homework3

January 30, 2024

[ ]:
```
import pandas
import numpy
import utils
import seaborn
import networkx as nx
import matplotlib.pyplot as plt
import gurobipy as gp
from gurobipy import GRB
from itertools import product, combinations
!lscpu | head -n 17
```

```
Architecture:                    x86_64
CPU op-mode(s):                  32-bit, 64-bit
Address sizes:                   39 bits physical, 48 bits virtual
Byte Order:                      Little Endian
CPU(s):                          8
On-line CPU(s) list:             0-7
Vendor ID:                       GenuineIntel
Model name:                      11th Gen Intel(R) Core(TM) i7-1185G7 @ 3.00GHz
CPU family:                      6
Model:                           140
Thread(s) per core:              2
Core(s) per socket:              4
Socket(s):                       1
Stepping:                        1
CPU(s) scaling MHz:              81%
CPU max MHz:                     4800.0000
CPU min MHz:                     400.0000
```

1 Excercise 1

1.0.1 Linear program

We say that two cities are adjacent if they are in range D of each other.
A city is covered if at least one city with a store is adjacent to it.
The objective is to find minimal placement of stores that covers every city.

[ ]:
```python
def solve(adjacency_matrix, relaxed=False, verbose=False):
        vtype = GRB.CONTINUOUS if relaxed else GRB.BINARY

        model = gp.Model()
        cities = model.addVars(len(adjacency_matrix), vtype=vtype, lb=0.0, ub=1.
 ↪0)

        model.setObjective(cities.sum(), GRB.MINIMIZE)

        for a in adjacency_matrix:
                model.addConstr(cities.prod(gp.tuplelist(a)) >= 1)

        model.Params.OutputFlag = verbose
        model.optimize()
        return model
```

1.0.2 Solving integer formulation

[ ]:
```python
coordinates, distances, cities = utils.load_dataset(frac=0.5, seed=123).values()
len(cities)
```

[ ]: 1459

[ ]:
```python
%time model = solve(distances < 50, verbose=True)
```

```
Gurobi Optimizer version 11.0.0 build v11.0.0rc2 (linux64 - "Fedora Linux 36
(Workstation Edition)")

CPU model: 11th Gen Intel(R) Core(TM) i7-1185G7 @ 3.00GHz, instruction set
[SSE2|AVX|AVX2|AVX512]
Thread count: 4 physical cores, 8 logical processors, using up to 8 threads

Optimize a model with 1459 rows, 1459 columns and 73047 nonzeros
Model fingerprint: 0x9dc2816c
Variable types: 0 continuous, 1459 integer (1459 binary)
Coefficient statistics:
  Matrix range     [1e+00, 1e+00]
  Objective range  [1e+00, 1e+00]
  Bounds range     [1e+00, 1e+00]
  RHS range        [1e+00, 1e+00]
Found heuristic solution: objective 74.0000000

CPU model: 11th Gen Intel(R) Core(TM) i7-1185G7 @ 3.00GHz, instruction set
[SSE2|AVX|AVX2|AVX512]
Thread count: 4 physical cores, 8 logical processors, using up to 8 threads

Optimize a model with 1459 rows, 1459 columns and 73047 nonzeros
Model fingerprint: 0x9dc2816c
```

```
Variable types: 0 continuous, 1459 integer (1459 binary)
Coefficient statistics:
  Matrix range     [1e+00, 1e+00]
  Objective range  [1e+00, 1e+00]
  Bounds range     [1e+00, 1e+00]
  RHS range        [1e+00, 1e+00]
Found heuristic solution: objective 74.0000000
Presolve removed 489 rows and 398 columns
Presolve time: 0.21s
Presolved: 970 rows, 1061 columns, 37479 nonzeros
Found heuristic solution: objective 69.0000000
Variable types: 0 continuous, 1061 integer (1061 binary)

Root relaxation: objective 4.740519e+01, 2618 iterations, 0.24 seconds (0.33
work units)

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

     0     0   47.40519    0  376   69.00000   47.40519  31.3%     -    0s
H    0     0                       67.0000000   47.40519  29.2%     -    0s
H    0     0                       65.0000000   47.40519  27.1%     -    0s
H    0     0                       57.0000000   47.40519  16.8%     -    0s
H    0     0                       56.0000000   47.40519  15.3%     -    0s
H    0     0                       54.0000000   47.40519  12.2%     -    0s
     0     0   47.60304    0  380   54.00000   47.60304  11.8%     -    0s
H    0     0                       53.0000000   47.61691  10.2%     -    0s
     0     0   47.61691    0  374   53.00000   47.61691  10.2%     -    0s
H    0     0                       52.0000000   47.73772  8.20%     -    1s
     0     0   47.73772    0  379   52.00000   47.73772  8.20%     -    1s
     0     0   47.75281    0  387   52.00000   47.75281  8.17%     -    1s
     0     0   47.86800    0  383   52.00000   47.86800  7.95%     -    1s
     0     0   47.89807    0  384   52.00000   47.89807  7.89%     -    1s
     0     0   47.90371    0  391   52.00000   47.90371  7.88%     -    1s
     0     0   47.90467    0  399   52.00000   47.90467  7.88%     -    1s
     0     0   47.90470    0  397   52.00000   47.90470  7.88%     -    1s
H    0     0                       51.0000000   47.90735  6.06%     -    2s
     0     0   47.98472    0  374   51.00000   47.98472  5.91%     -    2s
     0     0   47.99264    0  382   51.00000   47.99264  5.90%     -    2s
     0     0   47.99497    0  374   51.00000   47.99497  5.89%     -    2s
     0     0   47.99569    0  384   51.00000   47.99569  5.89%     -    2s
     0     0   47.99589    0  382   51.00000   47.99589  5.89%     -    2s
     0     0   48.01476    0  407   51.00000   48.01476  5.85%     -    2s
     0     0   48.01838    0  397   51.00000   48.01838  5.85%     -    2s
     0     0   48.01882    0  404   51.00000   48.01882  5.85%     -    2s
     0     0   48.01899    0  406   51.00000   48.01899  5.85%     -    2s
     0     0   48.02812    0  412   51.00000   48.02812  5.83%     -    2s
     0     0   48.03021    0  408   51.00000   48.03021  5.82%     -    2s
```

```
     0     0   48.03092    0  406   51.00000   48.03092  5.82%     -    2s
     0     0   48.03096    0  405   51.00000   48.03096  5.82%     -    2s
     0     0   48.03452    0  407   51.00000   48.03452  5.81%     -    3s
     0     0   48.03550    0  408   51.00000   48.03550  5.81%     -    3s
     0     0   48.03578    0  411   51.00000   48.03578  5.81%     -    3s
     0     0   48.04018    0  409   51.00000   48.04018  5.80%     -    3s
     0     0   48.04113    0  411   51.00000   48.04113  5.80%     -    3s
     0     0   48.04145    0  407   51.00000   48.04145  5.80%     -    3s
     0     0   48.04372    0  403   51.00000   48.04372  5.80%     -    3s
     0     0   48.04433    0  418   51.00000   48.04433  5.80%     -    3s
     0     0   48.04448    0  415   51.00000   48.04448  5.80%     -    3s
     0     0   48.04515    0  416   51.00000   48.04515  5.79%     -    3s
     0     0   48.04575    0  416   51.00000   48.04575  5.79%     -    3s
H    0     0                      50.0000000   48.04613  3.91%     -    5s
     0     2   48.04613    0  416   50.00000   48.04613  3.91%     -    5s
   301   211   48.49195    5  345   50.00000   48.37386  3.25%   270   10s
   755   386     cutoff   12        50.00000   48.58413  2.83%   229   15s
  1236   397   48.97990   12  314   50.00000   48.68417  2.63%   240   20s
  1606   327   48.98409   11  304   50.00000   48.76623  2.47%   230   25s

Cutting planes:
  MIR: 244
  Zero half: 1
  Mod-K: 2

Explored 2195 nodes (462700 simplex iterations) in 28.28 seconds (63.08 work
units)
Thread count was 8 (of 8 available processors)

Solution count 10: 50 51 52 … 69

Optimal solution found (tolerance 1.00e-04)
Best objective 5.000000000000e+01, best bound 5.000000000000e+01, gap 0.0000%
CPU times: user 2min 26s, sys: 1.24 s, total: 2min 27s
Wall time: 1min 12s
```

[ ]:
```
stores = coordinates[utils.vars(model, dtype=bool)]
seaborn.scatterplot(x = coordinates[:,0], y = coordinates[:,1], s=20)
seaborn.scatterplot(x = stores[:,0], y = stores[:,1], marker="x", color="red",␣
 ↪s=50)
plt.title("Store locations"); plt.show()
print(f"There are {model.objVal} stores placed")
```


```
There are 50.0 stores placed
```

I was able to compute the solution for at least half the dataset in a reasonable time. About 50 stores are required to cover all of the cities.

1.0.3 Model relaxation

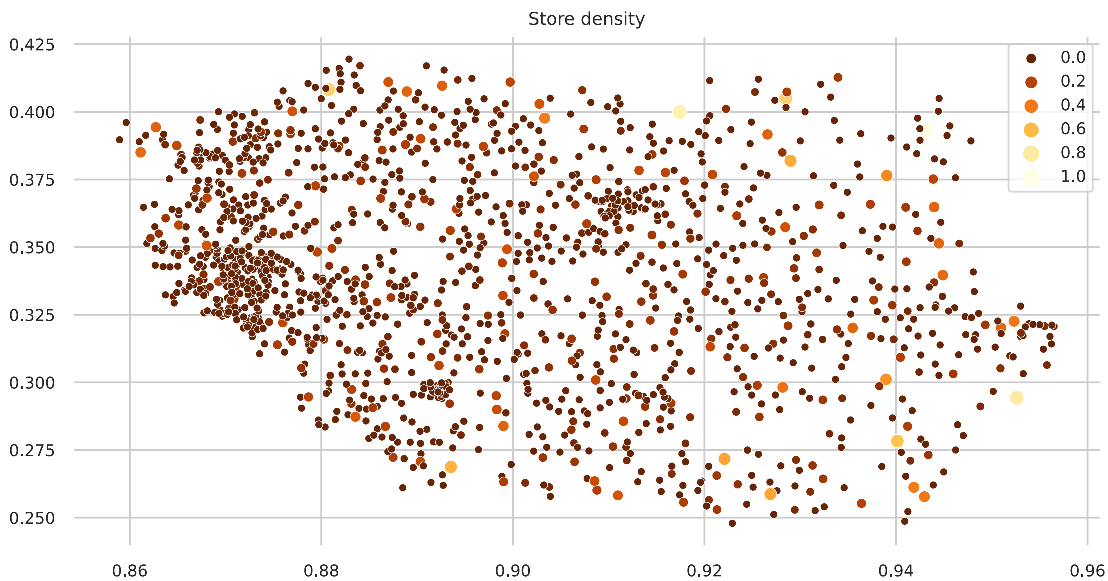
[ ]:
```
%time model = solve(distances < 50, verbose=True, relaxed=True)
```

```
Gurobi Optimizer version 11.0.0 build v11.0.0rc2 (linux64 - "Fedora Linux 36
(Workstation Edition)")

CPU model: 11th Gen Intel(R) Core(TM) i7-1185G7 @ 3.00GHz, instruction set
[SSE2|AVX|AVX2|AVX512]
Thread count: 4 physical cores, 8 logical processors, using up to 8 threads

Optimize a model with 1459 rows, 1459 columns and 73047 nonzeros
Model fingerprint: 0x3cc0f0be
Coefficient statistics:
  Matrix range     [1e+00, 1e+00]
  Objective range  [1e+00, 1e+00]
  Bounds range     [1e+00, 1e+00]
  RHS range        [1e+00, 1e+00]

CPU model: 11th Gen Intel(R) Core(TM) i7-1185G7 @ 3.00GHz, instruction set
[SSE2|AVX|AVX2|AVX512]
Thread count: 4 physical cores, 8 logical processors, using up to 8 threads

Optimize a model with 1459 rows, 1459 columns and 73047 nonzeros
```

```
Model fingerprint: 0x3cc0f0be
Coefficient statistics:
  Matrix range     [1e+00, 1e+00]
  Objective range  [1e+00, 1e+00]
  Bounds range     [1e+00, 1e+00]
  RHS range        [1e+00, 1e+00]
Presolve removed 18 rows and 18 columns
Presolve time: 0.02s
Presolved: 1441 rows, 1441 columns, 71463 nonzeros

Concurrent LP optimizer: dual simplex and barrier
Showing barrier log only…

Ordering time: 0.04s

Barrier statistics:
 AA' NZ     : 1.024e+05
 Factor NZ  : 2.611e+05 (roughly 3 MB of memory)
 Factor Ops : 5.665e+07 (less than 1 second per iteration)
 Threads    : 3

                  Objective                Residual
Iter       Primal          Dual         Primal    Dual     Compl     Time
   0   1.22398393e+03  0.00000000e+00  0.00e+00 0.00e+00  6.40e-01     0s
   1   1.34760819e+02  1.03938869e+01  0.00e+00 6.23e-02  6.69e-02     0s
   2   7.71352519e+01  3.19882854e+01  0.00e+00 8.12e-03  1.61e-02     0s
   3   5.64557841e+01  4.17874377e+01  0.00e+00 1.14e-03  4.61e-03     0s
   4   5.00547000e+01  4.56382592e+01  0.00e+00 6.56e-05  1.39e-03     0s
   5   4.80740777e+01  4.67903881e+01  0.00e+00 6.23e-06  4.17e-04     0s
   6   4.75743656e+01  4.72646457e+01  0.00e+00 7.77e-16  1.02e-04     0s
   7   4.74346072e+01  4.73842751e+01  0.00e+00 5.55e-16  1.68e-05     0s
   8   4.74124959e+01  4.73997242e+01  0.00e+00 8.88e-16  4.32e-06     0s
   9   4.74066777e+01  4.74036181e+01  0.00e+00 6.66e-16  1.04e-06     0s
  10   4.74058637e+01  4.74047201e+01  0.00e+00 4.44e-16  3.88e-07     0s
  11   4.74053489e+01  4.74050440e+01  0.00e+00 5.55e-16  1.04e-07     0s
  12   4.74052077e+01  4.74051438e+01  0.00e+00 5.55e-16  2.21e-08     0s
  13   4.74051957e+01  4.74051830e+01  0.00e+00 5.55e-16  4.40e-09     0s
  14   4.74051931e+01  4.74051927e+01  0.00e+00 8.88e-16  1.26e-10     0s

Barrier solved model in 14 iterations and 0.23 seconds (0.15 work units)
Optimal objective 4.74051931e+01

Crossover log…

     113 DPushes remaining with DInf 0.0000000e+00                 0s
       0 DPushes remaining with DInf 0.0000000e+00                 0s

     154 PPushes remaining with PInf 0.0000000e+00                 0s
```

```
       0 PPushes remaining with PInf 0.0000000e+00                 0s

  Push phase complete: Pinf 0.0000000e+00, Dinf 9.8674541e-14      0s


Solved with barrier
Iteration    Objective       Primal Inf.    Dual Inf.      Time
     238    4.7405193e+01   0.000000e+00   0.000000e+00      0s

Solved in 238 iterations and 0.30 seconds (0.17 work units)
Optimal objective  4.740519296e+01
CPU times: user 44.3 s, sys: 50.9 ms, total: 44.3 s
Wall time: 44 s
```

[ ]:
```
storiness = utils.vars(model, dtype=float)
seaborn.scatterplot(x = coordinates[:,0], y = coordinates[:,1], s=20,␣
  ↪hue=storiness, size=storiness, palette="YlOrBr_r")
plt.title("Store density"); plt.show()
print(f"There are {model.objVal} stores placed")
```




```
There are 47.40519295581441 stores placed
```

The relaxed model is computed in half the time of the original.
In the end we did get away with a little less overall storiness with our continous formulation, but we are not far off.

1.0.4 Minimal distance per placement size

Unfortunately, I was unable to come up with a much better solution than simply checking different distance ranges since distance is not a variable of my linear program.
Such experiment requires the dataset to be much smaller.

[ ]:
```
coordinates, distances, cities = utils.load_dataset(frac = 0.05, seed = 123).
  ↪values()
len(cities)
```

[ ]:
```
148
```

[ ]:
```
print(
        solve(distances < 400).ObjVal,
        solve(distances < 350).ObjVal,
        solve(distances < 50).ObjVal,
)
```

```
1.0 2.0 41.0
```

Objective for D = 50 is 41 when considering only 5% of the full dataset.

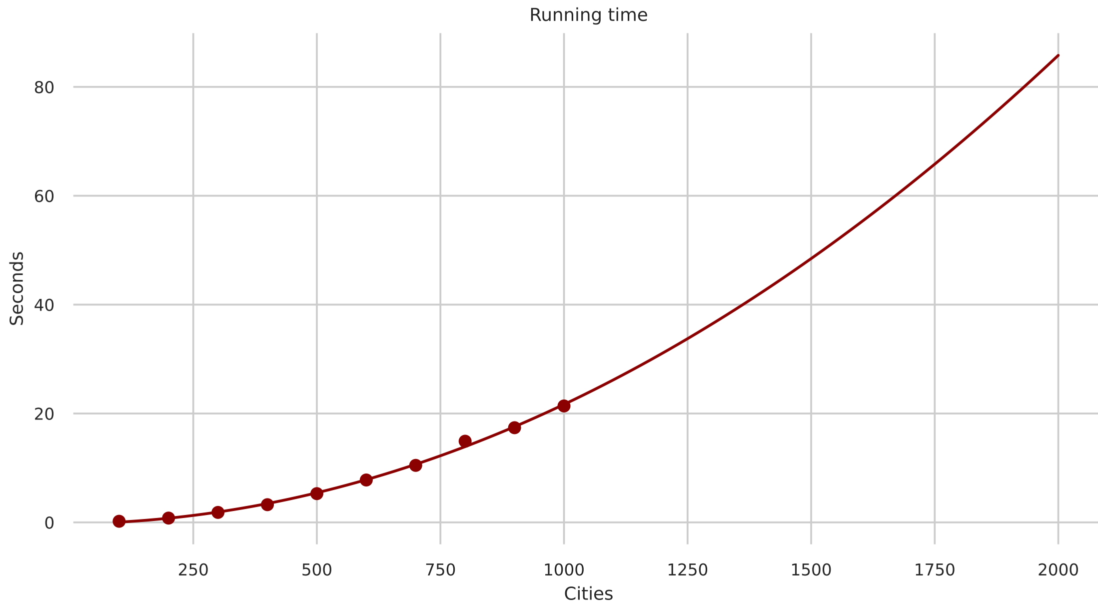
[ ]:
```
pandas.set_option('display.max_columns', None)
experiments = [(d, int(solve(distances < d).ObjVal)) for d in range(40, 400)]
pandas.DataFrame(experiments, columns=["distance", "k"]).groupby("k").min().
  ↪reset_index().transpose()
```

[ ]:
```
            0    1    2    3    4    5    6    7    8    9   10  11  12  13  \
k           1    2    3    4    5    6    7    8    9   10   11  13  14  15
distance  387  282  226  192  167  147  129  120  115  113   99  96  94  89

          14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  \
k         16  17  18  20  22  24  25  26  27  28  30  32  35  36  38  40  41
distance  87  79  77  73  72  70  67  65  62  60  57  56  55  54  53  52  50

          31  32  33  34  35  36  37  38
k         42  43  45  47  48  50  53  56
distance  49  48  46  45  44  43  42  40
```

1.0.5 Running time

[ ]:
```
adjacent = tuple(utils.load_dataset(seed=123).values())[1] < 50
x = numpy.arange(10) * 100 + 100
y = [utils.timeit(lambda: solve(adjacent[:n, :n])) for n in x]

utils.polyplot(x, y, extrapolate=2.0, degree=2, color="darkred")
plt.xlabel('Cities')
plt.ylabel('Seconds')
plt.title('Running time')
```

```python
plt.show()
```



2 Excercise 2

2.0.1 Linear program

Below is an implementation of the Dantzig-Fulkerson-Johnson linear program formulation for the metric traveling salesman problem.

[ ]:
```python
def findcycle(order, start = 0):
        cycle = [start]
        while cycle[0] != (next := order[cycle[-1]]):
                cycle.append(next)
        return cycle

def mincycle(order):
        unvisited = set(order)
        cycles = []
        while unvisited:
                start = unvisited.pop()
                cycle = findcycle(order, start)
                cycles.append(cycle)
                unvisited = unvisited.difference(cycle)
        return min(cycles, key=len)

def elimcycles(model, where):
        if where != GRB.Callback.MIPSOL:
```

```
                return

        X = model._vars
        C = utils.cycles(list(model.cbGetSolution(X).values()))

        Q = mincycle(C)
        if len(Q) < len(C):
                model.cbLazy(gp.quicksum(X[edge] for edge in product(Q, Q)) <=␣
 ↪len(Q)-1)


def solve(A: numpy.ndarray[int, int], lazy = True, verbose = False):
        model = gp.Model()
        model.Params.OutputFlag = verbose

        V = numpy.arange(n := len(A))
        X = model._vars = model.addVars(*A.shape, vtype=GRB.BINARY)

        model.setObjective( gp.quicksum(X[edge] * A[edge] for edge in␣
 ↪product(V, V)) )

        model.addConstrs( X[i, i] == 0 for i in range(n) )

        model.addConstrs( X.sum('*', j) == 1 for j in range(n) )
        model.addConstrs( X.sum(i, '*') == 1 for i in range(n) )

        if lazy:
                model.Params.LazyConstraints = 1
                model.optimize(elimcycles)
        else:
                powerset = utils.powerset(numpy.arange(n))
                model.addConstrs( gp.quicksum(X[edge] for edge in product(Q,␣
 ↪Q)) <= len(Q) - 1 for Q in powerset if n != len(Q) >= 2 )
                model.optimize()
        return model
```

2.0.2 Eager implementation

Model has all of the constraints enabled from the beginning.
With this approach I was able to solve instances of at most 16 cities in a reasonable time.

[ ]:
```
data = coordinates, distances, cities = utils.load_dataset(frac=0.007,␣
 ↪seed=123).values()
len(cities)
```

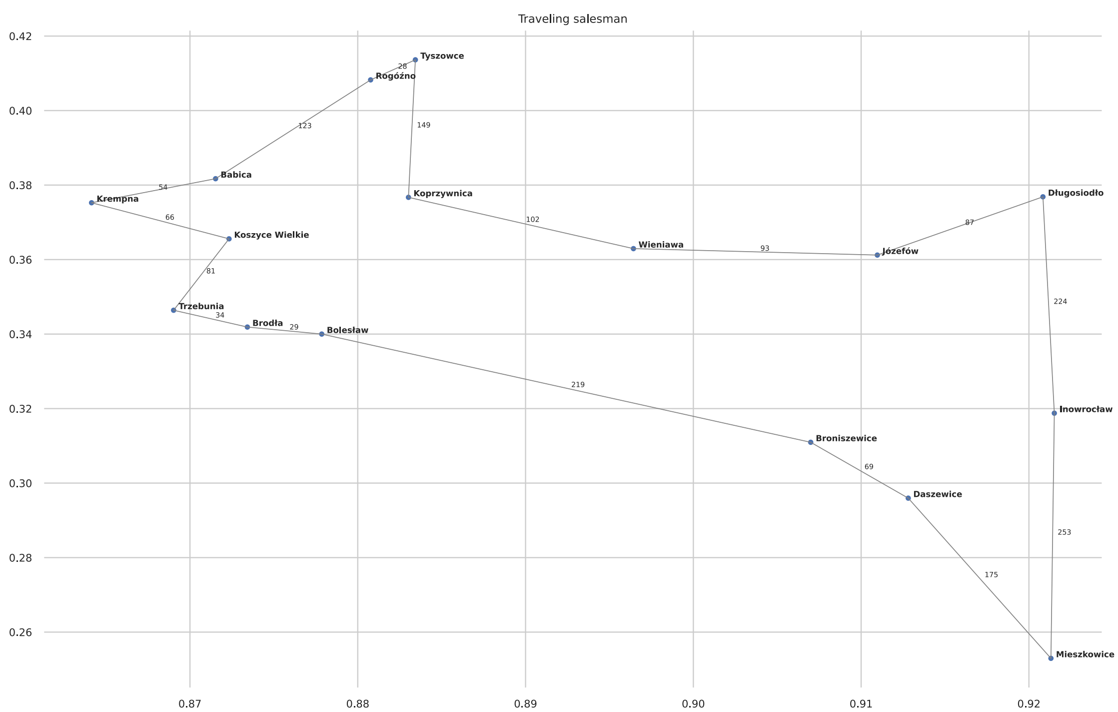
[ ]: 16

[ ]:
```
%time model = solve(distances, lazy=False, verbose=True)
```

```
Gurobi Optimizer version 11.0.0 build v11.0.0rc2 (linux64 - "Fedora Linux 36
(Workstation Edition)")

CPU model: 11th Gen Intel(R) Core(TM) i7-1185G7 @ 3.00GHz, instruction set
[SSE2|AVX|AVX2|AVX512]
Thread count: 4 physical cores, 8 logical processors, using up to 8 threads

Optimize a model with 65566 rows, 256 columns and 4456704 nonzeros

CPU model: 11th Gen Intel(R) Core(TM) i7-1185G7 @ 3.00GHz, instruction set
[SSE2|AVX|AVX2|AVX512]
Thread count: 4 physical cores, 8 logical processors, using up to 8 threads

Optimize a model with 65566 rows, 256 columns and 4456704 nonzeros
Model fingerprint: 0x2e06f7f5
Variable types: 0 continuous, 256 integer (256 binary)
Coefficient statistics:
  Matrix range     [1e+00, 1e+00]
  Objective range  [3e+01, 7e+02]
  Bounds range     [1e+00, 1e+00]
  RHS range        [1e+00, 1e+01]
Found heuristic solution: objective 4467.0000000
Presolve removed 16 rows and 16 columns (presolve time = 5s) …
Presolve removed 16 rows and 16 columns
Presolve time: 6.35s
Presolved: 65550 rows, 240 columns, 3932400 nonzeros
Variable types: 0 continuous, 240 integer (240 binary)
Root relaxation presolved: 240 rows, 65790 columns, 3932640 nonzeros


Root simplex log…

Iteration    Objective       Primal Inf.    Dual Inf.      Time
       0      handle free variables                          8s

Starting sifting (using dual simplex for sub-problems)…

    Iter     Pivots    Primal Obj      Dual Obj        Time
       0         75    -infinity      3.0010000e+03      8s

Sifting complete

     118    1.7860000e+03   0.000000e+00   0.000000e+00      9s
     118    1.7860000e+03   0.000000e+00   0.000000e+00      9s

Root relaxation: objective 1.786000e+03, 118 iterations, 2.16 seconds (2.26 work
```

```
units)

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

*    0     0               0     1786.0000000 1786.00000  0.00%     -    8s

Explored 1 nodes (118 simplex iterations) in 8.87 seconds (6.30 work units)
Thread count was 8 (of 8 available processors)

Solution count 2: 1786 4467

Optimal solution found (tolerance 1.00e-04)
Best objective 1.786000000000e+03, best bound 1.786000000000e+03, gap 0.0000%
CPU times: user 10.4 s, sys: 586 ms, total: 11 s
Wall time: 11.1 s
```

[ ]:
```
utils.graphplot(utils.edges(utils.vars(model)), *data, title='Traveling␣
↪salesman')
print(f"Value of the objective function is {int(model.objVal)}")
```




```
Value of the objective function is 1786
```

2.0.3 Lazy implementation

Each time a subcycle is detected we add an extra constraint to the model targetting that specific cycle.
Lazy computation turned out to be quite efficient and thus allowed me to solve instances of up to an order of magnitude greater than before.

```
[ ]: data = coordinates, distances, cities = utils.load_dataset(frac = 0.06, seed =
      ↪123).values()
     len(cities)
```

```
[ ]: 172
```

```
[ ]: %time model = solve(distances, lazy=True, verbose=True)
```

```
Set parameter LazyConstraints to value 1
Gurobi Optimizer version 11.0.0 build v11.0.0rc2 (linux64 - "Fedora Linux 36
(Workstation Edition)")

CPU model: 11th Gen Intel(R) Core(TM) i7-1185G7 @ 3.00GHz, instruction set
[SSE2|AVX|AVX2|AVX512]
Thread count: 4 physical cores, 8 logical processors, using up to 8 threads

Optimize a model with 516 rows, 29584 columns and 59340 nonzeros
Model fingerprint: 0x02487f94
Variable types: 0 continuous, 29584 integer (29584 binary)
Coefficient statistics:
  Matrix range     [1e+00, 1e+00]
  Objective range  [3e+00, 7e+02]
  Bounds range     [1e+00, 1e+00]
  RHS range        [1e+00, 1e+00]
Gurobi Optimizer version 11.0.0 build v11.0.0rc2 (linux64 - "Fedora Linux 36
(Workstation Edition)")

CPU model: 11th Gen Intel(R) Core(TM) i7-1185G7 @ 3.00GHz, instruction set
[SSE2|AVX|AVX2|AVX512]
Thread count: 4 physical cores, 8 logical processors, using up to 8 threads

Optimize a model with 516 rows, 29584 columns and 59340 nonzeros
Model fingerprint: 0x02487f94
Variable types: 0 continuous, 29584 integer (29584 binary)
Coefficient statistics:
  Matrix range     [1e+00, 1e+00]
  Objective range  [3e+00, 7e+02]
  Bounds range     [1e+00, 1e+00]
  RHS range        [1e+00, 1e+00]
Presolve removed 172 rows and 172 columns
Presolve time: 0.03s
Presolved: 344 rows, 29412 columns, 58824 nonzeros
```

```
Variable types: 0 continuous, 29412 integer (29412 binary)

Root relaxation: objective 4.212000e+03, 275 iterations, 0.00 seconds (0.00 work
units)

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

     0     0 4212.00000    0    -          - 4212.00000      -     -    0s
     0     0 4239.00000    0    -          - 4239.00000      -     -    0s
     0     0 4262.00000    0    -          - 4262.00000      -     -    0s
     0     0 4268.00000    0    -          - 4268.00000      -     -    0s
     0     0 4350.00000    0   38          - 4350.00000      -     -    0s
     0     0 4395.00000    0   38          - 4395.00000      -     -    0s
     0     2 4398.00000    0   38          - 4398.00000      -     -    0s
  1399  1038 5168.00000   41   38          - 4422.00000      -  11.4    5s
  1816  1282 5082.00000   23    -          - 5051.75000      -  12.4   10s
* 2551  1390              45    5260.0000000 5051.75000  3.96%  13.2   13s
  2675  1452 5195.75000   29   45 5260.00000 5059.00000  3.82%  13.4   15s
H 2697  1353                    5240.0000000 5059.00000  3.45%  13.4   15s
H 2777  1304                    5235.0000000 5060.00000  3.34%  13.5   15s
  4799  1526 5224.00000   33   83 5235.00000 5134.50000  1.92%  15.5   21s
* 4817   756              38    5207.0000000 5134.50000  1.39%  15.5   21s
* 5583   805              38    5206.0000000 5164.00000  0.81%  16.2   22s
* 5589   740              38    5204.0000000 5164.00000  0.77%  16.2   22s
* 5893   707              34    5202.0000000 5166.00000  0.69%  16.4   23s
  6922   598 5197.75000   29  226 5202.00000 5178.50000  0.45%  17.1   25s
* 7280   467              31    5199.0000000 5180.00000  0.37%  17.3   25s

Cutting planes:
  Gomory: 8
  Cover: 7
  Inf proof: 2
  Mod-K: 5
  Lazy constraints: 103

Explored 8102 nodes (142216 simplex iterations) in 26.70 seconds (28.67 work
units)
Thread count was 8 (of 8 available processors)

Solution count 8: 5199 5202 5204 … 5260

Optimal solution found (tolerance 1.00e-04)
Best objective 5.199000000000e+03, best bound 5.199000000000e+03, gap 0.0000%

User-callback calls 19510, time in user-callback 3.77 sec
CPU times: user 1min 35s, sys: 2.19 s, total: 1min 37s
Wall time: 27.1 s
```

[ ]:
```
utils.graphplot(utils.edges(utils.vars(model)), *data, title='Traveling salesman')
print(f"Value of the objective function is {int(model.objVal)}")
```


```
Value of the objective function is 5199
```

3 Excercise 3

3.0.1 Christofides algorithm

Below implementation uses the networkx library.
The minimal weight matching implementation is claimed to be in $O(|V|^3)$ time.

[ ]:
```
def christofides(distances):
        G = nx.Graph()
        G.add_nodes_from(numpy.arange(len(distances)))
        for edge in combinations(G.nodes, 2):
                G.add_edge(*edge, weight=distances[edge])

        mst = nx.minimum_spanning_tree(G)
        ods = G.subgraph([v for v, degree in mst.degree if degree % 2])

        MG = nx.MultiGraph()
        MG.add_edges_from(mst.edges)
        MG.add_edges_from(nx.min_weight_matching(ods))
```

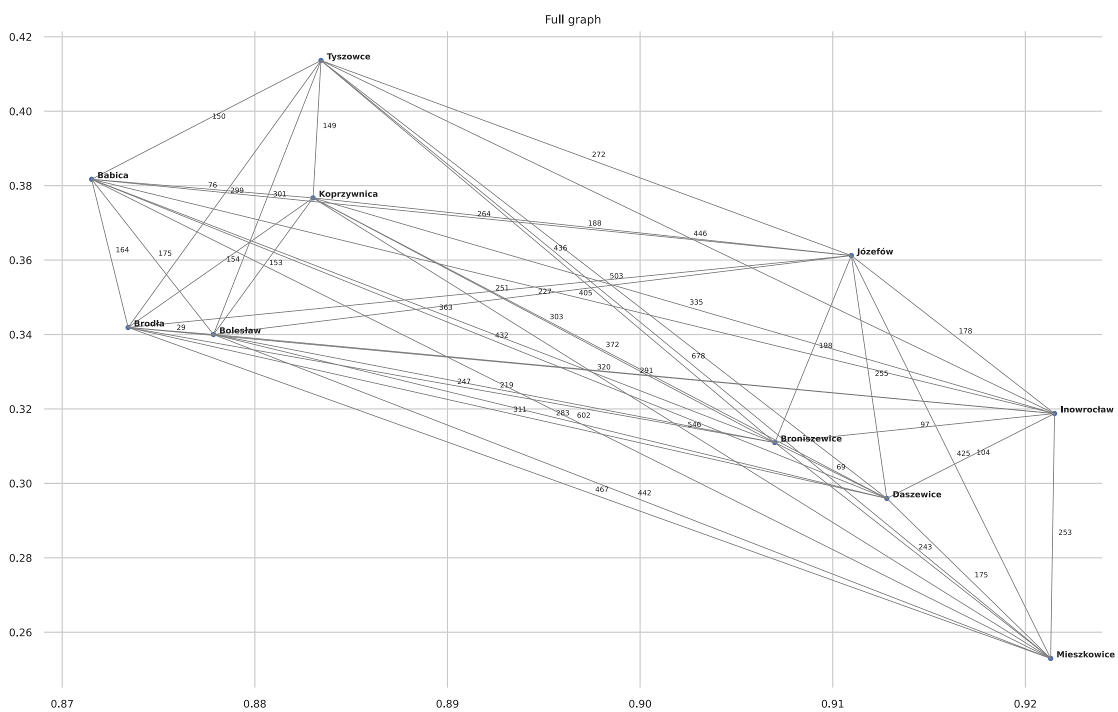
```
        return sum(distances[edge] for edge in utils.tour(MG))
```

3.0.2 Christofides step by step

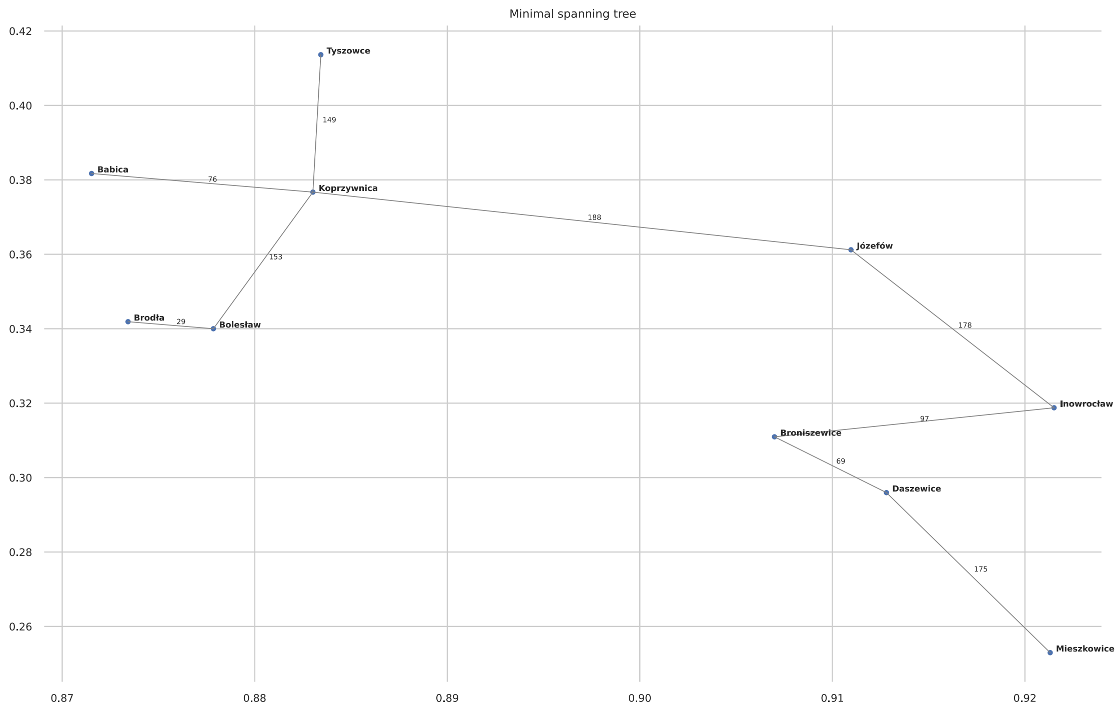
[ ]:
```
data = coordinates, distances, cities = utils.load_dataset(frac = 0.005, seed = 123).values()
len(cities)
```

[ ]: 10

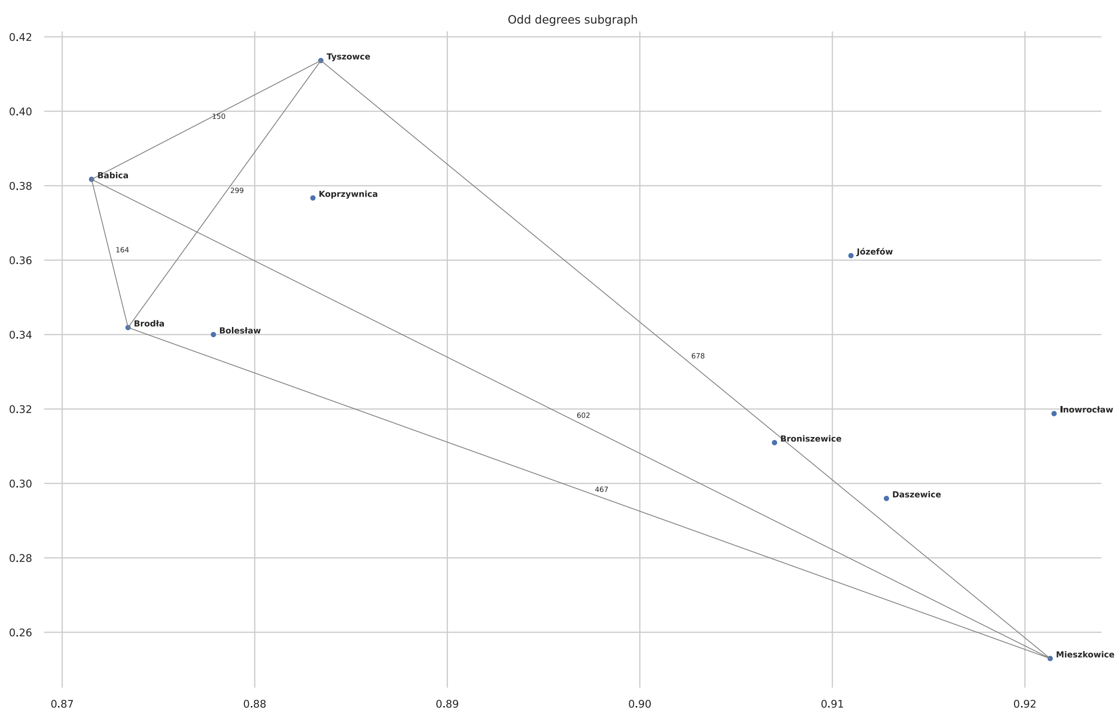
[ ]:
```
G = nx.Graph()
G.add_nodes_from(numpy.arange(n := len(cities)))
for edge in combinations(G.nodes, 2):
        G.add_edge(*edge, weight=distances[edge])
utils.graphplot(G.edges, *data, title=f"Full graph")
```



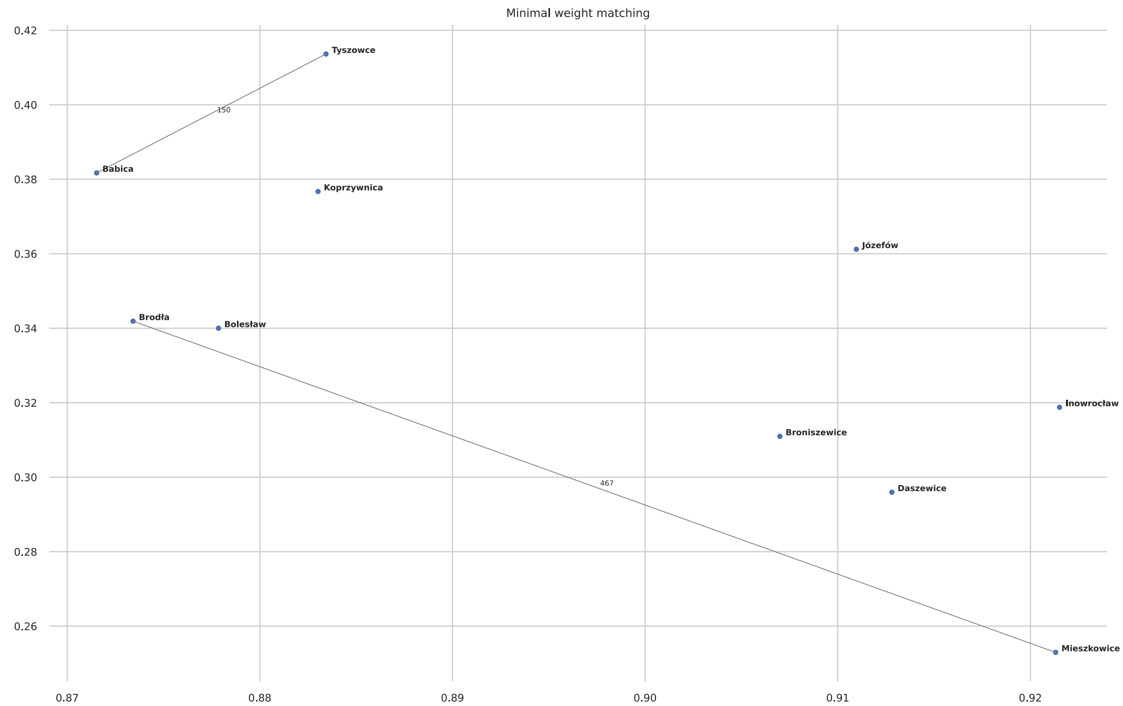
[ ]:
```
utils.graphplot((MST := nx.minimum_spanning_tree(G)).edges, *data, title="Minimal spanning tree")
```



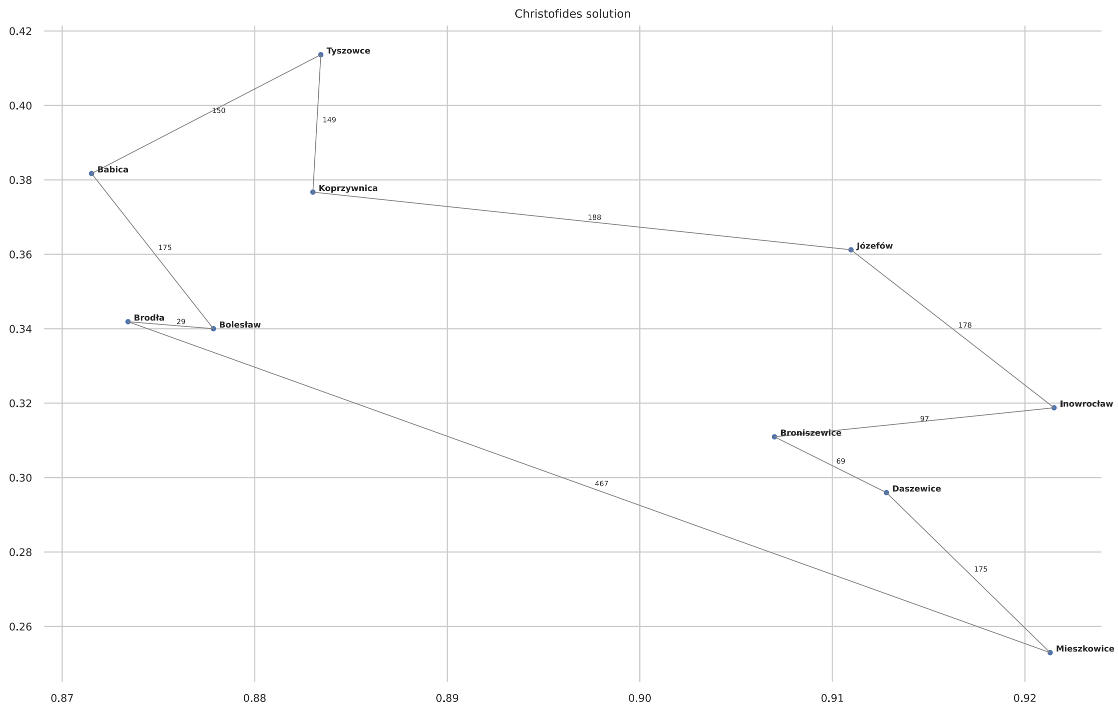
[ ]:
```python
utils.graphplot((ODS := G.subgraph([v for v, degree in MST.degree if degree %
    2])).edges, *data, title = "Odd degrees subgraph")
```



[ ]:
```
utils.graphplot((MWM := ODS.edge_subgraph(nx.min_weight_matching(ODS))).edges,␣
↪*data, title="Minimal weight matching")
```



[ ]:
```
MG = nx.MultiGraph()
MG.add_edges_from(MST.edges)
MG.add_edges_from(MWM.edges)
utils.graphplot(utils.tour(MG), *data, title="Christofides solution")
```

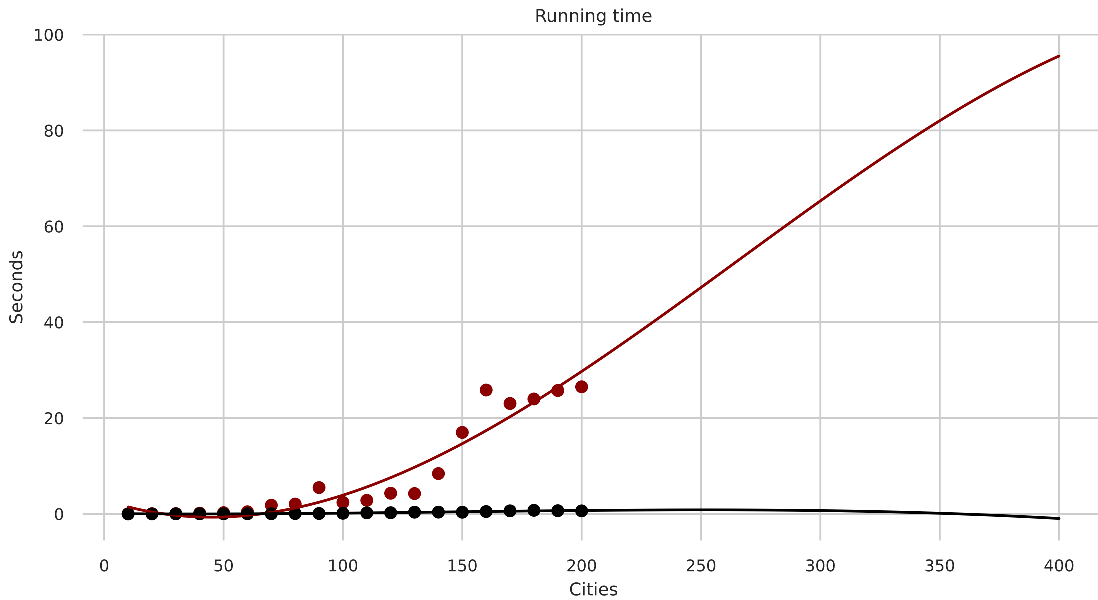


3.0.3 Efficiency and running time

```
[ ]: data = coordinates, distances, cities = utils.load_dataset(frac = 0.06, seed =␣
     ↪123).values()
     len(cities)
```

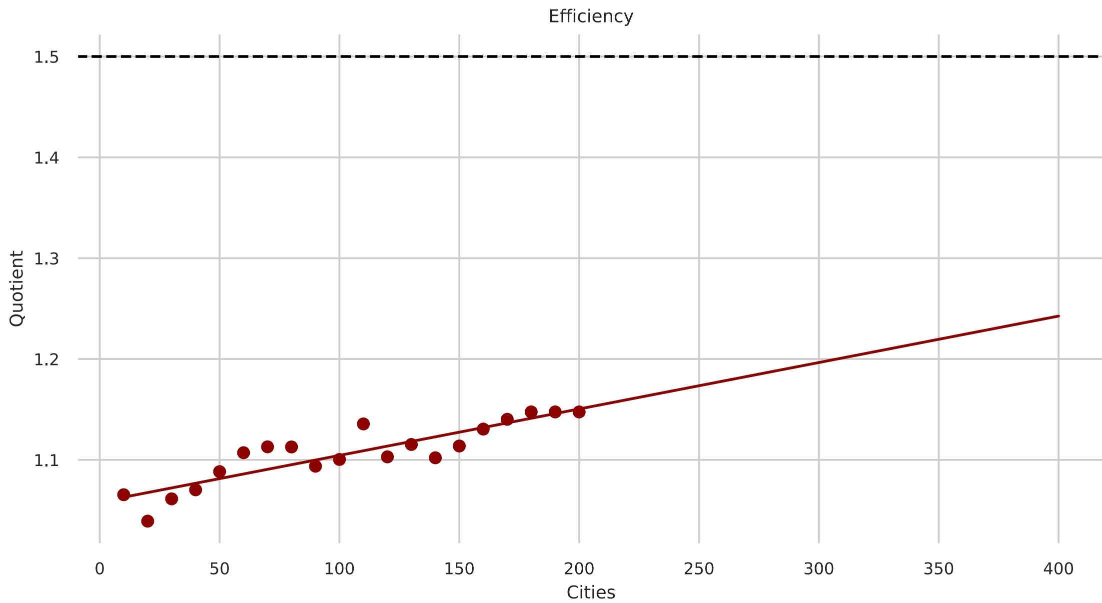
```
[ ]: 172
```

```
[ ]: x = numpy.arange(20) * 10 + 10
     y = [utils.timeit(lambda: solve(distances[:n, :n])) for n in x]
     z = [utils.timeit(lambda: christofides(distances[:n, :n])) for n in x]

     utils.polyplot(x, y, extrapolate=2.0, degree=3, color="darkred")
     utils.polyplot(x, z, extrapolate=2.0, degree=3, color="black")
     plt.xlabel('Cities')
     plt.ylabel('Seconds')
     plt.title('Running time')
     plt.show()
```



Since our Christofides implementation works in polynomial time while the linear program approach is ultimately exponential this is pretty much the result we would expect.

[ ]:
```
x = numpy.arange(20) * 10 + 10
y = [christofides(distances[:n, :n]) / solve(distances[:n, :n]).objVal for n in␣
↪x]

utils.polyplot(x, y, extrapolate=2.0, degree=1, color="darkred")
plt.axhline(y=3/2, color='black', linestyle='--')
plt.xlabel('Cities')
plt.ylabel('Quotient')
plt.title('Efficiency')
plt.show()
```



We see that the approximation error grows larger with the sample size but by the guarantees of the christofides approximation ratio we expect it to never cross the 1.5 boundary.