Math 381: Assignment 4: LPs and Graphs

Ella Kim

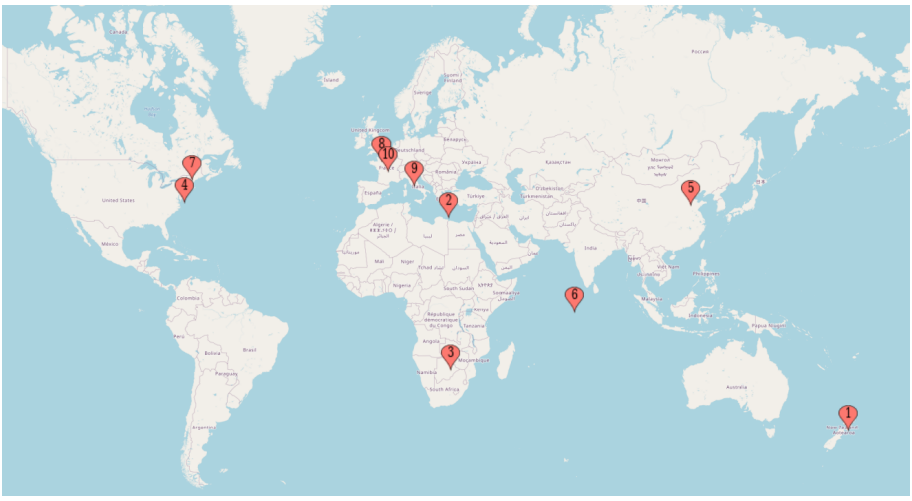
October 29 2021

1 Introduction

Whenever I am asked the question: "If you win the lottery, what would you want to do with the money," I have always included "traveling with my family" in my list. While it is nice to travel, I also would like to travel to all those places in the most efficient way possible, as there are many other things I would like to do with the rest of the lottery money (i.e, helping my parents to retire early, etc.). I will start it off with the goal of visiting 5 of the 10 cities/landmarks and see what the most efficient way to travel. Due to the direct relationship between distance and costs for transportation, I define the most efficient path of traveling as the shortest path/distance total to visit each city/location (later I will consider how much I want to visit each city as well as the total distance in miles). Below is a map of the 10 cities/landmarks I would like to visit. Labeled from 1 to 10 are:

1. Hobbiton, New Zealand
2. Santorini, Greece
3. Victoria Falls, Zimbabwe
4. New York, NY, USA
5. Forbidden City, Beijing, China
6. Sun Island, Maldives
7. Quebec City, Quebec, Canada
8. London, United Kingdom
9. Venice, Italy
10. Paris, France

Map of 10 Cities/Landmarks:



*www.mapcustomizer.com/; map data @ OpenStreetMap

Below is the Matrix/table of the 10 cities/landmarks I would like to visit, and the distance (in miles) between each of the cities. Here, the distance is the same regardless of which pair of cities is the destination or origin. Each one of the column and row names are labeled with the name of each location as well as the corresponding number ping in the Geographical Map above.

Matrix of distances between each City/Landmark:

(miles)	Hobbiton 1	Santorini 2	Victoria Falls 3	New York 4	Forbidden City 5	Sun Island 6	Quebec City 7	London 8	Venice 9	Paris 10
Hobbiton 1		10800.71	8112.62	8806.22	6554.33	7067.92	9070.07	11472.89	11433.68	11608.49
Santorini 2	10800.71		3753.93	5066.7	4714.64	3767.14	4723	1630.67	924.41	1447.54
Victoria Falls 3	8112.62	3753.93		7525.3	7033.19	3522.42	7442.82	5047.34	4459.22	4835.82
New York 4	8806.22	5066.7	7525.3		6827.54	8733.86	443.97	3461.19	4151.65	3625.68
Forbidden City 5	6554.33	4714.64	7033.19	6827.54		3703.12	6425.99	5057.37	4901.82	5105.5
Sun Island 6	7067.92	3767.14	3522.42	8733.86	3703.12		8329.79	5299.96	4638.72	5155.97
Quebec City 7	9070.07	4723	7442.82	443.97	6425.99	8329.79		3099.73	3800.51	3275.59
London 8	11472.89	1630.67	5047.34	3461.19	5057.37	5299.96	3099.73		706.82	212.6
Venice 9	11433.68	924.41	4459.22	4151.65	4901.82	4638.72	3800.51	706.82		526.07
Paris 10	11608.49	1447.54	4835.82	3625.68	5105.5	5155.97	3275.59	212.6	526.07	

Let integer n be the total number of cities/landmarks (10) and integer m be the number of cities I would like to visit (5, for now). In others words, m could be considered as the 5 steps/cities in the total path/distance to travel m number of cities.

Let there be x_{ij}, where integer i is the i-th stop/step I am at (1-5th step), integer j is the j-th city of the 10 cities on the Geographical Map (1-10th city). Each x_{ij} is 1 if and only if my current location is at step i and city j and 0 otherwise (binary).

Let there be e_{ab}, where integer a is the a-th city (1-10th city) the origin is and integer b is the b-th city (1-10th city) the destination is. Each e_{ab} is 1 if and only if I travel from origin city a to destination city b in one step and is 0 otherwise (also binary).

Let there also be d_{ab}, where integer a and b follow the definitions above in e_{ab}, where each d_{ab} is the double/float distance between origin city a and destination city b in miles.

My objective function then is defined as:

$$\sum_{1 \leq a,b \leq n} d_{ab} e_{ab}$$

Where the function is constrained:

There can only be one city visited at each individual step (i.e, I cannot be in two places at once):

$$\sum_{j=1}^{n} x_{ij} = 1, i = 1, ..., m$$

I only want to visit each city once (or never):

$$\sum_{i=1}^{m} x_{ij} \leq 1, j = 1, ..., n$$

I also want to ensure that if I visited origin city a on step i and destination city b on the next step (i.e, I went from a to b in one step), then e_{ab} of those two cities should be 1 (where I also assume that there is no negative distance, or in other words, direction of travel from cities does not matter):

$$e_{ab} \geq x_{ia} + x_{i+1b} - 1, i = 1, ..., m - 1$$

Therefore, the LP is:

Minimize $\sum_{1 \leq a,b \leq n} d_{ab} e_{ab}$ subject to:

$$\sum_{j=1}^{n} x_{ij} = 1 \qquad i = 1, ..., m \tag{1}$$

$$\sum_{i=1}^{m} x_{ij} \leq 1 \qquad j = 1, ..., n \tag{2}$$

$$e_{ab} \geq x_{ia} + x_{i+1b} - 1 \qquad i = 1, ..., m-1 \tag{3}$$

$$x_{ij}, e_{ab} \in \{0, 1\} \qquad 1 \leq i \leq m, 1 \leq a, b, j \leq n \tag{4}$$

2 Experiment 1: Distance

I use the LP above to solve my issue of wanting to travel 5 of the 10 cities/landmarks I want to visit in the shortest total distance possible. Below is the Python code for the input file:

Python code to solve LP in lpsolve:

```
import pandas as pd
# open pre-made txt file to write in
city_input = open("cities5.txt", 'w')
# start LP line for maximizing value
city_input.write("min: ")

# number of cities want to visit
m = 5

# total number of cities (does not change)
n = 10

# read in matrix of edge weight/distance
df_dist = pd.read_csv('/home/jovyan/lost+found/MATH_381/locations.csv',
                      header = None)

# function to get obective function
def getObj(n):
    # go though each city path
    for a in range(0, n):
        for b in range(0, n):
            # eliminate option of city traveling to itself
```

```
            if a != b:
                # write out all possible edges/distances
                city_input.write(" +" + str(df_dist.iat[a,b]) + "e_" +
                    str(a+1) + "_" + str(b+1))
    # end objective function
    city_input.write("; \n")

def getOneCityEachStep(m, n):
    # go though number of cities want to visit
    for i in range(0, m):
        # go though all cities
        for j in range(0, n):
            # eliminate option of city traveling to itself
            if i != j:
                # write out all possible edges/distances
                city_input.write("+x_" + str(i+1) + "_" + str(j+1) + " ")
        # end objective function
        city_input.write("= 1; \n")

def getEachCityOneVisit(m, n):
    # go though all cities
    for j in range(0, n):
        # go through number of cities want to visit
        for i in range(0, m):
            # eliminate option of city traveling to itself
            if i != j:
                # write out all possible edges/distances
                city_input.write("+x_" + str(i+1) + "_" + str(j+1)  + " ")
        # end objective function
        city_input.write("<= 1; \n")

def getEdgeOneIffTwoStep(m, n):
    for i in range(1, m):
        # go though all cities
        for a in range(1, n+1):
            # go though all cities
            for b in range(1, n+1):
                # eliminate option of city traveling to itself
                if a != b:
                    # write out all possible edge sum
                    city_input.write("e_" + str(a) + "_" + str(b) +
                                     " >= x_" + str(i) + "_" + str(a) +
                                     " + x_" + str(i+1) + "_" + str(b) +
                                     " - 1; \n")

def getBinary(m, n):
```

```
    city_input.write("bin x_1_2")
    # go though all steps
    for i in range(1, m+1):
        # go though all cities
        for j in range(1, n+1):
            # eliminate option of city traveling to itself
            if i != j:
                city_input.write(", x_" + str(i) + "_" + str(j))
    # go though all cities
    for a in range(1, n+1):
        # go though all cities
        for b in range(1, n+1):
            # eliminate option of city traveling to itself
            if a != b:
                city_input.write(", e_" + str(a) + "_" + str(b))
    # end objective function
    city_input.write("; \n")

getObj(n)
getOneCityEachStep(m, n)
getEachCityOneVisit(m, n)
getEdgeOneIffTwoStep(m, n)
getBinary(m, n)

# to ensure correctness/sanity check
city_input = open("cities5.txt", 'r')
```

* * *

Before using lpsolve, I now take the time to show the content of the input file, to see the objective function and the constraints:

Input lpsolve file:

```
(90 of the following e_a_b variables:
distance d_a_b from a to b city)
min: +10800.71e_1_2 +8112.62e_1_3+ ... +212.6e_10_8 +526.07e_10_9;
(5 lines of the following type:
ensure that only one city is visited at each step)
+x_1_2 +x_1_3 +x_1_4 +x_1_5 +x_1_6 +x_1_7 +x_1_8 +x_1_9 +x_1_10 = 1;
.
.
+x_5_1 +x_5_2 +x_5_3 +x_5_4 +x_5_6 +x_5_7 +x_5_8 +x_5_9 +x_5_10 = 1;
(10 of the following type:
ensure that every city is visited once)
```

```
+x_2_1 +x_3_1 +x_4_1 +x_5_1 <= 1;
.
.
+x_1_10 +x_2_10 +x_3_10 +x_4_10 +x_5_10 <= 1;
(90*4 = 360 of the following type:
ensure that an edge variable is accounted for if two cities
are visited in one step)
e_1_2 >= x_1_1 + x_2_2 - 1;
.
.
e_10_9 >= x_1_10 + x_2_9 - 1;
(45 x 90 e of the following types:
ensure that x_i_j and e_a_b are binary in value)
bin x_1_2, x_1_2, x_1_3,..., x_5_8, x_5_9, x_5_10,e_1_2, e_1_3,
e_1_4,..., e_10_7, e_10_8, e_10_9;
```

* * *

Solving the LP with the input file above in lpsolve results in the following output:

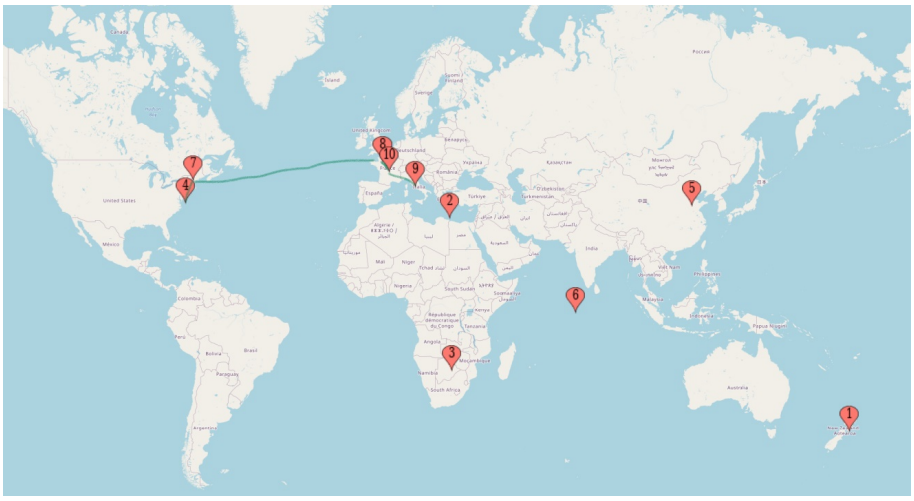
Output from lpsolve for 5 by 5:

```
      Time to load data was 0.000 seconds, presolve used 0.010 seconds,
       ... 0.305 seconds in simplex solver, in total 0.315 seconds.

Value of objective function: 4282.37000000

Actual values of the variables:
e_4_7                          1
e_7_8                          1
e_8_10                         1
e_10_9                         1
x_1_4                          1
x_2_7                          1
x_3_8                          1
x_4_10                         1
x_5_9                          1
```

* * *

Revisiting the Geographical Map in the Introduction Section, I have drawn the output from lpsolve above into the Map (green lines represent the path, and does not have a particular starting point as direction has no relation to distance).

Geographical Map of Output Path (in green) for 5 Cities:

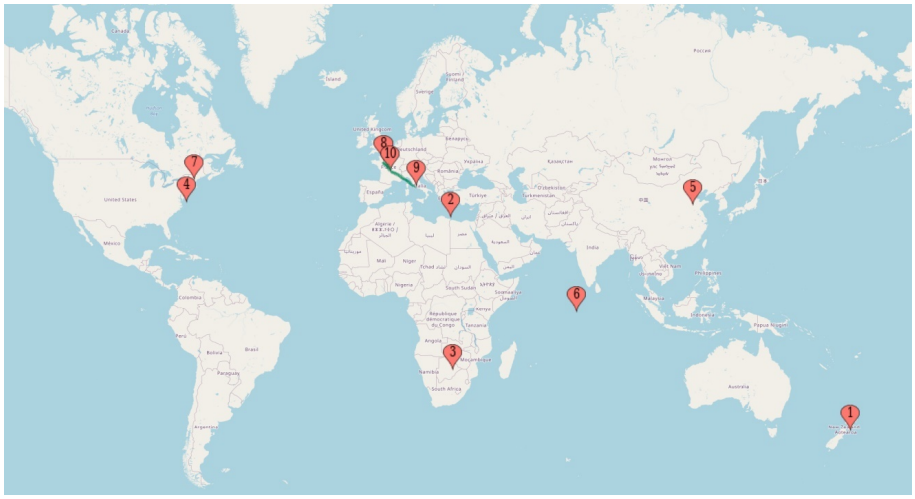


With the map, I can observe 4 & 7 and 8, 9, & 10 in clusters, so it would make sense for them to add small distances to the total (i.e, minimize the total). Although I was itially doubtful that making the jump accross the Atlantic Ocean to the city 4 & 7 cluster was the shortest distance (as opposed to adding city 2 and the next closest city), going back to the Matrix of Distance table in the Introduction Section, I verified that adding city 4 & 7 was really the shortest distance I could add to the existing city 8, 9, & 10 cluster. I now am curious as to see if these clusters are always the first cities visited. I will test for the minimum total distance traveled for 2 all the way to all 10 of the 10 total cities. I do this by changing the m value in the Python code to 2-10 for the other LPs and to run in lpsolve.

Geographical Map of Output Path for 2 Cities:

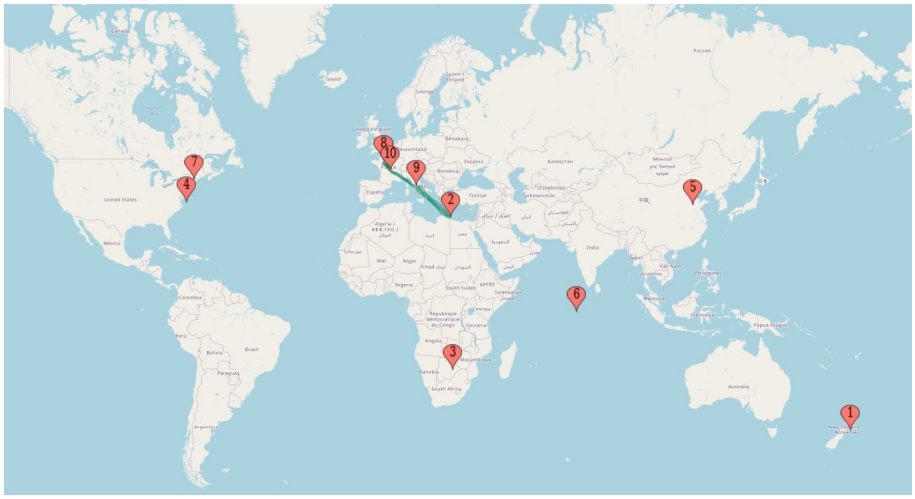
Referring back to the Matrix of Distances between Cities, this output makes sense as the distance between City 8 and 10 is the smallest out of all the distances. I conclude that this result is accurate.

Geographical Map of Output Path for 3 Cities:



Besides looking at the Matrix of Distances, the cluster of 8 9 and 10 on the Geographical Map makes sense to me to be the minimum distance traveled for 3 cities, so I conclude that this result is accurate.

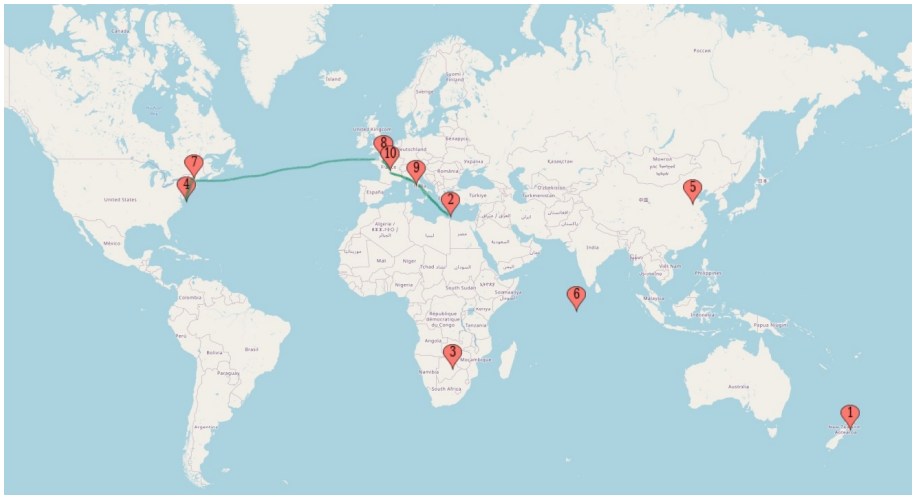
Geographical Map of Output Path for 4 Cities:



As I had previously concluded with the 3 cities total distance, the minimum distance is achieved when traveling from city 9 to 10 to 8, or vice versa. Now

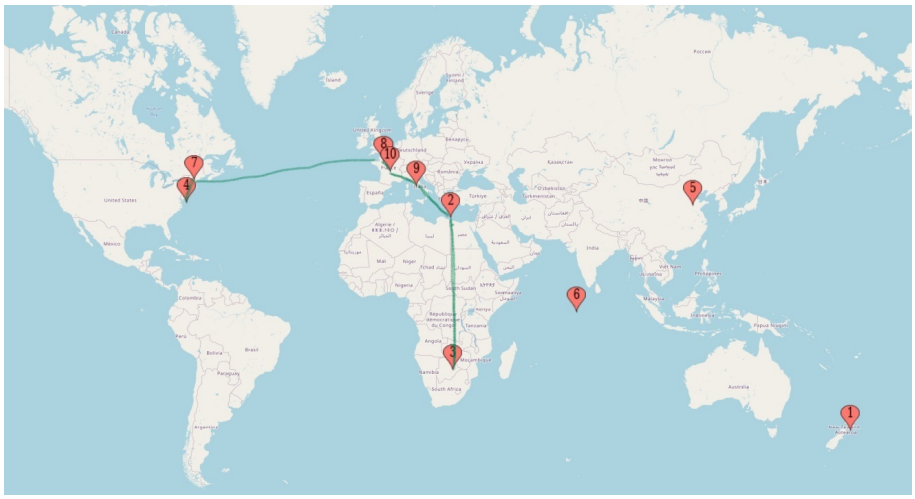
with 4 cities, it makes sense that the minimum would be the addition of the next closest city to this cluster (i.e, city 2). I conclude that this result appears to be accurate.

Geographical Map of Output Path for 6 Cities:



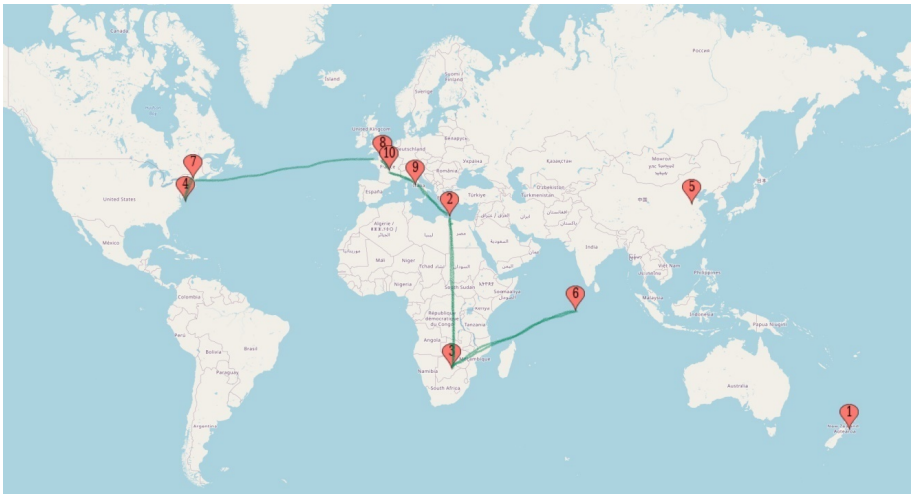
As seen in the Output Path for 5 Cities at the beginning of this Section, the Output changed from the Output Path of 4 Cities' (8, 10, 9, then 2) direction of going East and instead added the city 4 7 cluster. Here, the minimum distance keeps the two clusters and adds on the next closest city (like in 4 Cities), city 2. I conclude that, from my previous conclusions, this resulting path is accurate.

Geographical Map of Output Path for 7 Cities:



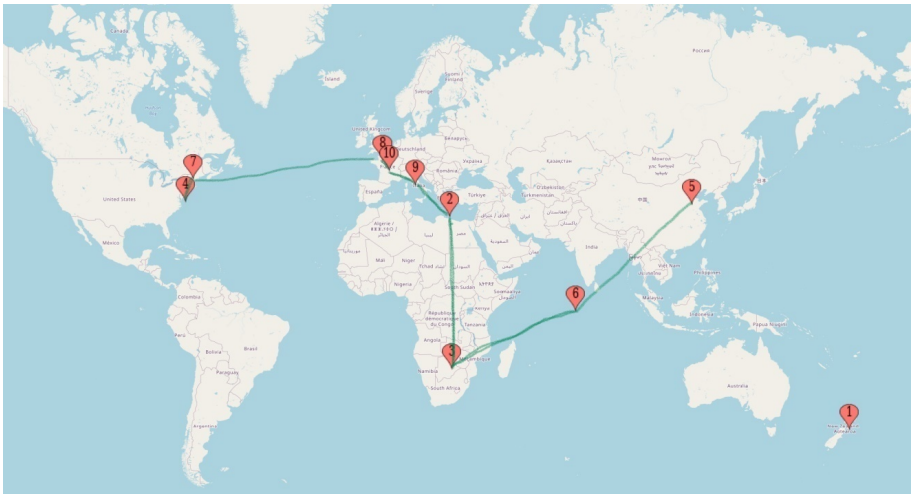
The pattern of adding the next closest city out of the ones yet to be visited continues. When only looking at the Geographical Map, it appeared to be that city 6 was closer to city 2 than city 3 was to city 2, but after referring to the Matrix of Distances table, I confirmed that the next closest city to city 2 was indeed city 3 (city 2 to 3: 3753.93 miles vs. city 2 to 6: 3767.14 miles). I confirm that this resulting path appears to be accurate.

Geographical Map of Output Path for 8 Cities:



This next city being city 6 seemed to be intuitive to me if the pattern was to add on the next closest city yet to be visited. I conclude that this minimum path appears to be accurate.

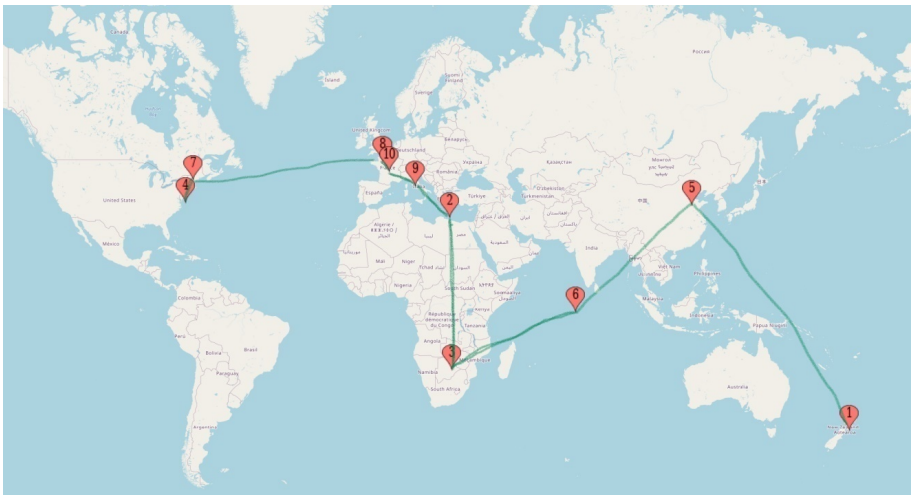
Geographical Map of Output Path for 9 Cities:



The pattern stays the same, as the next step goes to city 5, the next closest city un-visited. Looking at it in a different perspective and referring to the Matrix of Distances table, the shortest path for city 5 to any other city is the edge/distance between city 5 and city 6, so it makes sense that this path would be minimizing the total distance as much as possible. I conclude that this minimum path appears to be accurate.

While running the lpsolver for 10 cities (i.e. the shortest path to visit all cities), the runtime was too long. This last Geographical Visualization of the Output for 10 cities is not a result I was able to get from solving the LP, but rather it is what I would predict if the same pattern of adding the next closest city yet to be visited holds. It also follows the same pattern from the 9 cities output, where the edge/distance from city 5 to city 1 is the shortest distance any city has to city 1 in one step.

Geographical Map of the PREDICTED Output Path for 10 Cities:

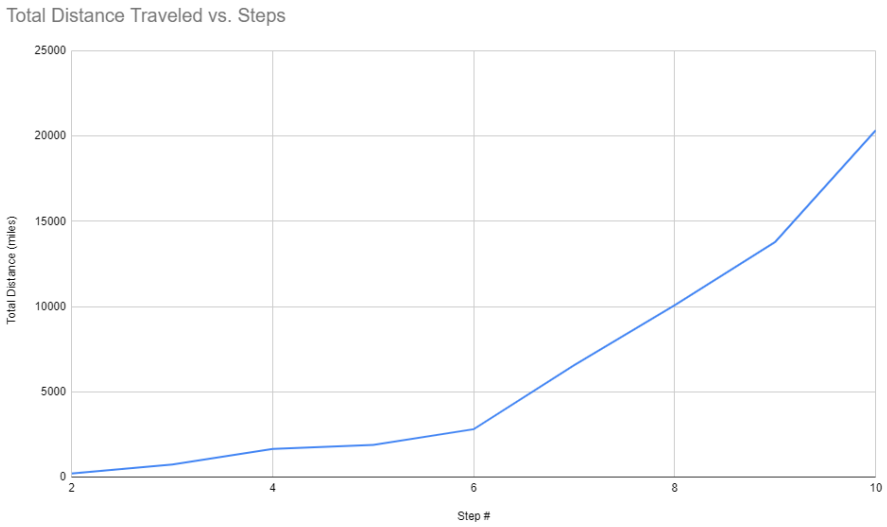


Now that I have the actual paths (and my predicted path for visiting all 10 cities), I have made a table and chart (below) of the number of steps/cities out of the 10 I want to visit, and the minimum distance for that number of steps/cities.

Table of Number of Steps and Total Distance Traveled:

Steps	Total Dist (mile)
2	212.6
3	738.67
4	1663.08
5	1889.46
6	2813.87
7	6567.8
8	10090.22
9	13793.34
10	20347.67

Chart Relation Between Number of Steps and Total Distance Traveled:



As seen in the table and chart, there was initially a slow increase to total distance, but after step 6, there is a sharp increase in the total distance. Looking back on the actual locations of each city, it does make sense to see this increase after all the clusters and nearby cities are already visited.

Although I was able to get my lpsolve to successfully run up to 9 cities, I still would like to try and make my input file efficient. The first adjustment I made

was to rearrange the order of the different constraints (i.e, from the numbered constraints at the very end of the Introduction Section: constraint 3, 1, then 2). I also attempted to add the following constraints (now assuming that distances are not all positive):

I want to ensure that e_{ab} is 0 if I never travel from city a to b in a single step. Let variables f_{iab} (where it uses the same definitions of integer i, a and b) be 1 if and only if I visit city a at step i and city b at step i + 1, and 0 otherwise:

$$3f_{iab} \leq 1 + x_{ia} + x_{i+1b} \quad i = 1, ..., m-1 \tag{5}$$

$$3f_{iab} \geq -1 + x_{ia} + x_{i+1b} \quad i = 1, ..., m-1 \tag{6}$$

$$e_{ab} \leq \sum_{i=1}^{m-1} f_{iab} \tag{7}$$

Adding in the constraints above gives the full LP:

Minimize $\sum_{1 \leq a,b \leq n} d_{ab} e_{ab}$ subject to:

$$\sum_{j=1}^{n} x_{ij} = 1 \qquad i = 1, ..., m \tag{8}$$

$$\sum_{i=1}^{m} x_{ij} \leq 1 \qquad j = 1, ..., n \tag{9}$$

$$e_{ab} \geq x_{ia} + x_{i+1b} - 1 \qquad i = 1, ..., m-1 \tag{10}$$

$$3f_{iab} \leq 1 + x_{ia} + x_{i+1b} \quad i = 1, ..., m-1 \tag{11}$$

$$3f_{iab} \geq -1 + x_{ia} + x_{i+1b} \quad i = 1, ..., m-1 \tag{12}$$

$$e_{ab} \leq \sum_{i=1}^{m-1} f_{iab} \tag{13}$$

$$x_{ij}, e_{ab}, f_{iab} \in \{0, 1\} \qquad 1 \leq i \leq m, 1 \leq a, b, j \leq n \tag{14}$$

Or, with the knowledge that $\sum_{i=1}^{m-1} f_{iab} = 0$ or 1 depending on whether the trip from city a or b was made in one step or not, equation 13 gets its $\leq$ replaced

with = and equation 10 is no longer necessary, to give this full LP:

Minimize $\sum_{1 \leq a,b \leq n} d_{ab} e_{ab}$ subject to:

$$\sum_{j=1}^{n} x_{ij} = 1 \qquad i = 1, ..., m \tag{15}$$

$$\sum_{i=1}^{m} x_{ij} \leq 1 \qquad j = 1, ..., n \tag{16}$$

$$3f_{iab} \leq 1 + x_{ia} + x_{i+1b} \quad i = 1, ..., m-1 \tag{17}$$

$$3f_{iab} \geq -1 + x_{ia} + x_{i+1b} \quad i = 1, ..., m-1 \tag{18}$$

$$e_{ab} = \sum_{i=1}^{m-1} f_{iab} \tag{19}$$

$$x_{ij}, e_{ab}, f_{iab} \in \{0, 1\} \qquad 1 \leq i \leq m, 1 \leq a, b, j \leq n \tag{20}$$

In order to add these equations to the input file, below is the additional Python methods added to the code.

Python code (partial) of methods added for lpsolve:

```
def getIffVisitAThenBLess(m, n):
for i in range(1, m):
    # go though all cities
    for a in range(1, n+1):
        # go though all cities
        for b in range(1, n+1):
            # eliminate option of city traveling to itself
            if a != b:
                # write out all possible edge sum
                city_input.write("3f_" + str(i) + "_" + str(a) + "_"
                                 + str(b) + " <= 1 + x_" + str(i) + "_"
                                 + str(a) + " + x_" + str(i+1) + "_"
                                 + str(b) + "; \n")

def getIffVisitAThenBMore(m, n):
    for i in range(1, m):
        # go though all cities
        for a in range(1, n+1):
```

```
            # go though all cities
            for b in range(1, n+1):
                # eliminate option of city traveling to itself
                if a != b:
                    # write out all possible edge sum
                    city_input.write("3f_" + str(i) + "_" + str(a) + "_"
                                     + str(b) + " >= -1 + x_" + str(i) +
                                     "_" + str(a) + " + x_" + str(i+1)
                                     + "_" + str(b) + "; \n")

def getEdgeVisitAB(m, n):
        # go though all cities
        for a in range(1, n+1):
            # go though all cities
            for b in range(1, n+1):
                # eliminate option of city traveling to itself
                if a != b:
                    city_input.write("+e_" + str(a) + "_" + str(b)
                                     + " = ")
                    for i in range(1, m):
                        # write out all possible edge sum
                        city_input.write("+f_" + str(i) + "_" + str(a)
                                         + "_" + str(b))
                    city_input.write("; \n")

def getBinary(m, n):
    city_input.write("bin x_1_2")
    # go though all steps
    for i in range(1, m+1):
        # go though all cities
        for j in range(1, n+1):
            # eliminate option of city traveling to itself
            if i != j:
                city_input.write(", x_" + str(i) + "_" + str(j))
    # go though all cities
    for a in range(1, n+1):
        # go though all cities
        for b in range(1, n+1):
            # eliminate option of city traveling to itself
            if a != b:
                city_input.write(", e_" + str(a) + "_" + str(b))
    for i in range(1, m+1):
        # go though all cities
        for a in range(1, n+1):
            # go though all cities
            for b in range(1, n+1):
```

```
            # eliminate option of city traveling to itself
            if a != b:
                city_input.write(", f_" + str(i) + "_" + str(a) +
                                 "_" + str(b))
# end objective function
city_input.write("; \n")
```

* * *

All the runs (that were possible to do under 20 minutes) resulted in the same paths as the LP without the additional conditions.

Binding Constraints:

From the 2nd to last full LP (with constraints 8-13), I will take a moment to talk about whether the constraints were binding/non-binding.

Of the constraints that are inequalities, #9 is binding, as the city j could be visited or not, depending on the number of cities I plan to visit (i.e, it could be 0 or 1). Some of the variables' sums would be exactly 1, so this inequality cannot decrease or else the solution would become infeasible.
The same goes for constraints #10, #11, #12, and #13. There are optimal solutions that are exactly equal to the inequality, so decreasing the right side of constraints #11 and #13 and increasing the right side of constraints #10 and #12 would make the LP infeasible.

Below is a table of all the run-times. Where there are blanks in the table, they represent a run-time too large (> 20 minutes) to record the run-time of a completed computation.

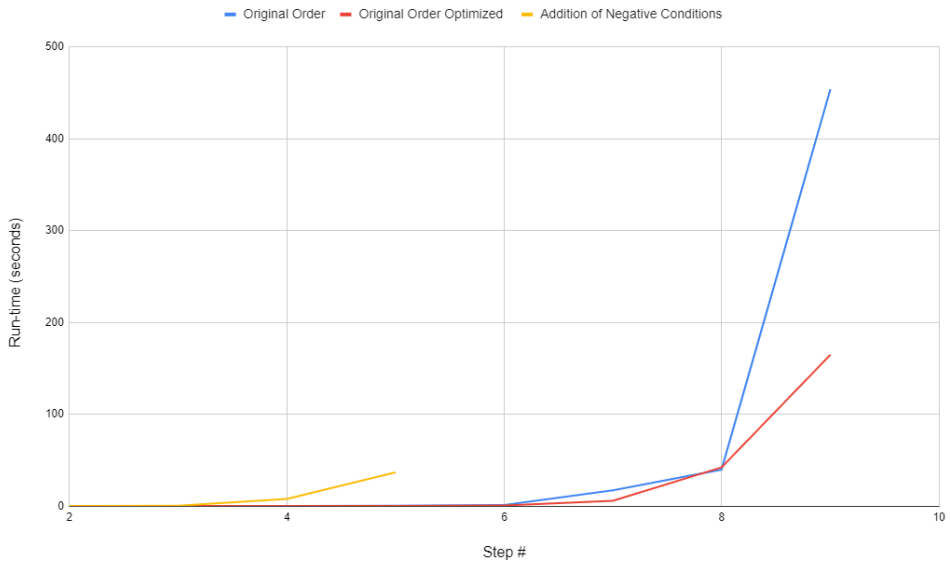
Table of Number of Steps and Run-time (in seconds):

# of Steps	Original Order	Orginal Optimized Order	Addition of Negative Conditions
2	0.012	0.009	0.011
3	0.027	0.011	0.3
4	0.091	0.062	7.764
5	0.309	0.193	36.705
6	1.151	0.733	
7	17.088	5.717	
8	39.625	42.165	
9	453.914	164.754	
10			

I initially believed that, although there is an increase in the number of constraints, it would actually help to decrease the run-time. However, as seen in

the table, the addition of the conditions that check for negative distances actually significantly increased run-time, so much that I was not able to get output for anything higher than 5 cities. One reason I can think is that, in my particular case of the locations and distances between cities (non-negative, and had cities that were clearly closer geographically), there was no need to have those extra conditions, that if anything they were redundant constraints as they gave the same output but with slower computation. Taking those out again and only focusing on the arrangement of #1-3 constraints, I found that placing constraint 3 right after the objective function in the input file gave the fastest run-time (although I was still unable to get anything efficient enough to get a true output for 10 cities). Below are all three run-times on the same chart to give a visual of the differing run-time patterns. They all follow the general trend of gradually (then rapidly) having significantly slower run-times after the addition of another step/city.

Chart of Number of Steps and Run-time (in seconds) for each LP Type/Order:



3 Experiment 2: Rating

Now, rather than caring about how efficient I am being with my traveling (i.e, I no longer care about having the most cost-efficient trip), I now only care about going to the cities I want to go to the most, so long as I do not go over 10,000 (and later 5,000) miles. The cities are rated as follows (1- least want to visit, 10- most want to visit):

- City #1: 1
- City #2: 5
- City #3: 7
- City #4: 2
- City #5: 8
- City #6: 3
- City #7: 4
- City #8: 10
- City #9: 9
- City #10: 6

I now have a new LP with similar constraints:

Let there be integer m, n, i, j, a, and b defined the same as the Introduction Section, as well as variables x_{ij}, d_{ab} and e_{ab}. However, I now define r_j, where each r variable from j = 1 to 10 is the coefficient value of how much I want to visit that city/landmark (1- lowest, 10-highest).

My objective function then is defined as:

$$\sum_{i=1}^{m}\sum_{j=1}^{n} r_j x_{ij}$$

Where the function is constrained:

Where it follows the same constraints as the total distance experiment (#8, 9, and 10) There can only be one city visited at each individual step, I visit each city once, and ensure 1 step traveling for city a to b is accounted as value 1 in e_{ab}:

$$\sum_{j=1}^{n} x_{ij} = 1, i = 1, ..., m$$

$$\sum_{i=1}^{m} x_{ij} \leq 1, j = 1, ..., n$$

$$e_{ab} \geq x_{ia} + x_{i+1b} - 1, i = 1, ..., m-1$$

I also add one more constraint, where I need to ensure that the total distance between the path with the places I want to go to the most does not exceed 10,000 miles:

$$\sum_{a=1}^{n}\sum_{b=1}^{n} d_{ab} e_{ab} \leq 10,000$$

Therefore, the LP is:

Maximize $\sum_{i=1}^{m} \sum_{j=1}^{n} r_j x_{ij}$ subject to:

$$\sum_{j=1}^{n} x_{ij} = 1 \qquad i = 1, ..., m \tag{21}$$

$$\sum_{i=1}^{m} x_{ij} \leq 1 \qquad j = 1, ..., n \tag{22}$$

$$e_{ab} \geq x_{ia} + x_{i+1b} - 1 \qquad i = 1, ..., m-1 \tag{23}$$

$$\sum_{a=1}^{n} \sum_{b=1}^{n} d_{ab} e_{ab} \leq 10,000 \tag{24}$$

$$x_{ij}, e_{ab} \in \{0, 1\} \qquad 1 \leq i \leq m, 1 \leq a, b, j \leq n \tag{25}$$

I use the LP above to solve my desire of wanting to travel my top 5 of the 10 cities/landmarks I want to visit under 10,000 miles. Below is the Python code for the input file:

Python code to solve LP in lpsolve:

```
import pandas as pd
# open pre-made txt file to write in
city_input = open("rate5.txt", 'w')

# number of cities want to visit
m = 5

# total number of cities (does not change)
n = 10

# max dist
d = 10000

# start LP line for maximizing value
ratings = [1, 5, 7, 2, 8, 3, 4, 10, 9, 6]
city_input.write("max: ")
# function to get obective function
def getObj(m, n):
    # go though each city path
    for a in range(0, m):
```

```
        for b in range(0, n):
            # eliminate option of city traveling to itself
            if a != b:
                # write out all possible verticies/ratings
                city_input.write(" +" + str(ratings[b]) + "x_" +
                                 str(a+1) + "_" + str(b+1))
    # end objective function
    city_input.write("; \n")

# read in matrix of edge weight/distance
df_dist = pd.read_csv('/home/jovyan/lost+found/MATH_381/locations.csv',
                      header = None)

def getMaxDist(m, n, d):
    # go though each city path
    for a in range(0, n):
        for b in range(0, n):
            # eliminate option of city traveling to itself
            if a != b:
                # write out all possible edges/distances
                city_input.write(" +" + str(df_dist.iat[a,b]) + "e_"
                                 + str(a+1) + "_" + str(b+1))
    city_input.write("<= " + str(d) + "; \n")

def getOneCityEachStep(m, n):
    # go though number of cities want to visit
    for i in range(0, m):
        # go though all cities
        for j in range(0, n):
            # eliminate option of city traveling to itself
            if i != j:
                # write out all possible edges/distances
                city_input.write("+x_" + str(i+1) + "_" + str(j+1) + " ")
        # end objective function
        city_input.write("= 1; \n")

def getEachCityOneVisit(m, n):
    # go though all cities
    for j in range(0, n):
        # go through number of cities want to visit
        for i in range(0, m):
            # eliminate option of city traveling to itself
            if i != j:
                # write out all possible edges/distances
                city_input.write("+x_" + str(i+1) + "_" + str(j+1)  + " ")
        # end objective function
```

```
        city_input.write("<= 1; \n")

def getEdgeOneIffTwoStep(m, n):
    for i in range(1, m):
        # go though all cities
        for a in range(1, n+1):
            # go though all cities
            for b in range(1, n+1):
                # eliminate option of city traveling to itself
                if a != b:
                    # write out all possible edge sum
                    city_input.write("e_" + str(a) + "_" + str(b) +
                                     " >= x_" + str(i) + "_" + str(a) +
                                     " + x_" + str(i+1) + "_" + str(b) +
                                     " - 1; \n")

def getBinary(m, n):
    city_input.write("bin x_1")
    # go though all steps
    for i in range(1, m+1):
        # go though all cities
        for j in range(1, n+1):
            # eliminate option of city traveling to itself
            if i != j:
                city_input.write(", x_" + str(i) + "_" + str(j))
    # go though all cities
    for a in range(1, n+1):
        # go though all cities
        for b in range(1, n+1):
            # eliminate option of city traveling to itself
            if a != b:
                city_input.write(", e_" + str(a) + "_" + str(b))
    # end objective function
    city_input.write("; \n")

getObj(m, n)
getEdgeOneIffTwoStep(m, n)
getOneCityEachStep(m, n)
getEachCityOneVisit(m, n)
getMaxDist(m, n, d)
getBinary(m, n)
```

* * *

I chose not analyze run-time for the m number of ratings, as there was never an issue of the computing time being too long for any of the input files I utilized.

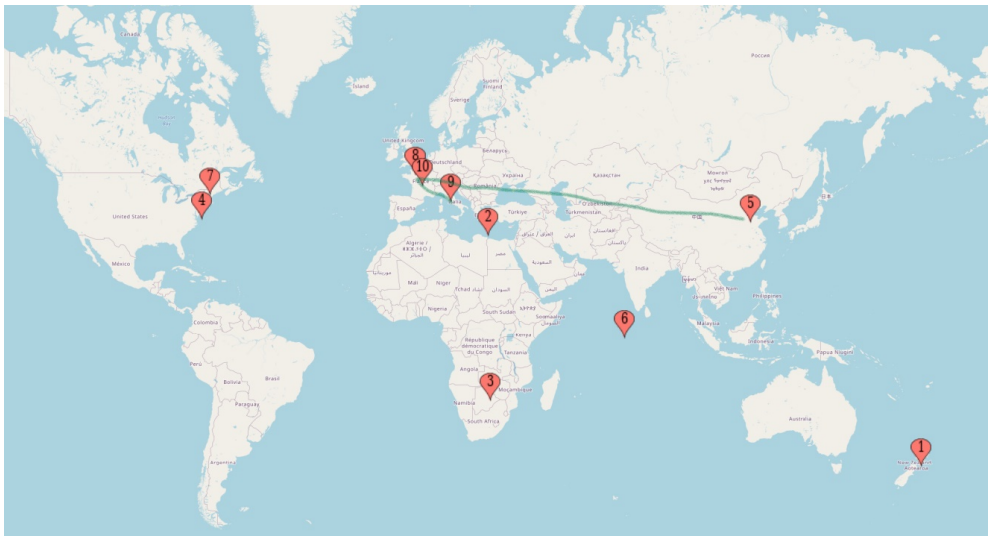
The previous code and input file gave the path of travel. The table below is the number of cities to visit and the path of travel taken (i.e, the numbers correspond to city # not rating). Here you can see that, as City #1 is the furthest and also had the lowest rating, it is never added in the list of places to visit:

Table of # of Cities and Path of Travel for 10000 & 5000 miles Constraint:

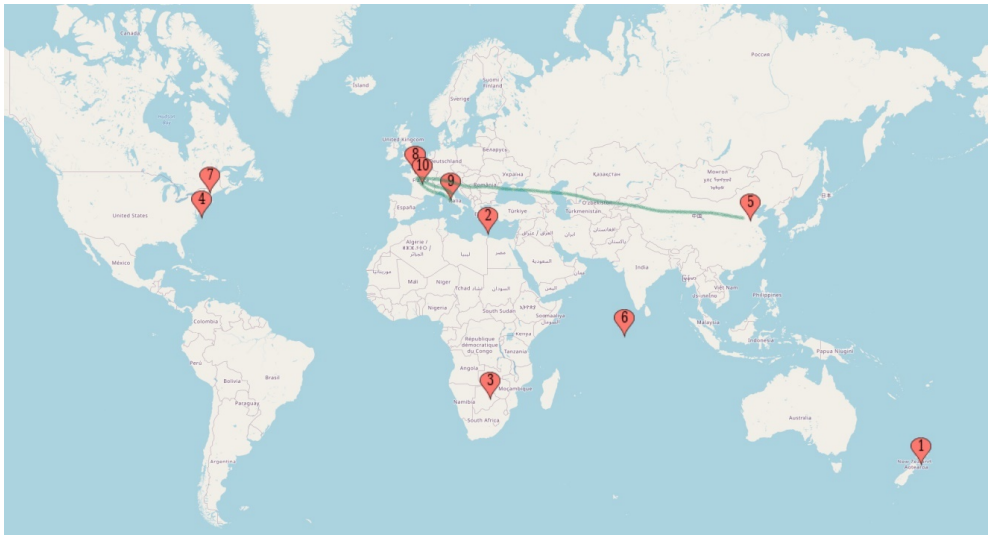
Number of cities	10000 miles	5000 miles
3	5,8,9	8,9,10
4	8,9,10,5	8,9,2,10
5	5,10,9,8,2	2,9,10,8,7
6	7,8,10,9,2,5	Infeasible
7	4,7,8,10,9,2,5	
8	Infeasible	

Here, I add in Geographical Maps of the path taken to show how the total distance is under 10,000 or 5,000 miles (for verification):

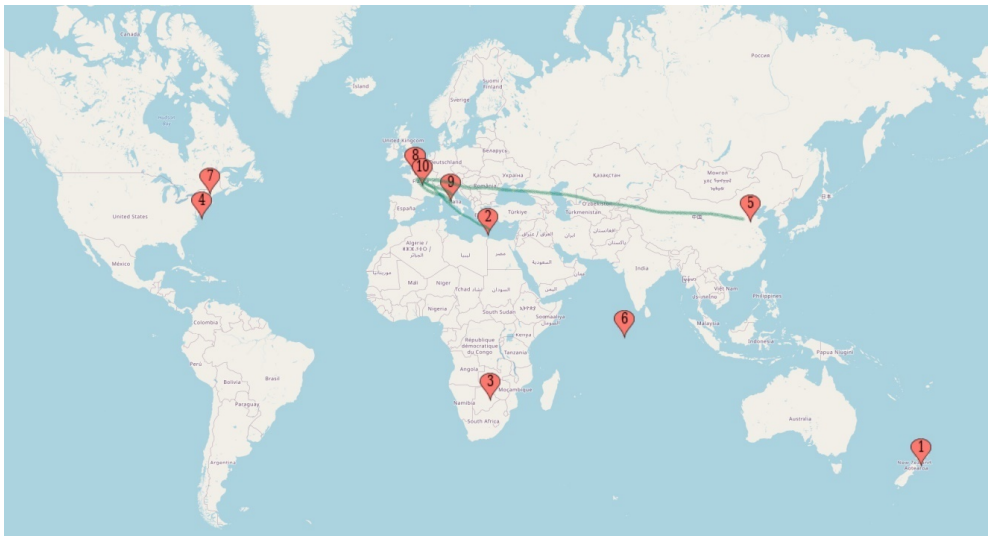
Map of 3 Cities and Path of Travel for 10,000 miles Constraint:



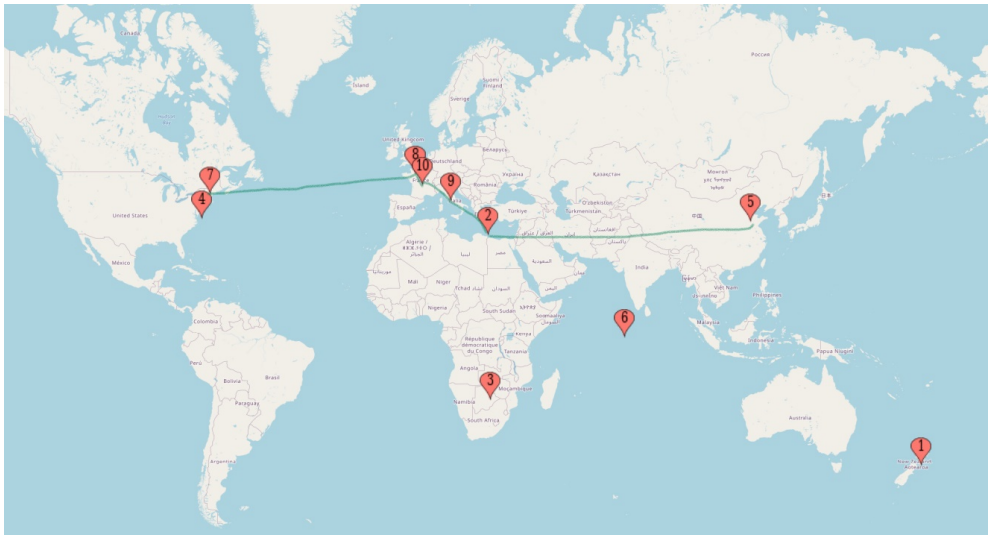
Map of 4 Cities and Path of Travel for 10,000 miles Constraint:



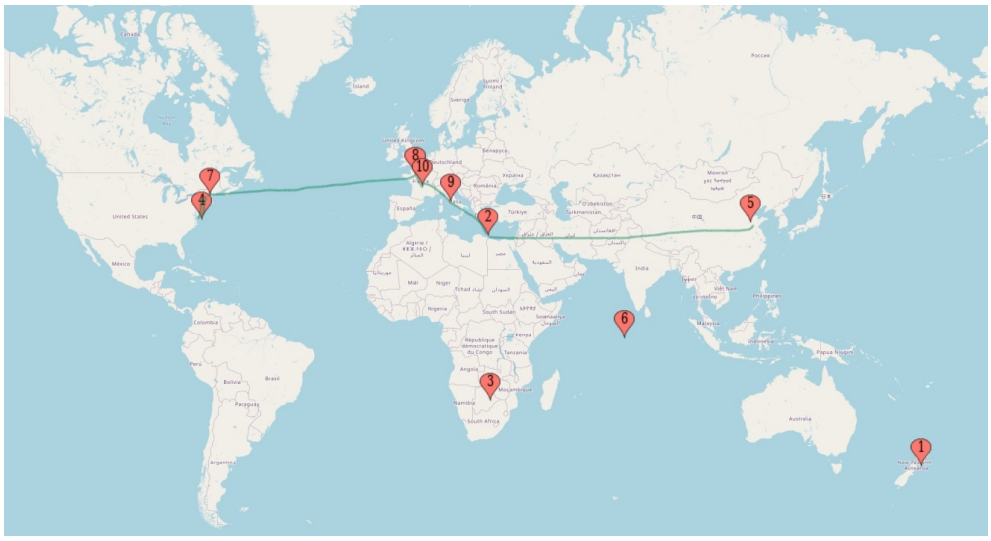
Map of 5 Cities and Path of Travel for 10,000 miles Constraint:



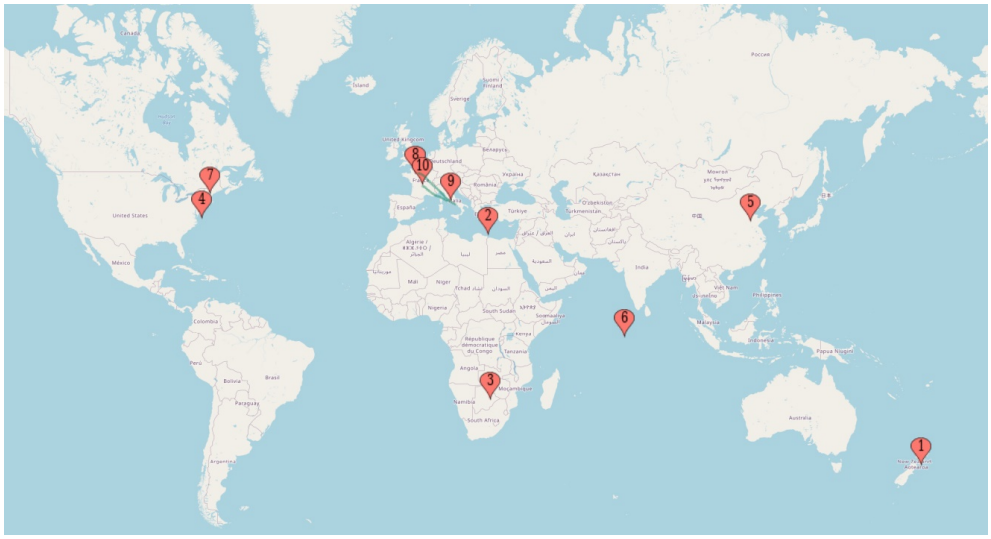
Map of 6 Cities and Path of Travel for 10,000 miles Constraint:



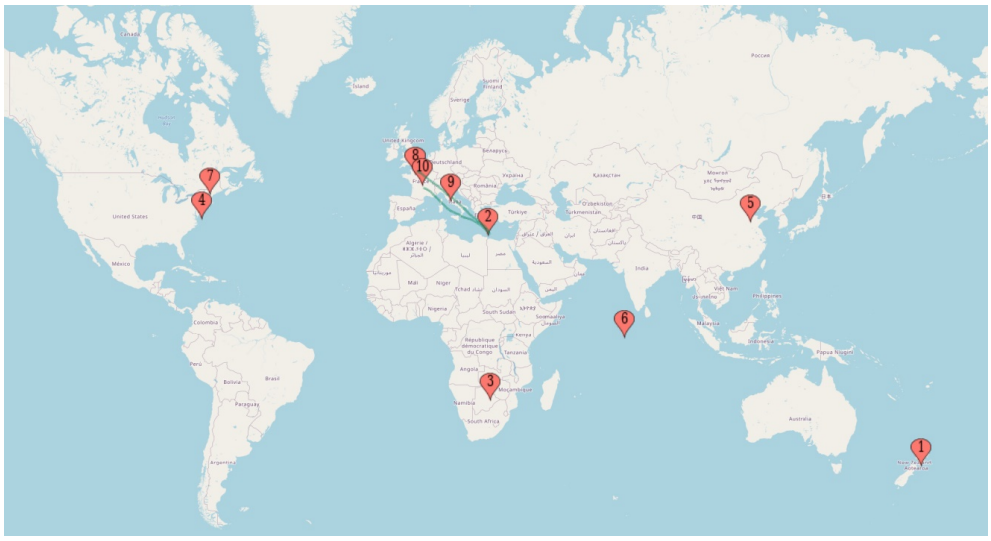
Map of 7 Cities and Path of Travel for 10,000 miles Constraint:



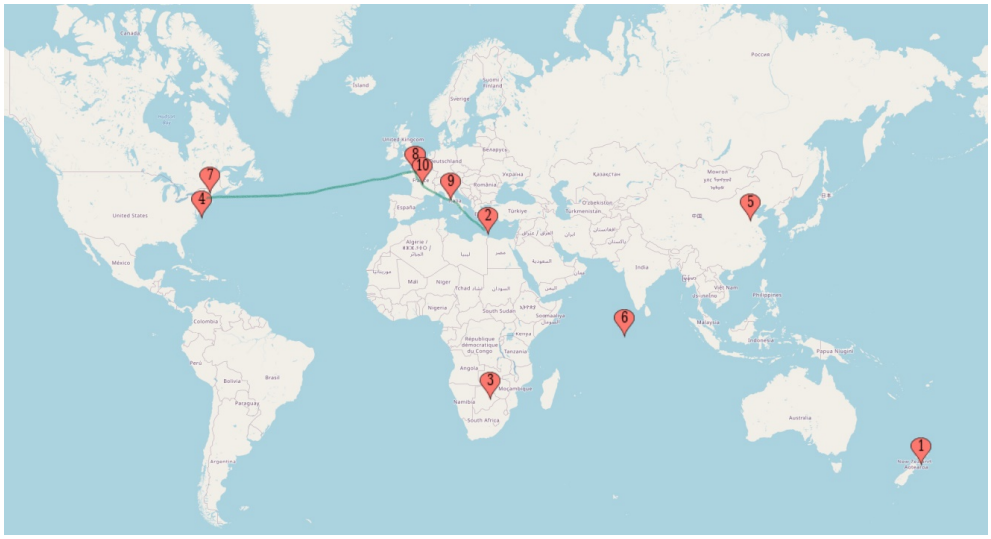
Map of 3 Cities and Path of Travel for 5,000 miles Constraint:



Map of 4 Cities and Path of Travel for 5,000 miles Constraint:



Map of 5 Cities and Path of Travel for 5,000 miles Constraint:

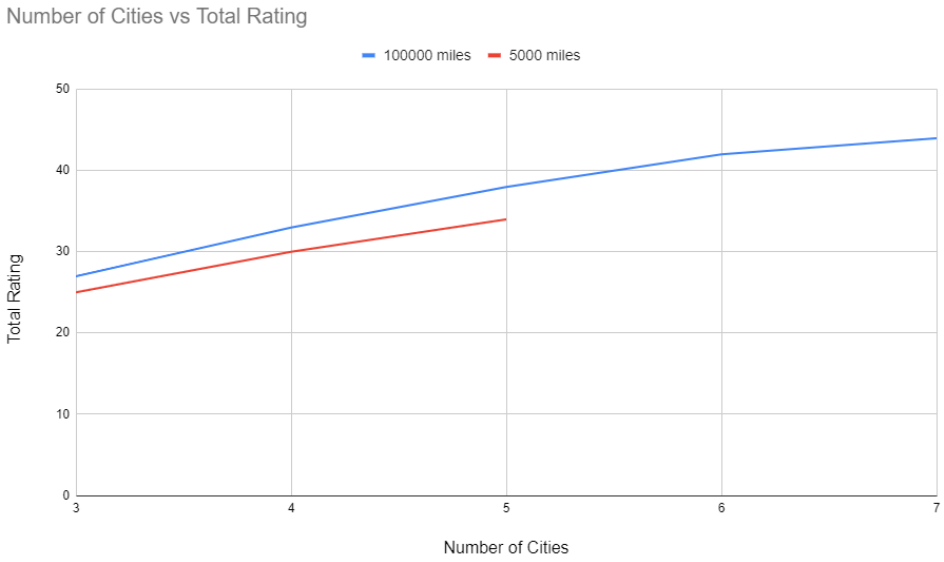


From the city #, I then connected to the ratings and totaled the ratings. The table and chart below the first one shows ("infeasible" states that there can no longer be any addition of cities or else it will go beyond the max total miles traveled constraint)

Table of # of Cities and Total Rating for 10000 & 5000 miles Constraint:

Number of cities	10000 miles	5000 miles
3	27	25
4	33	30
5	38	34
6	42	infeasible
7	44	
8	infeasible	

Chart of # of Cities and Total Rating for 10000 & 5000 miles Constraint:



As seen in the table and chart, the ratings increase slowly, as the total already starts very high with the highest/largest rated cities. In other words, when distance is not an issue (low of cities), the cities with the highest ratings (city 5, 8, and 9) will be visited. As seen in the 5,000 miles constraint, city 5 is too far to have the combination be under 5,000 miles (I refer to the Matrix of Distances and Geographical Maps above mapping the path to confirm that the distance was satisfied and the ratings are optimized).

Main Takeaways:

I think the main takeaway that I received was that there was less variation after each of cities are visited than I had thought. However, when considering my map and the clustering and location of the cities/landmarks, it seems more intuitive/makes more sense logically than I had realized.