros
Model fingerprint: 0x2d9d1a21
Model has 1 general constraint
Variable types: 2 continuous, 0 integer (0 binary)
Coefficient statistics:
  Matrix range     [0e+00, 0e+00]
  Objective range  [1e+00, 1e+00]
  Bounds range     [0e+00, 0e+00]
  RHS range        [0e+00, 0e+00]
Presolve added 1 rows and 197 columns
```

```
Presolve time: 0.00s
Presolved: 1 rows, 199 columns, 199 nonzeros
Presolved model has 1 SOS constraint(s)
Variable types: 199 continuous, 0 integer (0 binary)

Root relaxation: objective 1.364267e+00, 0 iterations, 0.00 seconds

    Nodes    |    Current Node    |     Objective Bounds      |     Work
 Expl Unexpl |  Obj  Depth IntInf | Incumbent    BestBd   Gap | It/Node Time

*    0     0               0       1.3642670    1.36427  0.00%     -    0s

Explored 0 nodes (0 simplex iterations) in 0.01 seconds
Thread count was 12 (of 12 available processors)

Solution count 1: 1.36427

Optimal solution found (tolerance 1.00e-04)
Best objective 1.364266996602e+00, best bound 1.364266996602e+00, gap 0.0000%
```

[8]: [<matplotlib.lines.Line2D at 0x12446e6d0>]

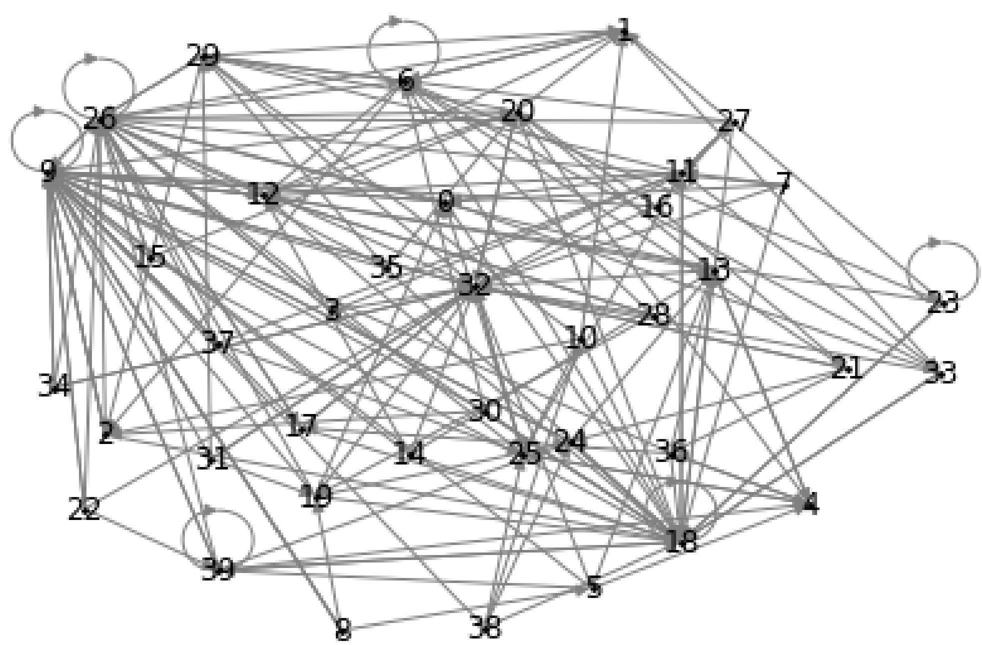


In fact there is a huge amount of non-linear functions you can optimize (denoted by "Gen" as general) and the full list can be found here. More general information about constraints is available here.

4 3) GUROBI Solution pool

When tackling a real-world problem, generally, we only care about finding a solution, in some cases requiring an optimal one regarding some objective. However, we usually don't care about suboptimal or other alternatives as long as the problem is solved. But what if you need to get alternative solutions to your problem because there are other preferences that cannot be easily embedded into the model, so you want to have the possibility to choose? Or what if the best solution can fail in real-life execution or become unavailable because of unforeseen events, so you want to have backup alternatives? We can use the Gurobi solution pool to get more solutions to the problem. We will demonstrate it on the shortest path problem using flows-like logic formulation since finding multiple best alternatives for the shortest path problem is something which is not implicitly provided by the shortest path algorithms we discussed.

[9]:
```
# Create random k-out-regular multidigraph representing some network of one-way roads
dg = nx.DiGraph()
dg = nx.random_k_out_graph(40, 5, 0.4, seed=22)  # 40 vertices

# Draw graph
pos = nx.spring_layout(dg)
nx.draw(dg, pos, node_color='k', node_size=3, edge_color='grey', with_labels=True)
```

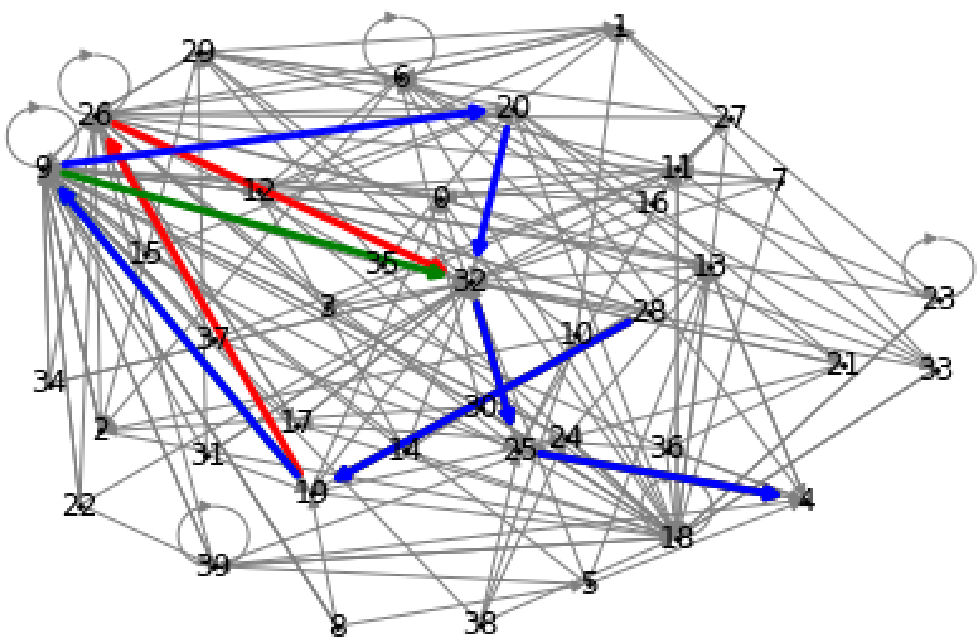


[10]:
```
s = 28  # start
t = 4  # target

# Get edges and vertices
E = dg.edges()
E = dict.fromkeys(E)
V = dg.nodes()

np.random.seed(69)
w = np.random.randint(0, 50, len(E))  # weight for edges

m = g.Model()
x = m.addVars(E.keys(), vtype=g.GRB.BINARY, ub=1, obj=w)  # Variables␣
 ↪representing the edges with their weight given by "w" array adding into␣
 ↪objective

m.addConstr(g.quicksum([x[s, j] for k, j in E if k == s]) == 1)  # Exactly one␣
 ↪edge going from start must be selected.
m.addConstr(g.quicksum([x[i, t] for i, k in E if k == t]) == 1)  # Exactly one␣
 ↪edge going into target must be selected.

# For every vertex except start and target we want the number of selected␣
 ↪outgoing edges to be equal to incoming edges
for i in V:
    if i not in [s, t]:
        m.addConstr(g.quicksum([x[i, j] for k, j in E if k == i]) == g.
 ↪quicksum([x[j, i] for j, k in E if k == i]))

m.setParam(g.GRB.Param.PoolSolutions, 3)  # How many (best) solutions we want␣
 ↪to find
m.setParam(g.GRB.Param.PoolSearchMode, 2)  # This says that when optimum is␣
 ↪found in the search tree, we will still keep looking for k-best solutions
m.optimize()

sols = [0] * m.solcount
colorlist = ['r', 'g', 'b']

print('Found {} solutions.'.format(m.solcount))
for sol_idx in range(m.solcount):
    print('Sol no. {}'.format(sol_idx + 1))
    sols[sol_idx] = []
    m.setParam(g.GRB.Param.SolutionNumber, sol_idx)
    for i, j in E:
        if x[i, j].xn > 0.5:
            print(i, j)
            sols[sol_idx] += [(i, j)]
```

```python
# Draw graph and shortest paths
nx.draw(dg, pos, node_color='k', node_size=3, edge_color='grey',␣
↪with_labels=True)
for k in range(m.solcount):
    nx.draw_networkx_edges(dg, pos, edgelist=sols[k], edge_color=colorlist[k],␣
↪width=3)
```

```
Changed value of parameter PoolSolutions to 3
   Prev: 10  Min: 1  Max: 2000000000  Default: 10
Changed value of parameter PoolSearchMode to 2
   Prev: 0  Min: 0  Max: 2  Default: 0
Gurobi Optimizer version 9.1.2 build v9.1.2rc0 (mac64)
Thread count: 6 physical cores, 12 logical processors, using up to 12 threads
Optimize a model with 40 rows, 174 columns and 331 nonzeros
Model fingerprint: 0x0bb97d12
Variable types: 0 continuous, 174 integer (174 binary)
Coefficient statistics:
  Matrix range     [1e+00, 1e+00]
  Objective range  [1e+00, 5e+01]
  Bounds range     [1e+00, 1e+00]
  RHS range        [1e+00, 1e+00]
Found heuristic solution: objective 314.0000000
Presolve removed 17 rows and 73 columns
Presolve time: 0.00s
Presolved: 23 rows, 101 columns, 185 nonzeros
Variable types: 0 continuous, 101 integer (101 binary)
Found heuristic solution: objective 43.0000000

Root relaxation: objective 4.100000e+01, 13 iterations, 0.00 seconds

    Nodes    |    Current Node    |     Objective Bounds      |     Work
 Expl Unexpl |  Obj  Depth IntInf | Incumbent    BestBd   Gap | It/Node Time

*    0     0               0      41.0000000   41.00000  0.00%     -    0s

Optimal solution found at node 0 - now completing solution pool...

    Nodes    |    Current Node    |      Pool Obj. Bounds     |     Work
             |                    |   Worst                   |
 Expl Unexpl |  Obj  Depth IntInf | Incumbent    BestBd   Gap | It/Node Time

     0     0          -    0         43.00000   41.00000  4.65%     -    0s
     0     0          -    0         43.00000   41.00000  4.65%     -    0s
     0     2          -    0         43.00000   41.00000  4.65%     -    0s

Explored 80 nodes (76 simplex iterations) in 0.03 seconds
Thread count was 12 (of 12 available processors)
```

```
Solution count 3: 41 42 42
No other solutions better than 42

Optimal solution found (tolerance 1.00e-04)
Best objective 4.100000000000e+01, best bound 4.100000000000e+01, gap 0.0000%
Found 3 solutions.
Sol no. 1
19 26
25 4
26 32
28 19
32 25
Sol no. 2
9 32
19 9
25 4
28 19
32 25
Sol no. 3
9 20
19 9
20 32
25 4
28 19
32 25
```



5 THEORETICAL INTERMEZZO - Introducing Constraint Programming

The CP is one of the alternatives to ILP in the sense that it provides a framework for solving NP problems. While the CP model excels at some problems (typically problems involving Cmax like objectives), it does perform poorly on more complex objectives like sums. Thus, it is usually a good idea to first consider the type of the problem before rushing into its formulation, and if scalability and speed are a goal, formulating the problem in multiple ways might also be a good idea.

For our purposes we will use CP Optimizer by IBM which is also free for academic purposes accessible here.

6 4) 1|r, đ|Cmax revisited - Different approaches to the same problem

When we started the semester, we formalized problem 1|r, đ|Cmax (minimizing time of one machine executing tasks with release and deadline times) using MILP programming. We called it "Catering problem". However, we noticed that the scalability of the approach is not so good. Thus, recently you were asked to program Bratley's algorithm solving the same, dramatically improving the scalability but in exchange forcing you to write one purpose piece of code. But what happens when we apply CP? First let's initiate the problem:

[11]:
```python
# Visualization
def plot_solution(s, p):
    """
    s: solution vector
    p: processing times
    """
    fig = plt.figure(figsize=(10, 2))
    ax = plt.gca()
    ax.set_xlabel('time')
    ax.grid(True)
    ax.set_yticks([2.5])
    ax.set_yticklabels(["oven"])
    eps = 0.25  # just to show spaces between the dishes
    ax.broken_barh([(s[i], p[i] - eps) for i in range(len(s))], (0, 5),
                   facecolors=('tab:orange', 'tab:green', 'tab:red', 'tab:
 ↪blue', 'tab:gray'))
```

[12]:
```python
# You can either load one of the provided instances
path = "./bratley_data/instances/test.txt"

with open(path, "r") as f_in:
    lines = f_in.readlines()
```

```
    n = int(lines[0].strip())
    r, d, p = [], [], []

    for i in range(n):
        (pi, ri, di) = list(map(int, lines[1 + i].split()))
        r.append(ri)
        p.append(pi)
        d.append(di)

print("r", r, "d", d, "p", p, sep="\n")
```

```
r
[0, 16, 82, 85, 99, 102, 120, 128, 146, 171, 181, 203, 227, 239, 251, 262, 274,
276, 277, 279, 306, 307, 323, 326, 327, 350, 364, 448, 457, 462, 488, 498, 532,
568, 581, 588, 589, 600, 602, 632]
d
[2892, 3397, 5375, 2624, 4367, 8087, 2947, 10234, 1373, 737, 330, 11679, 4897,
1400, 3918, 11945, 9839, 918, 2004, 1923, 4202, 6561, 2203, 4694, 3710, 11943,
1474, 472, 855, 3012, 3656, 4438, 10359, 1681, 3114, 12961, 2734, 720, 11207,
5065]
p
[73, 13, 6, 1, 29, 28, 72, 76, 86, 48, 94, 18, 32, 24, 33, 63, 11, 16, 69, 40,
38, 20, 45, 78, 61, 30, 80, 16, 57, 50, 2, 32, 97, 86, 27, 35, 76, 51, 66, 54]
```

[ ]:
```
# Or you can also use this data generator for more experimenting
if True:
    from numpy import random as rnd
    import numpy as np

    rnd.seed(15)

    n = 500
    p = [rnd.randint(1, 100) for i in range(n)]

    r = [0 for i in range(n)]
    for i in range(1, n):
        r[i] = int(round(r[i - 1] + rnd.exponential(0.5 * sum(p) / len(p))))

    d = [int(round(r[i] + p[i] + rnd.exponential(100 * sum(p) / len(p)))) for i␣
 ↪in range(n)]
    print("r", r, "d", d, "p", p, sep="\n")
```

6.1 ILP model

Let's start with the ILP model. The model should be nothing new for you since you programmed it at the start of the semester. We will execute the model on the loaded instance and see how well it will do.

[ ]:
```
m = g.Model()

# - add variables
s = m.addVars(n, vtype=g.GRB.CONTINUOUS, lb=0)
x = {}
for i in range(n):
    for j in range(i + 1, n):
        x[i, j] = m.addVar(vtype=g.GRB.BINARY)

Cmax = m.addVar(vtype=g.GRB.CONTINUOUS, obj=1)

# - add constraints
for i in range(n):
    m.addConstr(s[i] + p[i] <= Cmax)
    m.addConstr(s[i] >= r[i])
    m.addConstr(s[i] + p[i] <= d[i])

M = max(d)
for i in range(n):
    for j in range(i + 1, n):
        m.addConstr(s[i] + p[i] <= s[j] + M * (1 - x[i, j]))
        m.addConstr(s[j] + p[j] <= s[i] + M * x[i, j])

m.params.TimeLimit = 15

# call the solver -----------------------------------------------
m.optimize()

print()
if m.SolCount > 0:
    starts = [s[i].X for i in range(n)]
else:
    print("No solution was found.")

print("Done")
```

[ ]:
```
plot_solution(starts, p)
```

6.2 CP model

Now let's take a look at CP model. As we said above, in CP, we don't use only mathematical constraints. We also have a collection of expressions and constraints which we can use. In fact, using CP Optimizer, we could program our own constraints and expressions and use them as well, but this is far beyond the scope of this demonstration. Let's take a look at what we will use today: - Interval variable: An interval variable represents some task/activity spanning in the final solution. Its size is given by the instance description, while its start (and thus also end) in the solution is found by the CP solver. - Start_of predicate: This predicate points us to the point in the solution where the interval variable starts. Thus, using this predicate, we can, for example, enforce the release time of the activity by setting the Start_of larger or equal to the given release time of the activity. - End_of predicate: Same as the predicate above but for the end of the activity in the solution. - Min/Max/Pow/Log...: In CP, often used mathematical functions are directly accessible, and we do not need to think about how to encode them in a different way. - Sequence_var: We can think of Sequence_var as a wrapper for interval variables saying that they are part of the same sequence of activities. Then we can apply constraints working with this wrapper. - No_overlap: If applied on Sequence_var, we enforce that all the tasks in the sequence must not overlap. Thus, we can think of it as creating a chain of activities on one machine.

[13]:
```
import docplex.cp.model as cp
from docplex.cp.model import CpoModel

# Create model
m = CpoModel()

# - add variables
tasks = [m.interval_var(name="task{:d}".format(i), optional=False, size=p[i])␣
 ↪for i in range(n)]
seq = m.sequence_var(tasks, name='seq')

# - set objective
m.add(m.minimize(m.max([m.end_of(tasks[i]) for i in range(n)])))  # minimize␣
 ↪C_max

# - add constraints
for i in range(n):
    m.add(m.start_of(tasks[i]) >= r[i])  # release time
    m.add(m.end_of(tasks[i]) <= d[i])  # deadline

m.add(m.no_overlap(seq))  # one task executed at one time

# Solve the model
msol = m.solve(TimeLimit=10, LogVerbosity="Normal", LogPeriod=1, Workers=1)

# Print the solution
print()
if msol.is_solution():
    starts = [msol.get_value(tasks[i])[0] for i in range(n)]
    print(*starts, sep="\n")
else:
    print("No solution found.")

print("Done")
```

```
! --------------------------------------------------- CP Optimizer 22.1.0.0 --
! Minimization problem - 41 variables, 81 constraints
! TimeLimit            = 10
! Workers              = 1
```

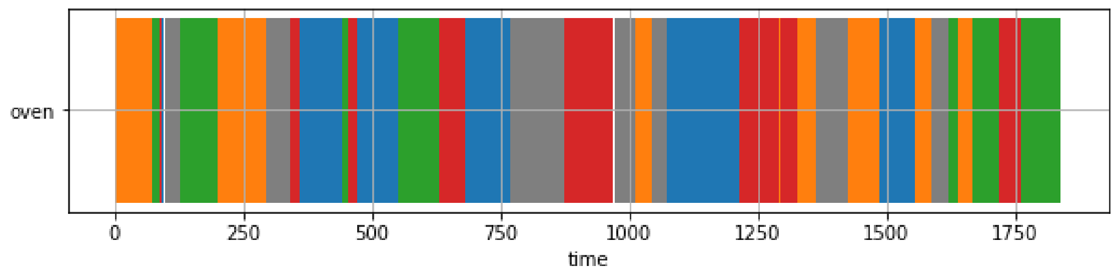
```
! LogPeriod             = 1
! Initial process time : 0.02s (0.02s extraction + 0.00s propagation)
!  . Log search space  : 210.7 (before), 210.7 (after)
!  . Memory usage      : 478.8 kB (before), 478.8 kB (after)
! Using sequential search.
! ----------------------------------------------------------------------------
!          Best Branches  Non-fixed            Branch decision
                      0         41                  -
+ New bound is 686
                      1         41                  -
                     81         41           1787  = startOf(task39)
           1841      81         41                  -
*          1841      81  0.03s                (gap is 62.74%)
           1841      99         29        F   398  = startOf(task9)
           1841     157         10        F  1752  = startOf(task31)
           1841     173         31        F   398  = startOf(task9)
           1841     179         34        F   275  = startOf(task16)
           1841     205         24        F  1323  = startOf(task24)
           1841     231         23        F   880  = startOf(task20)
           1841     257         23        F   542  = startOf(task12)
           1841     263         34        F   275  = startOf(task16)
           1841     275         34        F   181  = startOf(task10)
           1841     276         41                  -
           1841     277         40              0  = startOf(task0)
           1841     278         39             73  = startOf(task1)
           1841     279         38             86  = startOf(task2)
           1841     280         37             92  = startOf(task3)
           1841     281         36             99  = startOf(task4)
! Time = 0.03s, Average fail depth = 3, Memory usage = 962.5 kB
! Current bound is 686 (gap is 62.74%)
!          Best Branches  Non-fixed            Branch decision
           1841     282         35            128  = startOf(task6)
           1841     283         34            200  = startOf(task10)
           1841     284         33            294  = startOf(task9)
           1841     285         32            342  = startOf(task17)
           1841     286         31            358  = startOf(task8)
           1841     287         30            444  = startOf(task16)
           1841     288         29            455  = startOf(task27)
           1841     289         28            471  = startOf(task28)
           1841     290         27            528  = startOf(task13)
           1841     291         26            552  = startOf(task26)
           1841     292         25            632  = startOf(task37)
           1841     293         24            683  = startOf(task33)
           1841     294         23            769  = startOf(task39)
           1841     295         22            823  = startOf(task29)
           1841     296         21            873  = startOf(task32)
           1841     297         20            970  = startOf(task19)
           1841     298         19           1010  = startOf(task35)
```

```
          1841      299         18          1045  = startOf(task34)
          1841      300         17          1072  = startOf(task38)
          1841      301         16          1138  = startOf(task23)
! Time = 0.03s, Average fail depth = 3, Memory usage = 994.5 kB
! Current bound is 686 (gap is 62.74%)
!         Best Branches  Non-fixed            Branch decision
          1841      302         15          1216  = startOf(task7)
          1841      303         14          1292  = startOf(task30)
          1841      304         13          1294  = startOf(task12)
          1841      305         12          1326  = startOf(task20)
          1841      306         11          1364  = startOf(task24)
          1841      307         10          1425  = startOf(task15)
          1841      308          9          1488  = startOf(task18)
          1841      309          8          1557  = startOf(task25)
          1841      310          7          1587  = startOf(task14)
          1841      311          6          1620  = startOf(task21)
          1841      312          5          1640  = startOf(task5)
          1841      313          4          1668  = startOf(task11)
          1841      314          3          1686  = startOf(task31)
          1841      315          1          1718  = startOf(task22)
          1841      316         41        F 1763  = startOf(task36)
          1841      317          1        F         -
          1839      317         41                  -
*         1839      317  0.03s              (gap is 62.70%)
          1839      318         40             0  = startOf(task0)
          1839      319         39            73  = startOf(task1)
! Time = 0.03s, Average fail depth = 5, Memory usage = 1.0 MB
! Current bound is 686 (gap is 62.70%)
!         Best Branches  Non-fixed            Branch decision
          1839      320         38        F   86  = startOf(task2)
          1839      321         39        F         -
          1839      322         40             0  = startOf(task0)
          1839      323         39            73  = startOf(task1)
          1839      324         38           181  = startOf(task10)
          1839      325         37           448  = startOf(task27)
          1839      326         36           600  = startOf(task37)
          1839      327         35           275  = startOf(task9)
          1839      328         34           464  = startOf(task28)
          1839      329         33           323  = startOf(task17)
          1839      330         32           339  = startOf(task8)
          1839      331         31           521  = startOf(task13)
          1839      332         30           651  = startOf(task26)
          1839      333         29           731  = startOf(task33)
          1839      334         28           817  = startOf(task32)
          1839      335         27           914  = startOf(task23)
          1839      336         26           992  = startOf(task7)
          1839      337         25          1068  = startOf(task36)
          1839      338         24          1144  = startOf(task6)
```

```
         1839      339         23           1216  = startOf(task18)
! Time = 0.03s, Average fail depth = 5, Memory usage = 1.0 MB
! Current bound is 686 (gap is 62.70%)
!          Best Branches  Non-fixed            Branch decision
         1839      340         22           1285  = startOf(task38)
         1839      341         21           1351  = startOf(task15)
         1839      342         20           1414  = startOf(task24)
         1839      343         19           1475  = startOf(task39)
         1839      344         18            545  = startOf(task29)
         1839      345         17           1529  = startOf(task22)
         1839      346         16           1574  = startOf(task19)
         1839      347         15           1614  = startOf(task20)
         1839      348         14           1652  = startOf(task35)
         1839      349         13           1687  = startOf(task14)
         1839      350         12           1720  = startOf(task31)
         1839      351         10        F  1752  = startOf(task12)
         1839      352         12        F        -
         1839      353         40              0  = startOf(task0)
         1839      354         39             73  = startOf(task1)
         1839      355         38        F    86  = startOf(task2)
         1839      356         39        F        -
         1839      357         40              0  = startOf(task0)
         1839      358         39             73  = startOf(task1)
         1839      359         38        F    86  = startOf(task2)
! Time = 0.03s, Average fail depth = 5, Memory usage = 1.0 MB
! Current bound is 686 (gap is 62.70%)
!          Best Branches  Non-fixed            Branch decision
         1839      360         39        F        -
         1839      361         40              0  = startOf(task0)
         1839      362         39             73  = startOf(task1)
         1839      363         38        F    86  = startOf(task2)
         1839      364         39             86 != startOf(task2)
         1839      365         38        F    86  = startOf(task3)
         1839      366         39        F          !presenceOf(task2)
         1839      367         40        F          !presenceOf(task1)
         1839      368         41        F          !presenceOf(task0)
         1839      368         41        F        -
+ New bound is 1839 (gap is 0.00%)
         1839      369         41        F        -
! ----------------------------------------------------------------------------
! Search completed, 2 solutions found.
! Best objective         : 1839 (optimal - effective tol. is 0)
! Best bound             : 1839
! ----------------------------------------------------------------------------
! Number of branches     : 369
! Number of fails        : 20
! Total memory usage     : 1.1 MB (1.0 MB CP Optimizer + 0.1 MB Concert)
! Time spent in solve    : 0.03s (0.01s engine + 0.02s extraction)
```

```
 ! Search speed (br. / s) : 36900.0
 ! ----------------------------------------------------------------------------

0
73
86
92
99
1640
128
1216
358
294
200
1668
1294
528
1587
1425
444
342
1488
970
1326
1620
1718
1138
1364
1557
552
455
471
823
1292
1686
873
683
1045
1010
1763
632
1072
769
Done
```

[14]:
```
plot_solution(starts, p)
```



7 5) Travelling Salesman Problem revisited - Different approaches to the same problem 2

We showed that CP scales incredibly well in 1|r, đ|Cmax problem, but it is a perfect problem for it. We can try to solve a problem that might be less suitable for it. We pick the Travelling Salesman Problem (TSP) as it was discussed in-depth in lectures. First let us build initial structure for the problem:

[18]:
```
Point = namedtuple("Point", ['x', 'y'])


def length(point1, point2):
    return int(round(math.sqrt((point1.x - point2.x) ** 2 + (point1.y - point2.
 ↪y) ** 2)))  # CP works with int only


class TSP:
    def __init__(self):
        D = None  # distance matrix
        points = None  # vertices

    def load_instance(self, path):
        input_data_file = open(path, 'r')
        input_data = ''.join(input_data_file.readlines())

        # parse the input
        lines = input_data.split('\n')
        nodeCount = int(lines[0])

        points = []
        for i in range(1, nodeCount + 1):
            parts = lines[i].split()
            points.append(Point(float(parts[0]), float(parts[1])))

        # distance matrix
```

```
        D = [[0 for _ in range(nodeCount + 1)] for _ in range(nodeCount + 1)] ␣
↪# Add dummy vertex last
        for i in range(nodeCount):
            for j in range(nodeCount):
                D[i][j] = length(points[i], points[j])

        # the last vertex is the same as the first one
        for i in range(nodeCount):
            D[i][-1] = D[i][0]
            D[-1][i] = D[0][i]

        self.D = D
        self.points = points

        return self
```

[19]:
```
instances = [
    {"inst": TSP().load_instance("./tsp_data/tsp_5_1"),
     "init": [0, 1, 2, 4, 3]},
    {"inst": TSP().load_instance("./tsp_data/tsp_51_1"),
     "init": [0, 5, 2, 28, 10, 9, 45, 3, 46, 8, 4, 35, 13, 7, 19, 40, 18, 11,␣
↪42, 37, 20, 25, 1, 31, 22, 48, 32, 17, 49,
              39, 50, 38, 15, 44, 14, 16, 29, 43, 21, 30, 12, 23, 34, 24, 41,␣
↪27, 36, 6, 26, 47, 33]},
    {"inst": TSP().load_instance("./tsp_data/tsp_70_1"),
     "init": [0, 35, 50, 11, 57, 2, 56, 27, 21, 49, 58, 53, 41, 36, 38, 52, 6,␣
↪5, 7, 51, 55, 68, 46, 67, 24, 16, 44, 39,
              22, 1, 14, 15, 20, 29, 28, 45, 12, 31, 18, 26, 3, 59, 9, 25, 4,␣
↪10, 61, 43, 32, 8, 64, 54, 48, 62, 13, 19,
              60, 42, 37, 66, 40, 17, 30, 23, 69, 33, 65, 34, 47, 63]},
    {"inst": TSP().load_instance("./tsp_data/tsp_100_1"),
     "init": [0, 6, 69, 61, 76, 35, 84, 11, 9, 26, 72, 47, 40, 94, 81, 60, 64,␣
↪66, 8, 23, 70, 59, 33, 67, 43, 37, 65,
              71, 19, 15, 75, 14, 53, 46, 5, 29, 80, 38, 91, 57, 41, 50, 12,␣
↪55, 98, 39, 24, 68, 2, 28, 73, 87, 48, 85,
              21, 96, 42, 77, 16, 7, 10, 74, 30, 18, 17, 34, 22, 99, 93, 51, 3,␣
↪89, 13, 31, 44, 62, 25, 82, 86, 54, 1,
              27, 45, 88, 79, 97, 49, 90, 20, 63, 52, 92, 95, 78, 83, 32, 4,␣
↪56, 58, 36]},
    {"inst": TSP().load_instance("./tsp_data/tsp_200_1"),
     "init": [0, 103, 62, 192, 5, 48, 89, 148, 117, 9, 128, 83, 136, 23, 37,␣
↪108, 177, 181, 98, 106, 35, 160, 125, 131,
              123, 58, 73, 20, 145, 71, 111, 46, 97, 22, 114, 112, 178, 59, 61,␣
↪163, 119, 154, 141, 34, 85, 26, 11, 19,
              146, 130, 166, 76, 164, 179, 60, 24, 80, 101, 134, 68, 167, 129,␣
↪188, 158, 102, 172, 88, 168, 41, 30, 79,
```

```
              55, 199, 132, 144, 96, 180, 196, 3, 64, 65, 195, 25, 186, 151,␣
↪110, 183, 147, 69, 21, 15, 87, 143, 162,
              93, 150, 115, 17, 78, 52, 165, 18, 191, 198, 118, 109, 74, 135,␣
↪156, 173, 7, 113, 91, 159, 57, 176, 50,
              86, 56, 6, 8, 105, 153, 174, 82, 54, 107, 121, 33, 28, 45, 116,␣
↪124, 133, 189, 42, 2, 13, 197, 157, 40,
              70, 99, 187, 47, 127, 138, 137, 170, 29, 171, 182, 161, 84, 67,␣
↪72, 122, 49, 43, 169, 175, 190, 193, 194,
              149, 38, 185, 95, 155, 51, 77, 104, 4, 142, 36, 32, 75, 12, 94,␣
↪81, 1, 63, 39, 120, 53, 140, 66, 27, 92,
              126, 90, 44, 184, 31, 100, 152, 14, 16, 10, 139]}
]
```

[20]:
```
inst_id = 4  # Which instance to pick
inst = instances[inst_id]["inst"]
init_order = instances[inst_id]["init"]
```

[21]:
```
INITIALIZE = False  # If we want to provide the "init" warm start to the Gurobi␣
↪solver
```

7.1 ILP model

This ILP formulation was discussed in the lectures, so it should not be completely new to you.

[22]:
```
nodeCount = len(inst.points)
points = inst.points

# Create model
m = g.Model("tsp")

# - add variables
x = m.addVars(nodeCount, nodeCount, vtype=g.GRB.BINARY, name="x")
u = m.addVars(nodeCount, vtype=g.GRB.INTEGER, lb=0, name="u")

# - set objective
obj = g.quicksum(g.quicksum(inst.D[i][j] * x[i, j] for j in range(nodeCount))␣
↪for i in range(nodeCount))
m.setObjective(obj, g.GRB.MINIMIZE)

# - add constraints
m.addConstrs((1 == g.quicksum(x[i, j] for j in range(nodeCount)) for i in␣
↪range(nodeCount)))
m.addConstrs((1 == g.quicksum(x[j, i] for j in range(nodeCount)) for i in␣
↪range(nodeCount)))

for i in range(1, nodeCount):
    for j in range(1, nodeCount):
```

```python
        m.addConstr(u[i] - u[j] + 1 <= nodeCount * (1 - x[i, j]))

# Initialization
if INITIALIZE:
    for i, order in enumerate(init_order):
        u[order].start = i

m.Params.TimeLimit = 10
m.optimize()

# Print the solution
print()
if m.SolCount > 0:
    obj = m.objVal
    print("Objective {}".format(obj))

    order = [u[i].X for i in range(nodeCount)]
    indices = range(nodeCount)
    s = sorted(zip(order, indices), key=lambda x: x[0])
    print([x[1] for x in s])
else:
    print("No solution was found.")

print("Done")
```

```
Changed value of parameter TimeLimit to 10.0
   Prev: inf  Min: 0.0  Max: inf  Default: inf
Gurobi Optimizer version 9.1.2 build v9.1.2rc0 (mac64)
Thread count: 6 physical cores, 12 logical processors, using up to 12 threads
Optimize a model with 40001 rows, 40200 columns and 198405 nonzeros
Model fingerprint: 0xceefc9da
Variable types: 0 continuous, 40200 integer (40000 binary)
Coefficient statistics:
  Matrix range     [1e+00, 2e+02]
  Objective range  [1e+01, 4e+03]
  Bounds range     [1e+00, 1e+00]
  RHS range        [1e+00, 2e+02]
Presolve removed 199 rows and 200 columns
Presolve time: 0.31s
Presolved: 39802 rows, 40000 columns, 197808 nonzeros
Variable types: 0 continuous, 40000 integer (39801 binary)

Deterministic concurrent LP optimizer: primal and dual simplex
Showing first log only…

Concurrent spin time: 0.00s

Solved with dual simplex
```

```
Root relaxation: objective 2.312896e+04, 677 iterations, 0.18 seconds

    Nodes    |    Current Node    |     Objective Bounds      |     Work
 Expl Unexpl |  Obj  Depth IntInf | Incumbent    BestBd   Gap | It/Node Time

     0     0 23128.9600    0  411          - 23128.9600      -     -    2s

Explored 1 nodes (772 simplex iterations) in 10.92 seconds
Thread count was 12 (of 12 available processors)

Solution count 0

Time limit reached
Best objective -, best bound 2.312900000000e+04, gap -

No solution was found.
Done
```

7.2 CP model

We already explained most of the CP concepts used in the previous example. There are only two new constraints used, First and Last. These constraints ensure, that for a given sequence and interval_var, interval_var will be either the first one or the last one in the sequence. We use this to ensure that the travelling salesman starts and ends in the same city.

[23]:
```
import docplex.cp.model as cp
from docplex.cp.model import CpoModel

node_count = len(inst.points)
points = inst.points

# Create model
m = CpoModel()

# - add variables
cities = [m.interval_var(name="city{:d}".format(i), optional=False, size=1) for␣
 ↪i in range(node_count + 1)]
seq = m.sequence_var(cities, name='seq', types=([i for i in range(node_count)]␣
 ↪+ [0]))

# - set objective
m.add(m.minimize(m.max([m.end_of(cities[i]) for i in range(len(cities))]) -␣
 ↪len(cities)))

# - add constraints
m.add(m.first(seq, cities[0]))  # start from city 0
m.add(m.last(seq, cities[-1]))  # repeat the same city last
```

```python
m.add(m.no_overlap(seq, inst.D, True))

# Solve the model
msol = m.solve(TimeLimit=10, LogVerbosity="Terse", LogPeriod=1000, Workers=1)

# Print the solution
print()
if msol.is_solution():

    ovals = msol.get_objective_values()
    print("Objective {}".format(ovals[0]))

    starts = [msol.get_value(cities[i])[0] for i in range(len(cities) - 1)]
    indices = range(len(cities) - 1)
    s = sorted(zip(starts, indices), key=lambda x: x[0])

    print([x[1] for x in s])
else:
    print("No solution found.")

print("Done")
```

```
! --------------------------------------------------- CP Optimizer 22.1.0.0 --
! Minimization problem - 202 variables, 3 constraints
! TimeLimit            = 10
! Workers              = 1
! LogVerbosity         = Terse
! Initial process time : 0.03s (0.03s extraction + 0.00s propagation)
!  . Log search space  : 1537.9 (before), 1537.9 (after)
!  . Memory usage      : 1.5 MB (before), 1.5 MB (after)
! Using sequential search.
! ----------------------------------------------------------------------------
!          Best Branches  Non-fixed            Branch decision
                        0        202                 -
+ New bound is 19347
*        230214      403  0.05s               (gap is 91.60%)
*        228128     1341  0.07s               (gap is 91.52%)
*         36858     2516  0.10s               (gap is 47.51%)
*         36759     3349  0.13s               (gap is 47.37%)
          36759     9314          6         F        -
+ New bound is 19451 (gap is 47.09%)
*         36648    20328  0.33s               (gap is 46.92%)
*         36630    21012  0.36s               (gap is 46.90%)
*         36478    21729  0.39s               (gap is 46.68%)
*         36436    22456  0.42s               (gap is 46.62%)
*         36403    26838  0.54s               (gap is 46.57%)
*         36401    29239  0.58s               (gap is 46.56%)
*         36296    30086  0.60s               (gap is 46.41%)
```

```
*        36243    34528  0.82s               (gap is 46.33%)
*        36169    38446  0.93s               (gap is 46.22%)
*        36128    40406  1.00s               (gap is 46.16%)
*        36093    40779  1.02s               (gap is 46.11%)
*        36038    42294  1.12s               (gap is 46.03%)
*        35989    46657  1.21s               (gap is 45.95%)
*        35976    46883  1.21s               (gap is 45.93%)
! Time = 1.21s, Average fail depth = 140, Memory usage = 3.8 MB
! Current bound is 19451 (gap is 45.93%)
!          Best Branches  Non-fixed            Branch decision
*        35753    47734  1.24s               (gap is 45.60%)
*        35686    49696  1.28s               (gap is 45.49%)
*        35660    52046  1.33s               (gap is 45.45%)
*        35575    54489  1.41s               (gap is 45.32%)
*        35413    55232  1.42s               (gap is 45.07%)
*        35367    56032  1.45s               (gap is 45.00%)
*        35334    59987  1.55s               (gap is 44.95%)
*        35148    63571  1.62s               (gap is 44.66%)
*        34982    66568  1.72s               (gap is 44.40%)
*        34975    67903  1.79s               (gap is 44.39%)
*        34953    75883  2.09s               (gap is 44.35%)
*        34832    76662  2.11s               (gap is 44.16%)
*        34488    78594  2.18s               (gap is 43.60%)
*        34480    81746  2.27s               (gap is 43.59%)
*        34358    83524  2.34s               (gap is 43.39%)
*        34326    85762  2.41s               (gap is 43.33%)
*        34283    86776  2.50s               (gap is 43.26%)
*        34150    91104  2.62s               (gap is 43.04%)
*        34090    93490  2.72s               (gap is 42.94%)
*        34035    93737  2.74s               (gap is 42.85%)
! Time = 2.74s, Average fail depth = 124, Memory usage = 3.8 MB
! Current bound is 19451 (gap is 42.85%)
!          Best Branches  Non-fixed            Branch decision
*        33981    95188  2.79s               (gap is 42.76%)
*        33976    95434  2.80s               (gap is 42.75%)
*        33921     120k  3.03s               (gap is 42.66%)
*        33898     121k  3.06s               (gap is 42.62%)
*        33506     122k  3.09s               (gap is 41.95%)
*        33457     123k  3.13s               (gap is 41.86%)
*        33280     123k  3.14s               (gap is 41.55%)
*        33147     124k  3.21s               (gap is 41.32%)
*        33130     133k  3.66s               (gap is 41.29%)
*        33104     136k  3.75s               (gap is 41.24%)
*        33028     139k  3.84s               (gap is 41.11%)
*        32885     141k  3.90s               (gap is 40.85%)
*        32850     141k  3.91s               (gap is 40.79%)
*        32830     144k  4.03s               (gap is 40.75%)
*        32772     198k  4.46s               (gap is 40.65%)
```

```
*         32746     205k  4.75s               (gap is 40.60%)
*         32733     215k  5.26s               (gap is 40.58%)
*         32727     219k  5.36s               (gap is 40.57%)
*         32680     224k  5.50s               (gap is 40.48%)
*         32678     233k  5.81s               (gap is 40.48%)
! Time = 5.81s, Average fail depth = 139, Memory usage = 3.9 MB
! Current bound is 19451 (gap is 40.48%)
!          Best Branches  Non-fixed            Branch decision
*         32645     234k  5.82s               (gap is 40.42%)
*         32633     235k  5.88s               (gap is 40.39%)
*         32506     239k  6.12s               (gap is 40.16%)
*         32480     250k  6.39s               (gap is 40.11%)
*         32349     268k  7.04s               (gap is 39.87%)
*         32341     268k  7.07s               (gap is 39.86%)
*         32257     272k  7.24s               (gap is 39.70%)
*         32233     272k  7.25s               (gap is 39.66%)
*         32219     273k  7.29s               (gap is 39.63%)
*         32109     275k  7.37s               (gap is 39.42%)
*         32012     275k  7.39s               (gap is 39.24%)
*         31932     276k  7.41s               (gap is 39.09%)
*         31907     277k  7.49s               (gap is 39.04%)
*         31889     277k  7.51s               (gap is 39.00%)
*         31877     281k  7.67s               (gap is 38.98%)
*         31705     284k  7.82s               (gap is 38.65%)
*         31702     285k  7.84s               (gap is 38.64%)
*         31672     286k  7.90s               (gap is 38.59%)
*         31577     287k  7.92s               (gap is 38.40%)
*         31501     299k  8.43s               (gap is 38.25%)
! Time = 8.43s, Average fail depth = 132, Memory usage = 3.9 MB
! Current bound is 19451 (gap is 38.25%)
!          Best Branches  Non-fixed            Branch decision
*         31497     304k  8.56s               (gap is 38.24%)
*         31470     309k  8.74s               (gap is 38.19%)
*         31467     311k  8.85s               (gap is 38.19%)
*         31448     312k  8.89s               (gap is 38.15%)
*         31407     313k  8.95s               (gap is 38.07%)
*         31380     316k  9.08s               (gap is 38.01%)
*         31363     316k  9.08s               (gap is 37.98%)
*         31348     319k  9.18s               (gap is 37.95%)
! ----------------------------------------------------------------------------
! Search terminated by limit, 86 solutions found.
! Best objective         : 31348 (gap is 37.95%)
! Best bound             : 19451
! ----------------------------------------------------------------------------
! Number of branches     : 436836
! Number of fails        : 177795
! Total memory usage     : 4.0 MB (3.9 MB CP Optimizer + 0.1 MB Concert)
! Time spent in solve    : 10.00s (9.98s engine + 0.03s extraction)
```

```
 ! Search speed (br. / s) : 43815.0
 ! ----------------------------------------------------------------------------

Objective 31348
[0, 103, 62, 192, 5, 48, 89, 148, 117, 9, 128, 83, 136, 23, 37, 108, 177, 181,
106, 98, 125, 160, 35, 131, 123, 58, 73, 20, 145, 71, 111, 31, 184, 44, 97, 22,
114, 112, 178, 59, 61, 163, 119, 154, 141, 34, 85, 26, 11, 19, 146, 130, 166,
76, 164, 179, 60, 24, 80, 101, 134, 68, 167, 129, 158, 102, 172, 88, 168, 41,
30, 79, 55, 199, 132, 144, 96, 180, 196, 3, 64, 65, 195, 25, 186, 151, 110, 183,
147, 69, 21, 15, 87, 143, 162, 93, 150, 115, 17, 78, 52, 165, 18, 118, 198, 191,
109, 74, 135, 156, 173, 7, 113, 91, 159, 57, 176, 50, 86, 56, 6, 8, 105, 153,
174, 82, 54, 107, 121, 33, 28, 45, 2, 42, 116, 124, 189, 133, 157, 70, 40, 99,
197, 13, 187, 47, 127, 138, 137, 170, 29, 171, 182, 161, 84, 67, 72, 122, 43,
49, 190, 193, 194, 149, 38, 185, 95, 77, 155, 51, 104, 4, 142, 36, 32, 75, 12,
94, 81, 175, 169, 1, 63, 39, 120, 53, 188, 140, 66, 27, 92, 126, 90, 100, 152,
14, 16, 10, 139, 46]
Done
```

7.3 Comparison

Best objective found by ILP and CP model under 10s timelimit. (without initialization)

instance	ILP	CP
5	3	3
50	496	441
70	994	710
100	-	22247
200	-	31962

Interesting result. So isn't CP actually just superior to ILP? Well, no. In this case, even though TSP is technically a sum of the path travelled by the salesman, we managed to reformulate it as a Cmax problem. Also, the representation of the problem constraints is very natural in this case. However, remember that ILP has one more trick in its sleeve, lazy callbacks. If you ran the lazy constraints model instead, it would beat CP in this case.

8 6) CP Sudoku

Lastly, let's revisit the practical test assignment. While the MILP model was not so hard to formulate, it becomes a trivial problem to formulate using CP.

[24]:
```
def load_data(file_name):
    AB, numbers = {"A": [], "B": []}, {}
    for id_row, line in enumerate(open(file_name).readlines()):
        for id_column, char in enumerate(line):
            if char == "A" or char == "B":
                AB[char].append((id_row, id_column))
```

```
            elif char.isdigit():
                numbers[(id_row, id_column)] = int(char)
    return AB, numbers

variant = "A"
size, rect_c = 9, 3
AB, numbers = load_data("sudoku_data/input_" + variant + ".txt")

# MODEL AND VARIABLES
model = cp.CpoModel()
cells = cp.integer_var_dict([(i, j) for i, j in iter.product(range(size),␣
↪repeat=2)], min=0, max=size - 1)

# BASE SUDOKU CONSTRAINTS
model.add([cp.all_diff([cells[i, j] for j in range(size)]) for i in␣
↪range(size)])  # Rows
model.add([cp.all_diff([cells[j, i] for j in range(size)]) for i in␣
↪range(size)])  # Columns
model.add([cp.all_diff([cells[r_r * rect_c + i, r_c * rect_c + j] for i, j in␣
↪iter.product(range(rect_c), repeat=2)])
           for r_r, r_c in iter.product(range(rect_c), repeat=2)])  # Rectangles
model.add([cells[pos] == val for pos, val in numbers.items()])  # Fixed numbers

# CONDITIONS A/B
for pos in AB["A"]:  # Cond A - The sum of elements in 4-neighborhood of each␣
↪(i,j) from A is an integer multiple of 3
    neighborhood = [e for e in [(pos[0] - 1, pos[1]), (pos[0] + 1, pos[1]),␣
↪(pos[0], pos[1] - 1), (pos[0], pos[1] + 1)]
                    if 0 <= e[0] < size and 0 <= e[1] < size]
    model.add(cp.sum([cells[neigh] for neigh in neighborhood]) % 3 == 0)
for pos in AB["B"]:  # Cond B - The number in (i,j) <= to the min of numbers in␣
↪its north, west and north-west positions
    neighborhood = [e for e in [(pos[0] - 1, pos[1]), (pos[0], pos[1] - 1),␣
↪(pos[0] - 1, pos[1] - 1)]
                    if 0 <= e[0] < size and 0 <= e[1] < size]
    model.add(cells[pos] <= cp.min([cells[neigh] for neigh in neighborhood]))

# OBJECTIVES
if variant == "A":
    model.add(cp.minimize(cp.sum(cells[i, i] for i in range(size))))
elif variant == "B":
    model.add(cp.maximize(cp.sum([cells[0, 0], cells[0, size - 1], cells[size -␣
↪1, 0], cells[size - 1, size - 1],
                                  cells[int((size - 1) / 2), int((size - 1) /␣
↪2)]])))
```

```python
# OUTPUT
sol = model.solve().get_solution()
if sol is None:
    print("-1")
else:
    print(sol.objective_values[0])
    for row, column in iter.product(range(size), repeat=2):
        print(sol.get_var_solution(cells[row, column]).value, end="")
        print("", end="") if column != 8 else print("")
```

```
! --------------------------------------------------- CP Optimizer 22.1.0.0 --
! Minimization problem - 81 variables, 32 constraints
! Initial process time : 0.01s (0.01s extraction + 0.00s propagation)
!  . Log search space  : 243.2 (before), 243.2 (after)
!  . Memory usage      : 336.1 kB (before), 336.1 kB (after)
! Using parallel search with 12 workers.
! ----------------------------------------------------------------------------
!          Best Branches  Non-fixed    W       Branch decision
                        0         81                 -
+ New bound is 8
                        0         79    1            -
+ New bound is 9
                        2         79    1   F     0 != _INT_51
+ New bound is 10
*            33        52  0.04s        1      (gap is 69.70%)
*            26       122  0.04s        1      (gap is 61.54%)
*            25       913  0.04s        1      (gap is 60.00%)
             25      1000          4    1         6  = _INT_53
             25      1000          4    2         7  = _INT_5
*            23       766  0.04s        3      (gap is 56.52%)
*            22       987  0.04s        3      (gap is 54.55%)
             22      1000          4    3         0  = _INT_10
             22      1000          4    4         5 != _INT_34
             22      1000          4    5            -
             22      1000          8    6   F     5  = _INT_46
             22      1000          6    7   F     5 != _INT_54
             22      1000          4    8            -
             22      1000          6    9         1  = _INT_28
             22      1000          4   10         2 != _INT_72
             22      1000          6   11   F     2 != _INT_73
             22      1000          8   12   F     1 != _INT_11
! Time = 0.04s, Average fail depth = 31, Memory usage = 8.6 MB
! Current bound is 10 (gap is 54.55%)
!          Best Branches  Non-fixed    W       Branch decision
*            21      1689  0.05s        1      (gap is 52.38%)
*            19      1842  0.05s        1      (gap is 47.37%)
             19      2000          4    1         1 != _INT_63
*            18      1466  0.05s        3      (gap is 44.44%)
```

```
 *           15     1573  0.05s        3      (gap is 33.33%)
             15     2000          4    3         1 != _INT_23
             15     2000          4    4   F     8  = _INT_4
             15     2000          4    5   F     0  = _INT_16
             15     2000         19    6            -
             15     2000         17    7         6 != _INT_34
             15     2000         16    8   F     6  = _INT_30
             15     2000          6    9         7  = _INT_37
             15     2000          7    2         1  = _INT_24
             15     2000          4   11         3 != _INT_33
             15     2000          4   12   F     2 != _INT_69
             15     2736          4    1            -
 + New bound is 15 (gap is 0.00%)
 ! ----------------------------------------------------------------------------
 ! Search completed, 9 solutions found.
 ! Best objective         : 15 (optimal - effective tol. is 0)
 ! Best bound             : 15
 ! ----------------------------------------------------------------------------
 ! Number of branches     : 32509
 ! Number of fails        : 13641
 ! Total memory usage     : 8.7 MB (8.7 MB CP Optimizer + 0.0 MB Concert)
 ! Time spent in solve    : 0.06s (0.05s engine + 0.01s extraction)
 ! Search speed (br. / s) : 650180.6
 ! ----------------------------------------------------------------------------
15
135240687
607851243
482763015
253084761
874615320
061372854
528407136
746138502
310526478
```

[ ]: