Programmer: Yu-Ling Hsiao
Team Members: Cainan Parrish, Vijaya Mummidi, Keval Mehta

Final Project Technical Walkthrough

Using Google API and Gurobi in Python

Part I - Obtain distance data using Google Distance Matrix API

Introduction to API

Before getting started, make sure to sign up for Google Maps Platform, enable Google Distance Matrix API[1], and get an API key (This will be used later). To see the set up these things, check out this website: Get Started with Google Maps Platform

Google Distance Matrix API is a service that provides travel distance and time for given origins and destinations based on the recommended route. To use it, simply request a URL that contains the origins, destinations, and API key. The request takes the following form:

```
https://maps.googleapis.com/maps/api/distancematrix/json?units=imperial&origins=ORIGIN&destinations=DESTINATION&key=API_KEY
```

The red-colored words are subject to change. ORIGIN and DESTINATION are whatever the start point and end point you wish to put in, API_KEY is your unique credential.

The response will be either in JSON or XML format based on the parameter you pass, which includes route related data like duration, distance, and fare. These results need to be parsed if you want to extract desired values.

For more information about other optional parameters you can pass or how to parse the results, visit this website: Developer Guide

Usage in our project

Here are the example datasets after wiping out sensitive personal information.
(Click here for the complete client data file)

Client Number	Age	City	Sex	Zip	Cross Streets	Demand
1	69	Chandler	F	85286	Germann & McQueen	1
2	62	Chandler	F	85225	Cooper & Chandler	4
3	86	Chandler	F	85249	McQueen & Riggs	1
4	80	Gilbert	F	85297	Germann & Power	2
5	81	Chandler	M	85226	Chandler Blvd & Rural	1

(Click here for the complete volunteer data file)

Volunteer ID	City	Cross Streets	Zip
1	Chandler	Pecos & McQueen	85225
2	Chandler	Pecos & McQueen	85225
3	Tempe	202 & Rural	85281
4	Tempe	Warner & Priest	85284
5	Chandler	Gilbert-Riggs	85249

[1] Fees may apply. To understand the price or get free credit, go to the following website: https://developers.google.com/maps/documentation/distance-matrix/usage-and-billing

Since the distances between 180 volunteers and 335 clients were not provided, I chose to use Google API to gather them or it would take too much time searching manually.

In consideration of the privacy, though the company gave us complete address information of the personals, our team decided to use only the zip code to locate them. As a result, there would be a lot of duplicates because many people live in the same zip code area. In order to save time, avoid unnecessary efforts, and reduce the number of requests, we remained only unique zip code pairs of volunteers and clients for querying distance data.
(Click here for the unique zip code file)

Firstly, I constructed an empty table which in the headers of row and column represents volunteers' and clients' zip code. Then I filled in the distance (in miles) by requesting API with the row headers as origin and column header as the destination. Here is the example of the distance table:
(Click here for the distance code file) (Click here for the complete distance matrix file)

	85286	**85225**	**85249**	**85297**	**85226**	**85224**	**85233**	**85298**	**85234**
85225	3.9	1	6.2	9.2	9.8	4.5	3.6	12.2	7.8
85281	19.1	16.2	21.4	24.4	14.4	12.3	13.7	27.5	16.3
85284	11.6	8.6	13.9	16.9	4.3	4.7	8.3	20	16.2
85249	4.2	6.2	1	10.1	11.8	10.4	9.5	7	16.6
85224	8.2	4.5	10.5	13.5	7.3	1	5.2	16.5	10
85142	15.6	18	12.3	10.2	24	22.6	21	7.5	18.5
85248	5.8	7.9	3.4	11.7	9.8	8	11.1	8.9	18.7
85234	13.3	8.4	16.7	8.7	19.6	10.3	5.4	11	1
85226	9.7	9.8	12	15	1	7.4	12.2	18	20.2

Then, I populated the distance data into the 355×180 table which the row headers represent volunteers' number and column headers represent clients' ID. Here is how it looks like:
(Click here for the volunteer and client distance code file)
(Click here for the complete volunteer and client distance matrix file)
(*Equation 2*)[2]

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
1	3.9	1	6.2	9.2	9.8	1	4.5	3.6	3.6
2	3.9	1	6.2	9.2	9.8	1	4.5	3.6	3.6
3	19.1	16.2	21.4	24.4	14.4	16.2	12.3	13.7	13.7
4	11.6	8.6	13.9	16.9	4.3	8.6	4.7	8.3	8.3
5	4.2	6.2	1	10.1	11.8	6.2	10.4	9.5	9.5
6	8.2	4.5	10.5	13.5	7.3	4.5	1	5.2	5.2
7	15.6	18	12.3	10.2	24	18	22.6	21	21
8	11.6	8.6	13.9	16.9	4.3	8.6	4.7	8.3	8.3
9	5.8	7.9	3.4	11.7	9.8	7.9	8	11.1	11.1

This complete distance table (defined as "matrix" in the code in part II) between each volunteer and client will be used in our Gurobi model.

[2] The Equations in this doc indicate the parts that link to the corresponding equations in the mathematical model.

Part II – Use Gurobi to build model and solve problem

Introduction to Gurobi

Gurobi is a powerful optimization solver and compatible with many platforms and programming languages. In this tutorial, I will be using Anaconda (An analytics platform powered by Python) with the Gurobi library. Users need to have both Anaconda and Gurobi installed, and get a Gurobi license as well. To see how to prepare for the environment, follow this link: Gurobi and Anaconda

The model set up in Gurobi is similar to the mathematical model. Below are the 5 main components:

1) **Initialize an empty model object**

```
model = gurobipy.Model()
```

2) **Add decision variable to model**

```
decision = model.addVar()
```

3) **Set the optimization objective**

```
model.setObjective()
```

4) **Add constraints to model**

```
model.addConstr()
```

5) **Optimize the model**

```
model.optimize()
```

For the actual parameter settings and mathematical model implementations, refer to this link: Gurobi Python Documentation

Usage in our project

Besides the distance data I obtained in part I, we also need the demand of every client in the model. This is a portion of demand data (Defined as "demand_list" in the code) (*Equation 1*):

Client Number	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**
Demand	1	4	1	2	1	5	5	2	1	1	5	1	1	2	1	2	1	1	1	2

Next part, I will explain the actual codes step by step and show the corresponding mathematical equation.
(Click here for the Gurobi code file)

1) **Initialize an empty model object**

```
model = gp.Model()
```

2) **Add decision variable to model** (*Equation 3*, *Equation 8*)
Just like the distance data, the decision variables are defined as a 355×180 table data in which the rows represent volunteers and columns represent clients. Here, the value type is set to integer.

```
decision = model.addVars(matrix.shape[0], matrix.shape[1],
name='decision', vtype='I')
```

3) **Create distance dictionary for every volunteer and client pair**

The meaning of this step is to create a searchable collection (distance) recording every volunteer and client pair and its associated distance.

```
distance = {}
for i in matrix.index:
    for j in matrix.columns:
       distance[(i - 1, int(j) - 1)] = matrix.iloc[i - 1, int(j) - 1]
```

4) **Set the optimization objective** (*Equation 4*)

The first line is to sum the decision variables multiply distance for each volunteer and client pair, and set this as the objective and aim to find the minimum optimal value.

```
objective = decision.prod(distance)
model.setObjective(objective, GRB.MINIMIZE)
```

5) **Add constraints to model** (*Equation 5*, *Equation 6*, *Equation 7*, *Equation 9*)

The first one makes sure the client demands are met. The following two limit the number of visits a volunteer performs should between 1 and 4. The last one indicates the decision variables must not be negative.

```
for c in matrix.columns:
    model.addConstr(decision.sum('*', int(c) - 1) ==
demand_list['Demand'][int(c)])
for r in matrix.index.values:
    model.addConstr(decision.sum(r - 1, '*') <= 4)
    model.addConstr(decision.sum(r - 1, '*') >= 1)
for i in decision.values():
    model.addConstr(i >= 0)
```

6) **Optimize the model**

```
model.optimize()
```

If you want to access the values of objective and decision variables, apply the first and second line of code respectively below.

```
model.objVal
model.getVars()
```

The response list of getting variables function (getVars) may need to be further processed to extract the values to the desired format. The first below can get the variable's name and the second can get the value (i is position in the list).

```
model.getVars()[i].varName
model.getVars()[i].x
```

After successfully processing the python code, we got 1685.3 miles as the optimal objective and arrived with the result of decision variables.

```
Gurobi Optimizer version 9.0.0 build v9.0.0rc2 (win64)
Optimize a model with 60995 rows, 60300 columns and 241200 nonzeros
Model fingerprint: 0xa7f9b49a
Variable types: 0 continuous, 60300 integer (0 binary)
Coefficient statistics:
  Matrix range     [1e+00, 1e+00]
  Objective range  [1e+00, 1e+02]
  Bounds range     [0e+00, 0e+00]
  RHS range        [1e+00, 5e+00]
Found heuristic solution: objective 7776.4000000
Presolve added 0 rows and 360 columns
Presolve removed 59940 rows and 0 columns
Presolve time: 0.20s
Presolved: 1055 rows, 60660 columns, 121680 nonzeros
Variable types: 0 continuous, 60660 integer (28080 binary)

Root relaxation: objective 1.685300e+03, 2239 iterations, 0.11 seconds

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

*    0     0               0    1685.3000000 1685.30000  0.00%     -    0s

Explored 0 nodes (2239 simplex iterations) in 0.44 seconds
Thread count was 8 (of 8 available processors)

Solution count 2: 1685.3 7776.4

Optimal solution found (tolerance 1.00e-04)
Best objective 1.685300000000e+03, best bound 1.685300000000e+03, gap 0.0000%
```

Below is a small snippet of the result (This is modified to show both the scenarios):
([Click here for the complete result file])

	Client Number						
Volunteer ID		**1**	**2**	**3**	**4**	**5**	**6**
	1	0	0	3	0	0	0
	2	2	0	0	0	0	0
	3	0	1	0	0	2	0
	4	0	0	0	0	0	4
	5	0	0	0	1	0	0
	6	0	0	0	1	0	0

In the above matrix, 0 represents that the volunteer is not assigned to the corresponding client, whereas the cells in green show the number of visits between each volunteer and client.

Appendix

Mathematical Model

Parameters:

i= Client

j= Volunteer

V_i: Visits needed by client i in a month *Equation 1*

D_{ij}: Distance between client i and volunteer j *Equation 2*

Decisions:

X_{ij}: Number of visits made by volunteer j to client i *Equation 3*

Objective:

Minimum Total Distance: $\sum_i \sum_j X_{ij} \times D_{ij}$ *Equation 4*

Constraints:

$\sum_j X_{ij} = V_i, \forall i$ (Visits needed by clients should be meet) *Equation 5*

$\sum_i X_{ij} \leq 4$ (Volunteers should have at most 4 visits) *Equation 6*

$\sum_i X_{ij} \geq 1$ (Volunteers should have at least 1 visit) *Equation 7*

$X_{ij} \in Integer$ (Number of visits should be integer) *Equation 8*

$X_{ij} \geq 0$ (Number of visits should be positive) *Equation 9*