merten-integration

August 25, 2021

1 M7 - Heuristics

1.1 *Jason Merten*

[1]:
```
import mysql.connector
import gurobipy as gp
from gurobipy import GRB
import pandas as pd
import numpy as np
```

[2]:
```
db = mysql.connector.
 ↪connect(user='root',password='root',host='localhost',database='final_integration')
cur = db.cursor()
```

[3]:
```
def populateMilage():
    file = pd.read_excel('final-integration-US-mileage.
 ↪xlsx',header=0,index_col=0)
    final = file.where(np.triu(np.ones(file.shape)).astype('bool')).stack().
 ↪reset_index().rename(columns={' ':'Source','level_1':'Destination',0:
 ↪'Distance'})
    # final = file.stack().reset_index().rename(columns={' ':'Source','level_1':
 ↪'Destination',0:'Distance'})
    print(len(final))
    for i in final.index:
        cur.execute('insert into distance (source,destination,distance) values␣
 ↪(%s,%s,%s)',(final['Source'][i],final['Destination'][i],final['Distance'][i]))
        if i%500 == 0:
            db.commit()
        if i%50000 == 0:
            print('--- Still Running --- Row #: {}'.format(i))
    db.commit()
```

[4]:
```
populateMilage()
cur.close()
db.close()
```

```
496509
--- Still Running --- Row #: 0
```

```
--- Still Running --- Row #: 50000
--- Still Running --- Row #: 100000
--- Still Running --- Row #: 150000
--- Still Running --- Row #: 200000
--- Still Running --- Row #: 250000
--- Still Running --- Row #: 300000
--- Still Running --- Row #: 350000
--- Still Running --- Row #: 400000
--- Still Running --- Row #: 450000
```

[4]:
```
# Gurobi code
m = gp.Model('SharkTank')

# Set decision variables
demand = {'Boston, MA': 1051,
          'Chicago, IL': 940,
          'Dallas, TX': 1131,
          'Denver, CO': 466,
          'Los Angeles, CA': 1301,
          'Richmond, VA': 1171,
          'Miami, FL': 1463,
          'New York City, NY': 1120,
          'Phoenix, AZ': 665,
          'Pittsburgh, PA': 1280,
          'San Francisco, CA': 615,
          'Seattle, WA': 528}
select = {}
service = {}
for source in demand.keys():
    m.addVar(vtype=GRB.BINARY,name=source)
    m.update()
    select[source] = m.getVarByName(source)
    service[source] = {}
    for destination in demand.keys():
        m.addVar(vtype=GRB.BINARY,name=source+'_'+destination)
        m.update()
        service[source][destination] = m.getVarByName(source+'_'+destination)
    m.update()

# Convert to data frame for ease of access
service = pd.DataFrame(service)

# Add constraints
m.addConstr(gp.quicksum(select.values()),GRB.EQUAL,3)
for source in demand.keys():
    m.addConstr(gp.quicksum(service[source].values.tolist()),GRB.EQUAL,1)
for site in demand.keys():
```

```
    m.addConstr(gp.quicksum(service.loc(axis=0)[site].values.tolist()),GRB.
 ↪LESS_EQUAL,select[site]*len(demand.keys()))
m.update()
```

```
------------------------------------------
Warning: your license will expire in 7 days
------------------------------------------

Academic license - for non-commercial use only - expires 2021-08-30
Using license file C:\Users\jmert\gurobi.lic
```

[7]:
```
# Populate distance matrix
db = mysql.connector.
 ↪connect(user='root',password='root',host='localhost',database='final_integration')
cur = db.cursor()
dis_matrix = {x:{} for x in demand.keys()}
i = 0
for source in demand.keys():
    for destination in demand.keys():
        if i == 6:
            print('--- Still Running ---')
        if source == destination:
            dis_matrix[source][destination] = 0
            continue
        cur.execute('select distance from distance where (source = %s or␣
 ↪destination = %s) and (source = %s or destination =␣
 ↪%s)',(source,source,destination,destination))
        dis_matrix[source][destination] = cur.fetchone()[0]
        i += 1
    i = 0
dis_matrix = pd.DataFrame(dis_matrix)
cur.close()
db.close()
```

```
--- Still Running ---
--- Still Running ---
--- Still Running ---
--- Still Running ---
--- Still Running ---
--- Still Running ---
--- Still Running ---
--- Still Running ---
--- Still Running ---
--- Still Running ---
--- Still Running ---
--- Still Running ---
--- Still Running ---
--- Still Running ---
--- Still Running ---
```

```
--- Still Running ---
```

[13]:
```
# Define objective value

m.setObjective(gp.quicksum(demand[city] * (dis_matrix[city].values @␣
 ↪service[city].values) / 1000 for city in demand.keys()),GRB.MINIMIZE)
m.optimize()
```

```
Gurobi Optimizer version 9.1.2 build v9.1.2rc0 (win64)
Thread count: 12 physical cores, 24 logical processors, using up to 24 threads
Optimize a model with 25 rows, 156 columns and 312 nonzeros
Model fingerprint: 0x5a060cfe
Variable types: 0 continuous, 156 integer (156 binary)
Coefficient statistics:
  Matrix range     [1e+00, 1e+01]
  Objective range  [2e+02, 5e+03]
  Bounds range     [1e+00, 1e+00]
  RHS range        [1e+00, 3e+00]

Loaded MIP start from previous solve with objective 5836.64

Presolve time: 0.00s
Presolved: 25 rows, 156 columns, 312 nonzeros
Variable types: 0 continuous, 156 integer (156 binary)

Root relaxation: objective 0.000000e+00, 24 iterations, 0.00 seconds

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

     0     0    0.00000    0   10 5836.64185    0.00000   100%     -    0s
     0     0 2719.37045    0   17 5836.64185 2719.37045  53.4%     -    0s
H    0     0                     5437.4714683 2719.37045  50.0%     -    0s
     0     0 4111.23112    0   25 5437.47147 4111.23112  24.4%     -    0s
H    0     0                     5378.7324575 4111.23112  23.6%     -    0s
     0     0 4614.13390    0   33 5378.73246 4614.13390  14.2%     -    0s
*    0     0               0    5043.0859993 5043.08600  0.00%     -    0s

Explored 1 nodes (130 simplex iterations) in 0.03 seconds
Thread count was 24 (of 24 available processors)

Solution count 4: 5043.09 5378.73 5437.47 5836.64

Optimal solution found (tolerance 1.00e-04)
Best objective 5.043085999341e+03, best bound 5.043085999341e+03, gap 0.0000%
```

1.2 Model

To import the Solver model into Gurobi I needed to create 12 source binary variables and 144 source_destination binary variables. To help with later in the process, I stored each set of these variables into dictionaries. The source_destination dictionary was converted to a DataFrame for ease of access when setting the objective. From there, creating the constraints was relatively straight forward using the gp.quicksum() function. There were a few problem areas when using the quicksum function with pandas, which is why you'll see the .values.tolist() calls to help get them into lists for the function to work properly. From this point, I read in the distances from SQL and stored them into a DataFrame and set the objective function by performing matrix multiplication. The results are nearly identical (~1.73 difference in total distance, probably due to rounding error in Excel) with the same cities being chosen for the manufacturing locations.

2 Question 1:

2.1 Where should you locate your manufacturing centers to minimize the total distance traveled to meet shipping demand?

[32]:
```
for x in select:
    if select[x].X > 0:
        print('Manufacturing location: {}'.format(x))
```

```
Manufacturing location: Los Angeles, CA
Manufacturing location: Miami, FL
Manufacturing location: Pittsburgh, PA
```

3 Question 2:

3.1 What manufacturing sites will service each city?

[40]:
```
for x in select:
    if select[x].X > 0:
        print('{} services: '.format(x))
        for j in service.loc(axis=0)[x]:
            if j.X > 0:
                print('\t'+j.VarName.split('_')[0])
```

```
Los Angeles, CA services:
        Denver, CO
        Los Angeles, CA
        Phoenix, AZ
        San Francisco, CA
        Seattle, WA
Miami, FL services:
        Miami, FL
Pittsburgh, PA services:
        Boston, MA
        Chicago, IL
```

```
Dallas, TX
Richmond, VA
New York City, NY
Pittsburgh, PA
```

4 Question 3:

4.1 What is the total number of miles traveled to satisfy the demand?

[56]:
```
print('Total miles traveled: %.3f miles' % m.getAttr('objVal'))
```

```
Total miles traveled: 5043.086 miles
```

5 Question 4:

5.1 Can you devise a greedy heuristic algorithm that produces the same recommendation?

To create a greedy algorithm to solve the same problem, you would need to start by looking at the coverage of each city with relation to the others using a maximum radius value (below I used 1200 miles). From there, you would select the city that has the highest number of covered cities as the first manufacturing site. You'll need to store all cities covered by the first manufacturing site into a list or dictionary to track what other cities are left and use it to find the next city that will provide the most additional coverage as the second manufacturing site. You'll repeat the process for the third manufacturing site that completes the total coverage of all cities. This should give you an approximate optimal solution.

[110]:
```
coverage = dis_matrix.where(dis_matrix.values <= 1200)
sizes = {x:0 for x in demand.keys()}
for x in demand.keys():
    sizes[x] = (demand[x] * coverage[x].notnull().sum()) / coverage[x].
 ↪notnull().sum()
# cities = [x for x in demand.keys()]
# coverage['Boston, MA'].notnull()
```

[115]:
```
covered = []
covered.append(coverage[coverage['Miami, FL'].notnull()].index.tolist())
```

[115]:
```
['Richmond, VA', 'Miami, FL']
```