UCLA

ECE 232 - Large Scale Social and Complex Networks: Design and Algorithms
Spring 2021

Project 4: Graph Algorithms

Authors:
Swapnil Sayan Saha (UID: 605353215)
Grant Young (UID: 505627579)

Question 1:

In this question, we asked to provide the range of correlation coefficient ρ_{ij} between the log-normalized stock-return time series data of stocks i and j and also provide a justification for using log-normalized stock return instead of regular return. The log normalized stock return $r_i(t)$ of stock i over a period of $[t-1, t]$ is given by:

$$r_i(t) = \log(1 + q_i(t))$$

where,

$$q_i(t) = \frac{p_i(t) - p_i(t-1)}{p_i(t-1)}$$

$q_i(t)$ is the regular return of stock i over a period of $[t-1, t]$, $p_i(t)$ is the closing price of stock i on t^{th} day. ρ_{ij} is thus given by:

$$\rho_{ij} = \frac{\langle r_i(t) r_j(t) \rangle - \langle r_i(t) \rangle \langle r_j(t) \rangle}{\sqrt{(\langle r_i(t)^2 \rangle - \langle r_i(t)^2 \rangle)(\langle r_j(t)^2 \rangle - \langle r_j(t)^2 \rangle)}}$$

The numerator refers to the covariance of $r_i(t)$ and $r_j(t)$, where as the denominator refers to the product of the standard deviation of $r_i(t)$ and $r_j(t)$. ρ_{ij} provides the a measure of linear correlation among $r_i(t)$ and $r_j(t)$, and is identical to the Pearson's correlation coefficient, essentially providing a normalized measurement of the covariance. As a result, the upper bound of ρ_{ij} is always +1 and the lower bound of ρ_{ij} is always -1, according to Cauchy-Schwarz Inequality. When ρ_{ij} is +1, it indicates that the two stocks i and j are strongly positively correlated, with their prices altering in the same direction (though the scale of growth may be different). When ρ_{ij} is -1, it means that the two stocks i and j are strongly negatively or inversely correlated, with the stock prices moving in opposite directions.

There are several reasons why we prefer the log-normalized return over regular return:

- Reduces the effects of outliers and variance in the data, thus constraining the values within finite bounds.
- Reduces any skewness that may be present in the data.
- Better able to represent relative changes in the stock prices over absolute numbers. Relative changes are more practical as the percentage scale of those changes would be the same for stocks that are usually treated in the same way.
- Amplifies the scale of smaller stocks and shrinks the scale of larger stocks, thus providing a homogenous scale to visualize the relative changes of all stocks regardless of their prices.

Question 2:

In this question, we are asked to plot a histogram showing the un-normalized distribution of edge weights in the correlation graph. The correlation graph consists of the stocks as vertices and the edge weights as follows:

$$w_{ij} = \sqrt{2(1 - \rho_{ij})}$$

- If ρ_{ij} is +1 (strong positive and perfect correlation), w_{ij} = 0.
- If ρ_{ij} is -1 (strong negative correlation), w_{ij} = 2.

Thus, w_{ij} should be between 0 and 2. To construct the graph, we first omit the stocks with missing data (NaN values) and only keep those stocks with 765 entries. We then calculate vector of $r_i(t)$, containing log-normalized return for each stock i. Afterwards, we create a file to store the w_{ij} (calculated using the formula shown above) for each stock i and j, where stock i is the source node and stock j is the sink node. We use the `igraph` library in R for generating the graph. Figure 1 shows the histogram plot.

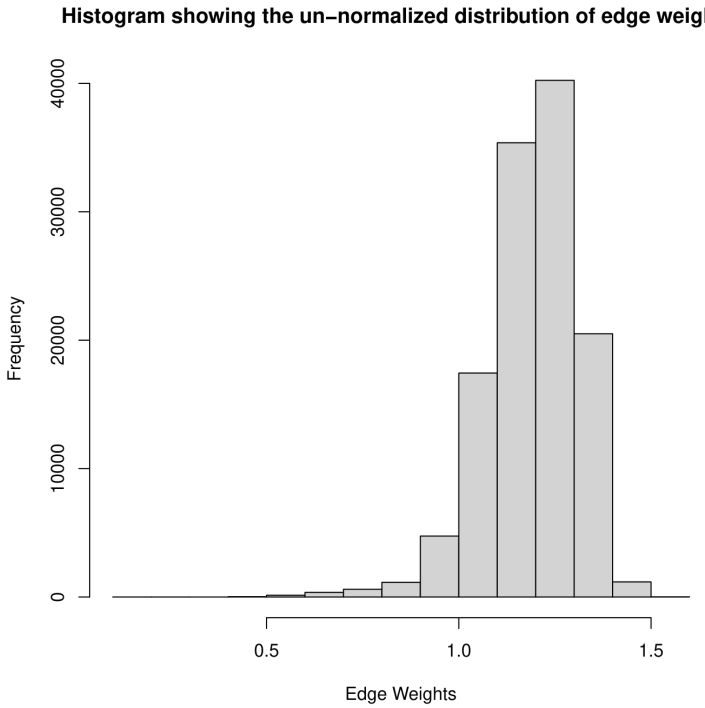


Figure 1: Histogram showing the un-normalized distribution of edge weights.

From Figure 1, we observe that most of the edge weights are between 1.0 and 1.4. As expected, all the weights lie between 0 and 2, however, we observe that the lower bound for edge weights is 0.5 and not 0. This means that there are no stocks with perfect positive correlation, as this would require some of the edge weights to be 0. Most

of the stocks are weakly correlated with each other, and majority of the stocks have negative correlation with each other.

Question 3:

In this question, we are asked to plot the minimum spanning tree (MST) for the correlation graph, color coded to show different sectors (categories) of stocks. The MST connects all the nodes of a connected and edge-weighted undirected graph without any cycles and with the lowest possible cumulative edge weights. We use Prim's algorithm to create the MST from the correlation graph. Figure 2 shows the MST for the correlation graph, with stocks belonging to the same sector color coded the same.

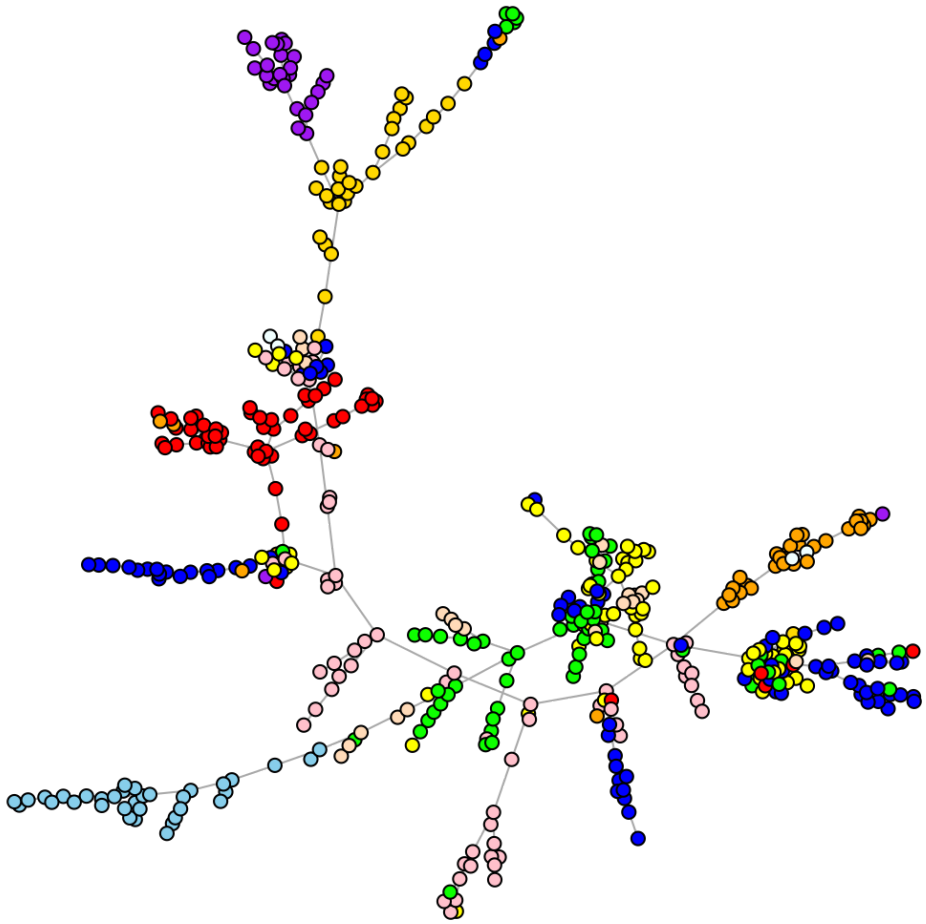


Figure 2: MST of the correlation graph for daily data.

We can make several observations and patterns from Figure 2. Most of the stocks that have a common color (belong to the same sector) tend to flock together in the MST. On the other hand, nodes of different colors (belonging to dissimilar sectors) are not connected to each other. In other words, stocks that are highly correlated tend to be connected on the correlation graph with the least possible edge weights (the higher the correlation ρ_{ij} between stocks,, the lower the edge weights w_{ij}, as $w_{ij} = \sqrt{2(1-\rho_{ij})}$. On the other hand,

the stocks which are not strongly correlated have large edge weights. Since the goal of MST is to connect all nodes with lowest cumulative edge weights, it tends to cluster the highly correlated stocks together. Such structures are referred to as vine clusters, as they look like grapes hanging off a main branch. The clusters represent distinct sectors. Stocks belonging to the same cluster tend to have changes in the same direction and thus require similar investment strategies. Vine clusters provide a picture of the stock market on a relatively longer time-scale such as daily data.

Question 4:

In this question, we are asked to evaluate two sector clustering techniques of an unknown stock v_i in the MST. The two methods are:

1. $P(v_i \in S_i) = \frac{|Q_i|}{|N_i|}$ (technique 1)
2. $P(v_i \in S_i) = \frac{|S_i|}{|V|}$ (technique 2)

where, S_i is the sector of node i, Q_i is the set of neighbors of node i that belong to the same sector as node i, N_i is the set of all neighbors of node i, V is the set of all vertices of the MST. The performance metric for each of the methods is as follows:

$$\alpha = \frac{1}{|V|} \sum_{v_i \ inV} P(v_i \in S_i)$$

The higher the value of α, the more effective is the clustering technique.The values of α for the two methods are:

1. Technique 1: 0.828930
2. Technique 2: 0.114188

We observe that technique 1 provides a much higher value of α than technique 2. This is expected because technique 1 exploits the MST vine cluster structures of the correlation graph, taking into account which nodes are highly correlated and flock together instead of all the nodes. In other words, technique 1 exploits local connectivity among neighboring nodes instead of the global correlation graph to make decisions. On the other hand, technique 2 considers all the nodes that belong to a sector and fails to extract local spatial connections or cluster formations by taking into account the entire graph. Technique 2 thus provides only a general probability estimate.

Question 5:

In this question, we are asked to plot the MST for the correlation graph for weekly data. The weekly correlation graph is created by subsampling the stock data weekly on Mondays, ignoring those weeks where Monday is a holiday. We omit the stocks with missing data (NaN) when creating the correlation graph, only keep those files with 143 entries and use Prim's algorithm to create the MST from the correlation graph. Figure 3 shows the MST with stocks in the same sector color-coded the same.

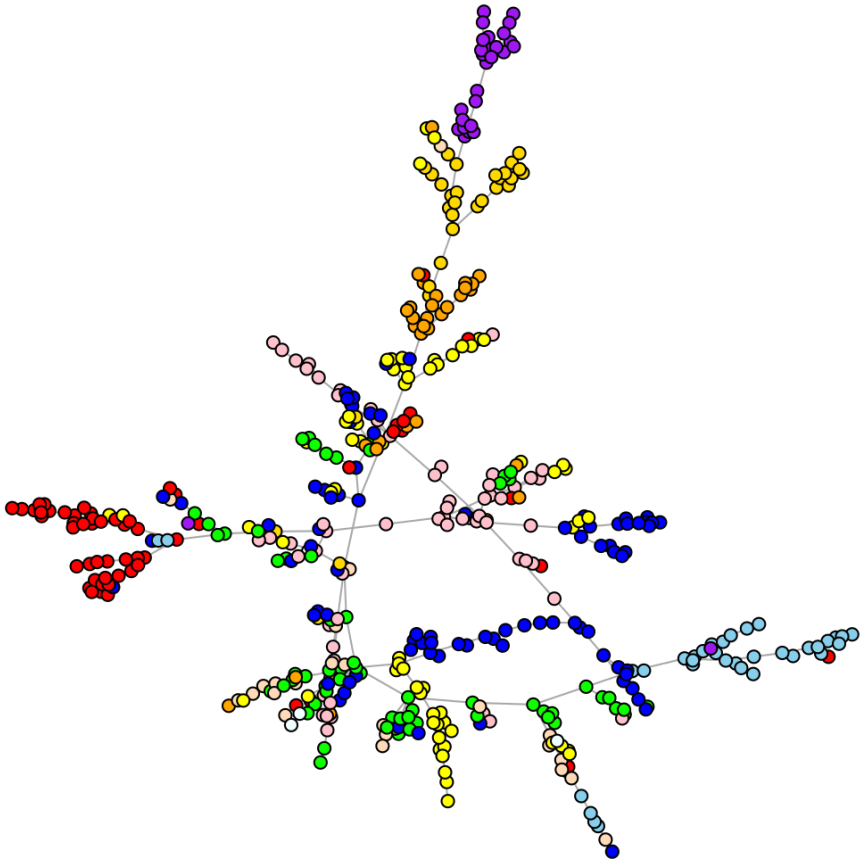


Figure 3: MST of the correlation graph for weekly data.

There are several differences in the pattern that we can observe between the MST in Figure 3 and the MST in Figure 2. Although some of the stocks are still forming vine clusters, a significant number of stocks belonging to the same sector are not forming clusters, with the nodes not forming clearly separable regions in the MST graph. This indicates that clustering is better with daily data than weekly data. In other words, stocks from the same sector lose their correlation when the time-scale increases, causing the edge weights in the correlation graph to increase among them even though they belong to the same node. In addition, it means that it will be more difficult to assign a sector to an unknown stock if the data is sampled weekly. We also observe a circular structure

in Figure 3 compared to Figure 1. The presence of loops in the MST indicate that most of the edge weights are similar regardless of the sector to which the stocks belong to.

Question 6:

In this question, we asked to report the number of nodes and edges of the graph that we constructed from the Uber Dataset for the month of December. The nodes correspond to locations (latitude and longitude), and the edges represent mean travel times between the locations. For simplicity, we remove isolated nodes, merge duplicate edges by averaging their weights and keep only the giant connected component of the graph. We use the '`los_angeles-censustracts-2019-4-All-MonthlyAggregate.csv`' file to import the source ID, destination ID and mean travel times to construct the graph, using the `igraph` library in Python. The properties of the graph are as follows:

- Number of nodes: 2649
- Number of edges: 1004955

Question 7:

In this question, we are asked to build a MST from the graph we constructed in Question 6 and report street addresses of two endpoints for a few edges. The MST is shown in Figure 4.

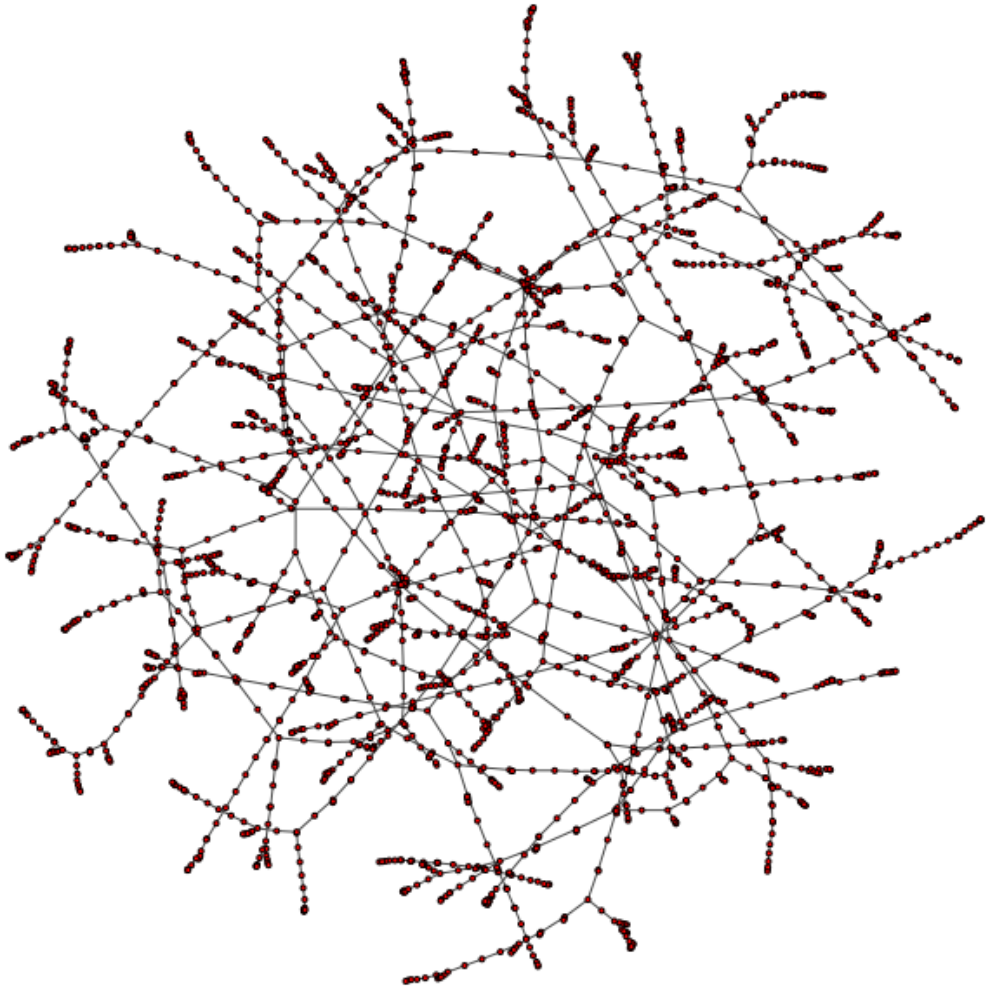


Figure 4: MST for the Uber Dataset Graph.

To report the street address for a few edges and show them on a map, we took the following steps:

- Print the coordinates (longitude and latitude) near the two endpoints for a few edges.
- Use the following website to get the street address for the coordinates: https://www.gps-coordinates.net/
- Plot the street address on a map for each pair of endpoints using the following website: https://www.mapcustomizer.com/

Table 1 shows the coordinates and street address near the centroid locations for a few edges.

Table 1: Street addresses near the two endpoints (the centroid locations) of a few edges

ID	Census Tract (a)	Coordinates (a) (Lon., Lat.)	Census Tract (b)	Coordinates (b) (Lon., Lat.)	Street Address (a)	Street Address (b)
1	554001	[-118.133298 33.904119]	554002	[-118.141448 33.896526]	9421, Rosecrans Avenue, Bellflower, California, 90706, United States of America	9020, Somerset Boulevard, Bellflower, Los Angeles County, California, United States of America
2	461700	[-118.159325 34.153604]	460800	[-118.172425 34.180286]	500, Prospect Boulevard, Pasadena, California, 91103, United States of America	Brookside Golf Club, Arroyo Woods, Pasadena, CA 91109, United States of America
3	302201	[-118.251349 34.146334]	302202	[-118.248811 34.142665]	First United Methodist Church of Glendale, 134 North Kenwood Street, Glendale, CA 91206, United States of America	East Colorado Street, Glendale, CA 91205, United States of America
4	407101	[-117.967696 34.04101]	407002	[-117.980572 34.051745]	619 North Sunset Avenue, La Puente, CA 91744, United States of America	652 Puente Avenue, El Monte, CA 91746, United States of America
5	433401	[-118.043316 34.058606]	433402	[-118.038157 34.056957]	10468 Fern Street, South El Monte, CA 91733, United States of America	2433 Continental Avenue, El Monte, CA 91733, United States of America
6	543603	[-118.28871 33.803659]	294410	[-118.289861 33.797764]	Harbor Freeway, West Carson, CA 90710, United States of America	828 Lomita Boulevard, West Carson, CA 90710, United States of America
7	482001	[-118.165951 34.062293]	530700	[-118.176633 34.062274]	I-10 Metro ExpressLanes, Los Angeles, CA 90032, United States of America	4225 Drucker Street, East Los Angeles, CA 90032, United States of America
8	460800	[-118.172425 34.180286]	463800	[-118.167761 34.141469]	Brookside Golf Club, Arroyo Woods, Pasadena, CA 91109, United States of America	Lower Arroyo Park, 415 Arroyo Boulevard, Pasadena, CA 91105, United States of America
9	269100	[-118.398461 34.057095]	217001	[-118.385502 34.04886]	475 Smithwood Drive, Beverly Hills, CA 90212, United States of America	Robertson & Airdrome, Robertson Boulevard, Los Angeles, CA 90035-4232, United States of America
10	011000	[-117.950433 33.881015]	001901	[-117.965225 33.868368]	1382 West Valley View Drive, Fullerton, CA 92833, United States of America	2069 West Walnut Avenue, Fullerton, CA 92833, United States of America

Figure 5 shows the map of each pair of endpoints.

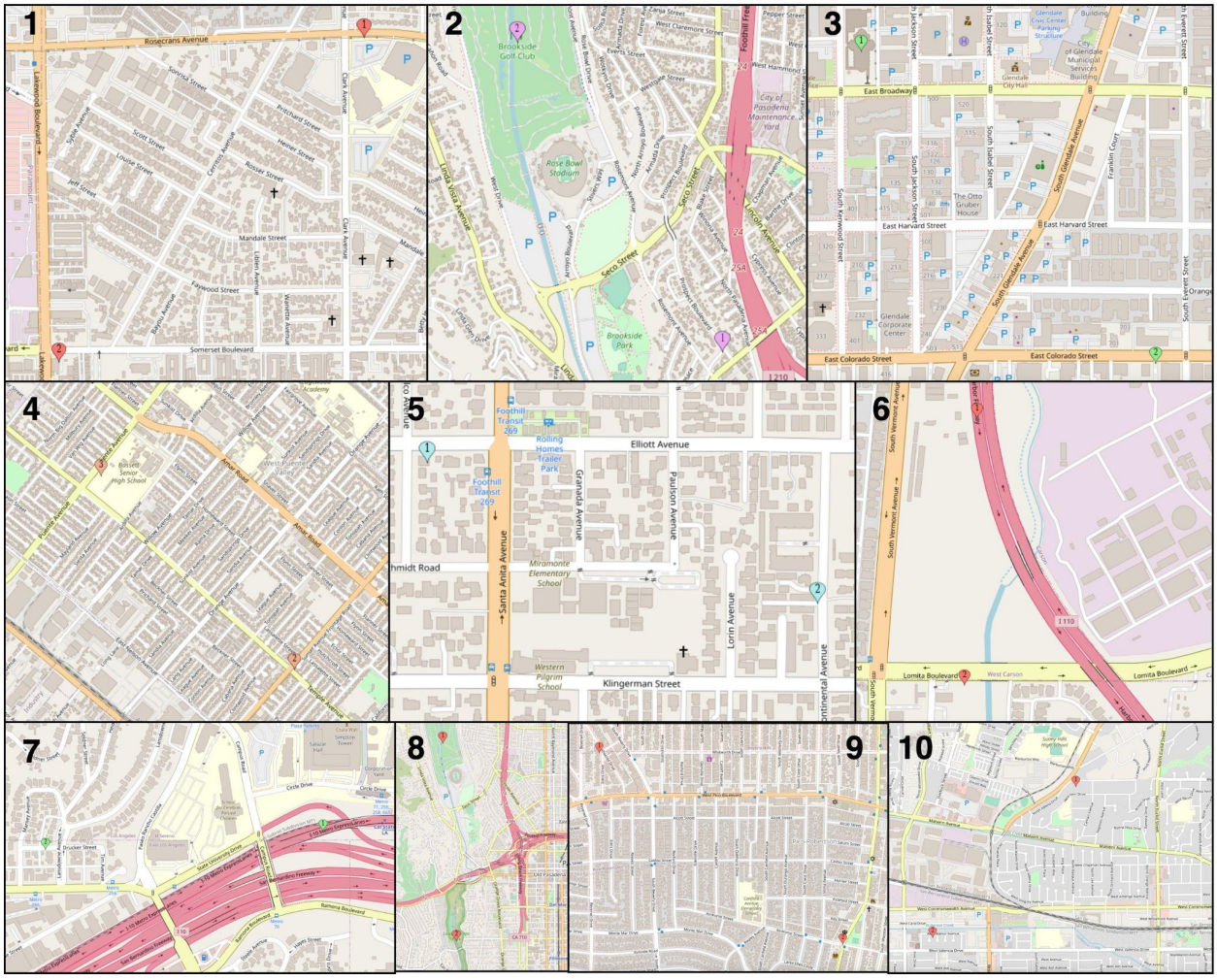


Figure 5: Maps showing each pair of endpoints, the ID corresponding to each pair is provided on each image.

From Table 1 and Figure 5, we can see that the street address of the endpoints are very close to each other. This makes intuitive sense because the MST is supposed to connect all the addresses such that the lowest cumulative mean travel time (lowest weight of edges) is achieved. An easy way to achieve this is to connect the nodes that have a small distance between them as the mean travel time between those nodes will be smaller as well. Thus, Prim's algorithm chooses the edges with the smallest weights (mean travel times), ultimately choosing nodes with a small pairwise distance between them. In fact, we observed that the average pairwise distance among the endpoints is around 0.7 miles.

Question 8:

The triangle inequality states that the sum of any two sides of a triangle must always be greater than the third side:

$$a + b > c, \quad b + c > a, \quad a + c > b$$

In the case of our graph, it means the mean travel time between two addresses will always be smaller than the mean travel time between those addresses via a third address.

After randomly sampling 1000 triangles (3 nodes each), we found out that 92.4% of the triangles satisfy the triangle inequality.

Question 9:

In this question, we are asked to implement the 1-approximation algorithm for the travelling salesman problem (TSP) for the graph in Question 6 and find an upper bound on the empirical performance of the approximation algorithm given by:

$$\rho = \frac{\text{Approximate TSP cost}}{\text{Optimal TSP cost}}$$

The TSP consists of an undirected weighted graph, with the nodes being cities and edges representing roads or streets. The edge weights represent the distance between cities. The goal is to find the shortest route from a city such that all other cities are visited only once and the traveller returns to the original city. The route is also called a Hamiltonian cycle satisfying the triangle inequality. The algorithm is NP-Hard, hence approximation methods are useful to reduce compute time for obtaining the solution. We use the 1-approximation algorithm for solving the TSP, which is given as follows:

- Find the MST
- Create a multi-graph (each edge is replaced by two directed edges) from the MST
- Find the Euler cycle in the multi-graph to construct the tour sequence. An Euler cycle is a cycle covering every edge only once.
- Search for each edge in the tour sequence in the MST. Use Djkstra's algorithm to find the shortest path between the nodes that contain the edges if the edge does not exist.
- Calculate the approximate TSP cost as the sum of the weights of the edges in the tour sequence (or shortest path cost if edge is absent in the MST). The optimal TSP cost can be approximated by the MST cost as the optimal TSP cost (best possible route) is always greater than MST cost. Thus,

$$\rho = \frac{\text{Approximate TSP cost}}{\text{MST cost}}$$

In theory:

Length of the tour returned by the algorithm $\leq$ Distance covered by Euler cycle $\leq 2\times$Weight of minimum spanning tree (Length of Euler tour) $\leq 2\times$Weight of minimum tour length $\equiv$ MST cost $\leq$ Optimal TSP cost $\leq$ Approximate TSP cost $\leq 2\times$MST cost

Our MST cost was 269084.54500000016, Approximate TSP cost was 421489.3149999998 (which was less than $2\times$MST cost = 538169.09). Normalizing all the costs by dividing everything with MST cost, we get the following inequality:

$$1 \leq \frac{\text{Optimal TSP cost}}{\text{MST cost}} \leq \rho \leq 2$$

Thus, the value of ρ we obtained was 1.5663824728395292. This satisfies the inequality we obtained above:

$$1 \leq \frac{\text{Optimal TSP cost}}{\text{MST cost}} \leq 1.5663824728395292 \leq 2$$

We observe that the upper bound of ρ is 2, and the lower bound of ρ is 1.

Question 10:

In this question, we are asked to plot the trajectory that Santa has to travel. Figure 6 shows the plots of the places Santa needs to visit on the Los Angeles map, downloaded from https://www.openstreetmap.org/, while Figure 7 shows the trajectory.

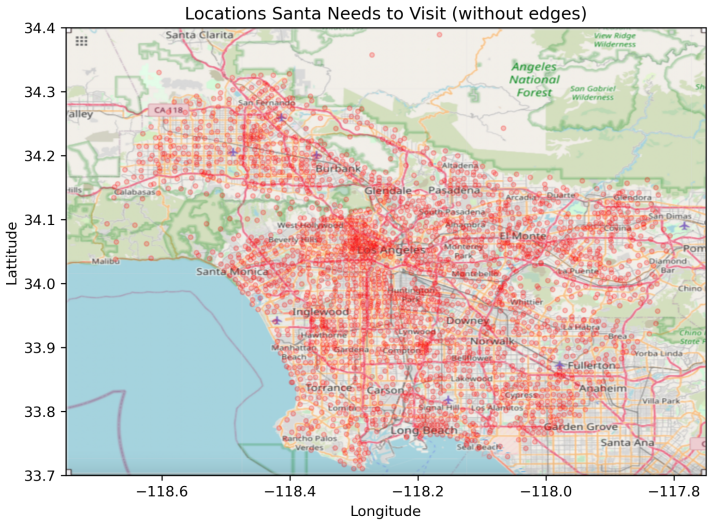


Figure 6: Plot showing all the nodes Santa needs to visit on the Los Angeles map.

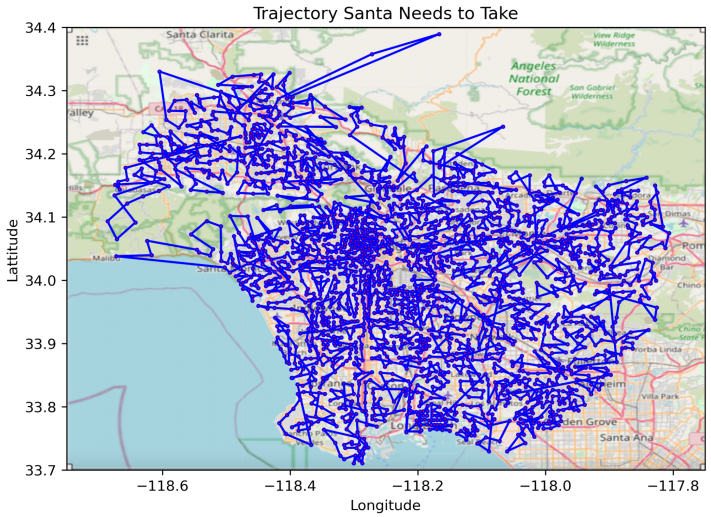


Figure 7: Plot showing the trajectory Santa needs to take on the Los Angeles map.

To see if the results make intuitive sense, we plot the first 10 addresses Santa needs to visit in the order the 1-approximation TSP algorithm suggested using https://maps.co/gis/ . The plot is shown in Figure 8.

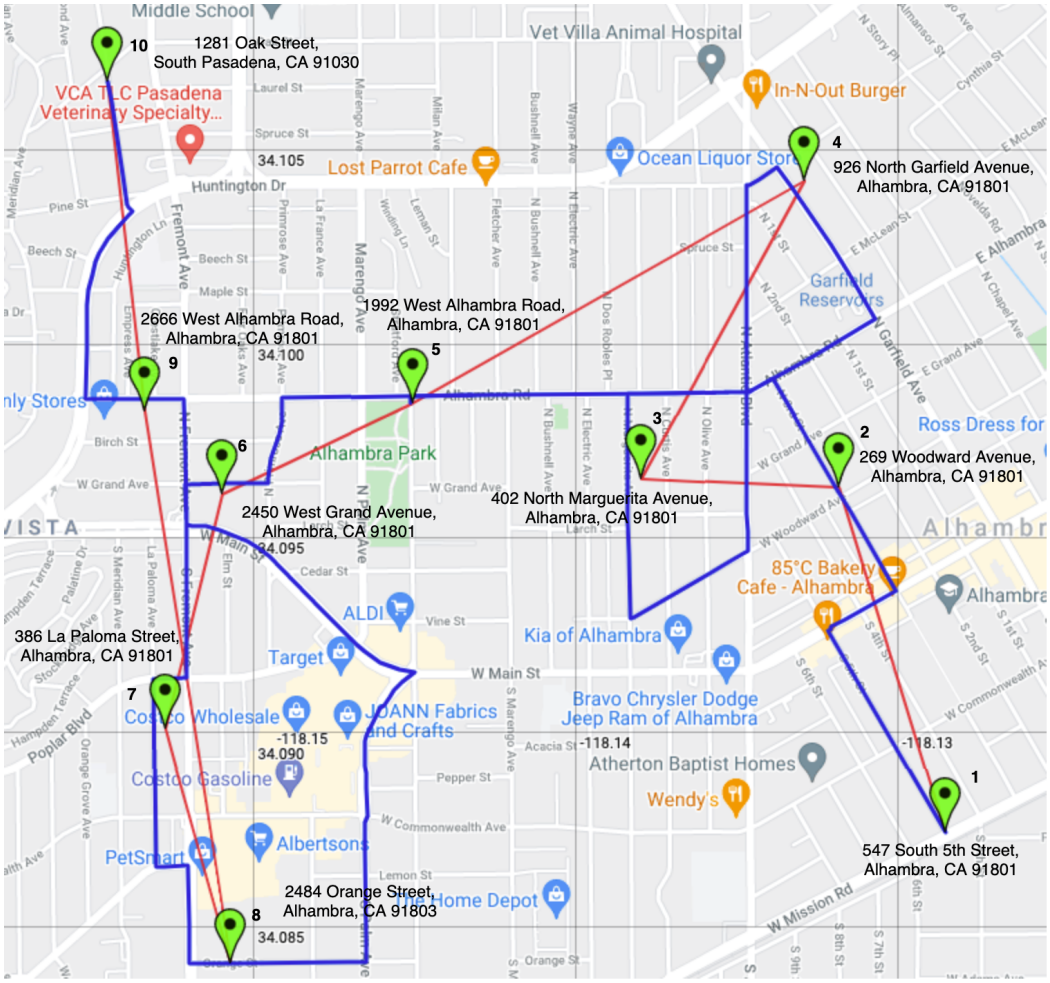


Figure 8: Routes suggested by 1-approximation TSP algorithm (in order). Red line represents the TSP suggested route, blue line represents the actual driving route suggested by Google Maps. The street addresses are marked on the plot.

From Figures 6, 7 and 8, we see that the results make intuitive sense. Most of the edges do not overlap, but there are some overlapping edges as the graph is not fully connected. This causes us to use Djkstra's shortest path algorithm in some of the cases, which does not have the strict notion from the Eulerian circuit that an edge can be traversed only once. This causes a few edges to overlap with each other such that Santa has to traverse back to some other node. In addition, since the TSP does not have actual road information, some of the edges it produced pass over mountains, oceans or even buildings without any actual roads, as observed in Figure 7 and 8. Furthermore, From Figure 8, we observe that the TSP is indeed choosing the closest addresses (or the address that minimizes the overall mean travel time) for each successive traversal, with none of the addresses being visited more than once. We also calculated that the average distance between each successive point for the locations in Figure 8 is ~0.6 miles.

Question 11:

In this question, we are asked to use the Delaunay triangulation (DT) algorithm to estimate the map of roads without using road information. DT is a triangulation algorithm for a given set of discrete points such that no point in the set of points lies within the circumcircle of any triangle. In other words, DT tries to maximize the "minimum angle of all the angles of the triangles in the triangulation", and avoids very narrow triangles (silver triangles). This is useful for discovering more practical and plausible road structures from a set of locations, as road structures in real life follow geometric patterns similar to what DT would produce. The road mesh obtained from DT is shown in Figure 9, while the induced graph G_Δ produced from the triangulation edges are shown in Figure 10.

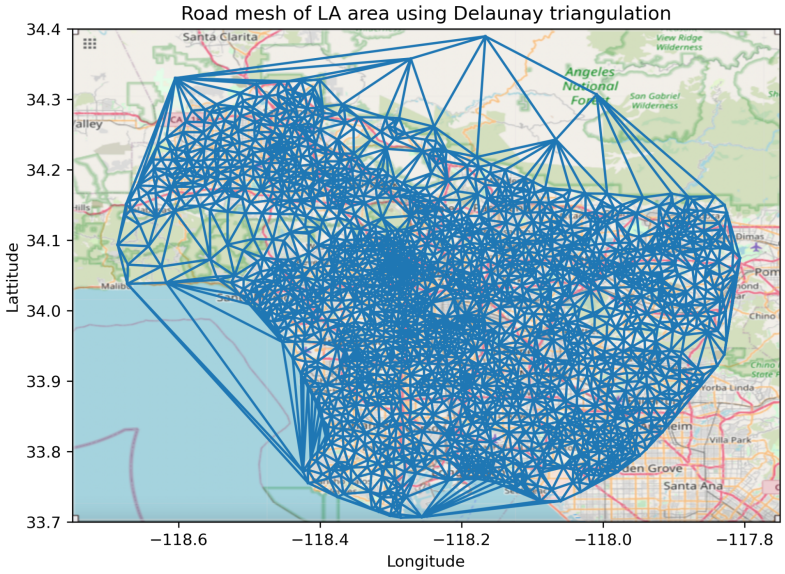


Figure 9: Road mesh of Los Angeles found using Delaunay triangulation.

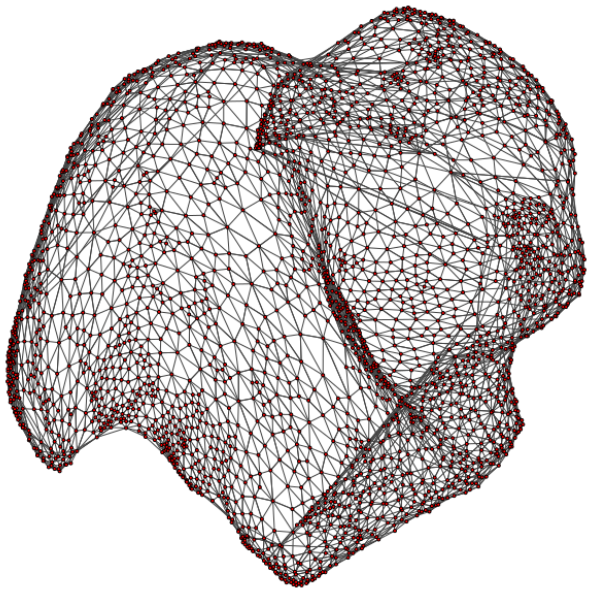


Figure 10: G_{Δ} produced from the triangulation edges.

From Figure 9, we can see that the triangulation algorithm has mostly extracted the road structures of Los Angeles, particularly in the downtown area. However, we also see roads going over oceans and mountains (non existent roads). This happens due to the nature of the DT algorithm, which tries to avoid generating silver triangles or very narrow triangles by ensuring no point in the set of points lies within the circumcircle of any triangle. In other words, DT algorithm tries to fit every triangle inside a circumcircle. This is clear from Figure 10, where the nodes represent the locations and the edges are produced by triangulation. Most of the polygons do not have extremely small acute angles.

Question 12:

The derivation for traffic flow is as follows:

- $\text{Total distance} = \dfrac{\text{Velocity of car} \times \text{Mean Travel time}}{60 \times 60}$ (divide by 3600 to convert data from seconds to hours)
- $\text{Gap} = 0.003 + \dfrac{2 \times \text{Velocity of car}}{60 \times 60}$ (to accommodate for length of each car and the distance between the cars)
- $\text{Total number of cars on the road} = \dfrac{2 \times \text{Total distance}}{\text{Gap}}$ (multiplied by 2 to accommodate flow in both directions)
- $\text{Traffic Flow (cars/hour)} = \dfrac{60 \times 60}{\text{Mean Travel Time}} \times \text{Total number of cars on the road}$

Simplifying the derivation, we get:

$$\boxed{\text{Traffic Flow (cars/hour)} = \frac{3600 \times \text{Velocity of car}}{5.4 + \text{Velocity of car}}}$$

The unit of velocity is in miles per hour. From Pythagoras Theorem, the velocity of car between two coordinates is given as:

$$\text{Velocity of car} = \frac{69}{\text{Mean Travel Time}} \times \sqrt{(\text{Lattitude}_{\text{point 1}} - \text{Lattitude}_{\text{point 2}})^2 + (\text{Longitude}_{\text{point 2}} - \text{Longitude}_{\text{point 2}})^2}$$

Question 13:

In this question, we are asked to calculate the maximum number of cars that can commute per hour from Malibu [34.04, -118.56] to Long Beach [33.77, -118.18] and also calculate the number of edge-disjoint paths between the two spots.

Note that the coordinates for Malibu given in the question is wrong and instead points to Pacific Palisades. Instead of using [34.04, -118.56] as coordinates of Malibu, we use [34.026, -118.78] as the coordinates of Malibu. This change is important because if we do not make this change, then Question 15 does not yield any change.

Using the derivation shown in Question 12 and applying the max-flow algorithm, we calculate the maximum number of cars from Malibu to Long Beach to be 13095 cars/hour.. The number of disjoint paths between the two spots is 6. This makes more sense when we check the roadmap near the two locations, shown in Figure 11.

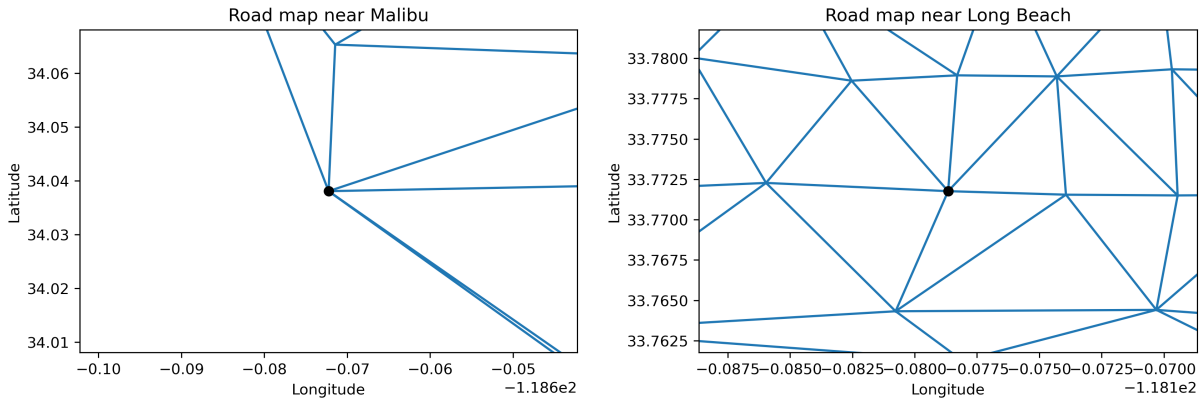


Figure 11: Road maps near Malibu and Long Beach.

From the plots in Figure 11, we see that both Malibu and Long Beach have 6 outgoing and incoming edges respectively and we can also see the degree of each node, which are the same. For two nodes on a graph, the minimum of the number of edges outgoing and incoming to each node provides the number of edge-disjoint paths, which in this case is 6.

Question 14:

In this question, we are asked to prune our graph to remove non-existent and unreal roads, particularly those crossing the oceans and going over mountains. To do so, we apply a threshold on the travel time of the roads via on G_Δ. In theory, the thresholding method removes large edges that the DT algorithm might produce to fit triangles within the circumcircle. The mean travel time would be greater for this edge than other edges, as other edges cross through several nodes of smaller distances. We improve the threshold by adding distance information into account, such that points which are close but have a large travel time are pruned out. This ensures that we do not cut out roads between points that are actually far apart. By incorporating distance information, we essentially put a threshold on the speed, which we set to be 19.2 miles per hour. Figure 12 shows the resulting road mesh map and Figure 13 shows the resulting graph $\tilde{G}_\Delta$.

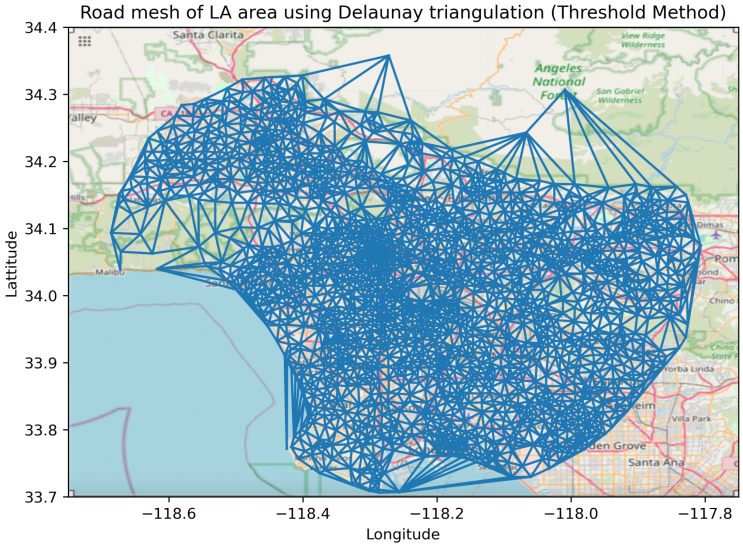


Figure 12: Pruned Road mesh of Los Angeles found using Delaunay triangulation (after thresholding)

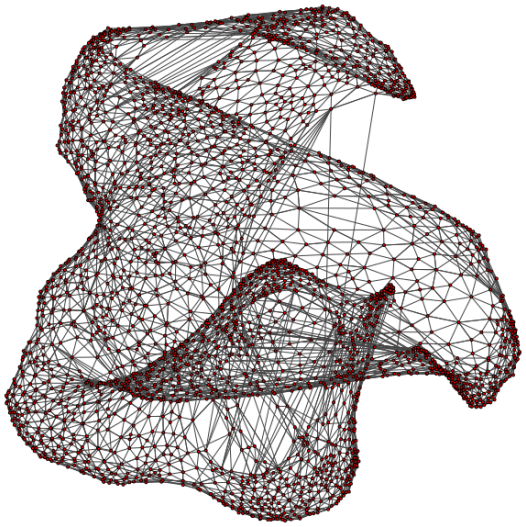


Figure 13: $\tilde{G}_{\Delta}$ produced from the triangulation edges (after thresholding).

From Figure 12, we can see that the long non-existent routes over the ocean, as well as the routes over the Topanga mountain ranges have disappeared. This indicates that the thresholding method did work. The road mesh network now more closely resembles the actual Los Angeles road map, with most of the roads stretching within the coastline and not going through beaches or mountains. In addition, we see fewer long edges among neighboring nodes in Figure 13 compared to Figure 10, indicating that the threshold on speed has indeed removed only those edges which are long but connect neighboring nodes.

Question 15:

After removing the non-existent paths, the number of disjoint paths have now decreased from 6 to 3. This is clearer from the road maps of Malibu and Long Beach shown in Figure 14.

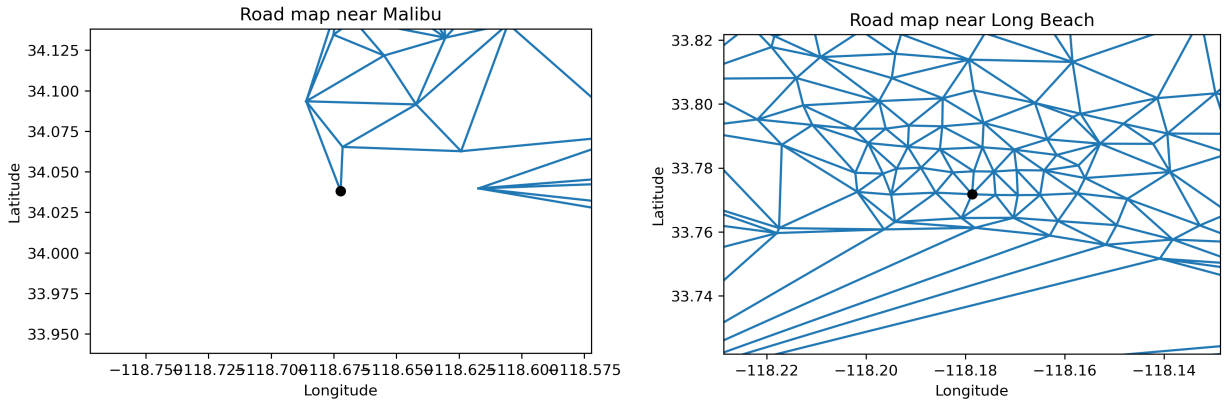


Figure 14: Road maps near Malibu and Long Beach (after thresholding).

From Figure 14, we see that there has not been any significant change to the road map near long long beach. However, several non-existent paths to Malibu have been pruned out. For two nodes on a graph, the minimum of the number of edges outgoing and incoming to each node provides the number of edge-disjoint paths, which in this case is 3. Since some of the paths to Malibu have been cut, we see a reduction in the number of possible edge-disjoint paths between Malibu and Long Beach.

However, the maximum flow is still 13095 cars / hour. This is because there still exist many paths that cars can take to reach Long Beach from Malibu. In other words, the capacity of all the roads leading from Malibu to Long Beach is still larger than 13095 cars. It also has to do with how the max-flow algorithm works. The maximum flow is achieved by summing the flow across the minimum cuts (edges with least weights). It is unlikely that the non-existent roads will have the lowest weights, and hence not contribute to the maximum flow at all. Thus, there will be no significant difference in the maximum flow.

Define Your Own Task:

In this task, we showcase a simple example of the capacitated vehicle routing problem (CVRP). CVRP is concerned with finding the optimal route for a set of vehicles with limited carrying capacity to pick up and deliver goods to a given set of customers. CVRP generalizes the TSP, with the goal of delivering items with least cost, i.e., minimize the sum of travel costs for all vehicles.

We can formulate the problem in terms of graph theory. Consider a graph $G(V, E)$, where V (nodes) represents the location of customers and E (edges) represents the routes. The edges of the graph are weighted, with c_{ij} representing the travel distance (cost) between node i and node j. Node 0 refers to the warehouse where the packages are stored, and the other nodes ($[1, N]$) represent the customer locations. K represents the fleet of vehicles. $d_i \geq 0$ is the number of packages required by customer i and $Q \geq 0$ is the maximum carrying capacity of the vehicles (we assume all vehicles in K have the same capacity). We also define a boolean variable x_{ij}^k that indicates whether a vehicle $k \in K$ will take route e_{ij}:

$$x_{ij}^k = \begin{cases} 1 & \text{if route taken} \\ 0 & \text{otherwise} \end{cases}$$

The constraints and objective functions for CVRP are as follows (vehicle flow formulation):

- Objective function of CVRP to minimize travel distances of the fleet: $\min \sum_{k \in K} \sum_{(i,j) \in E} c_{ij} x_{ij}^k$
- Only one vehicle can visit each customer: $\sum_{k \in K} \sum_{i \in V, i \notin j} x_{ij}^k = 1 \quad \forall j \in V \setminus 0$
- Each vehicle must start from the warehouse: $\sum_{j \in V \setminus \{0\}} x_{0j}^k = 1$
- Number of vehicles entering and exiting a node is equal: $\sum_{i \in V, i \neq j} x_{ij}^k - \sum_{i \in V} x_{ji}^k = 0$
- Each vehicle can carry at most Q packages: $\sum_{i \in V} \sum_{j \in V \setminus \{0\}, i \neq j} q_j x_{ij}^k \leq Q \quad \forall k \in K$
- Vehicle starts and returns to warehouse: $\sum_{k \in K} \sum_{(i,j) \in S, i \neq j} x_{ij}^k \leq |S| - 1 \quad S \subseteq V \setminus \{0\}$
- Boolean variable (0 or 1): $x_{ij}^k \in \{0, 1\} \quad \forall k \in K, \forall (i, j) \in E$

Suppose we have 9 customers, 1 warehouse, 4 vehicles, with each customer requiring between 10 and 20 packages to be delivered and each vehicle having a maximum carrying capacity of 50.

We set the (latitude, longitude) of the warehouse to be (40.748817, -73.985428). The longitudes and latitudes of the customer locations are generated from a Gaussian distribution around the location of the warehouse. The locations, along with the number of packages that need to be delivered to each location are as follows:

	latitude	longitude	package_count
0	40.748817	-73.985428	0
1	40.743057	-73.972162	14
2	40.748359	-73.990814	10
3	40.743823	-73.995250	16
4	40.755161	-73.989855	15
5	40.754181	-73.989340	17
6	40.754599	-73.994061	15
7	40.739551	-73.988505	15
8	40.736550	-73.979024	18
9	40.755834	-73.983573	11

Node 0 is the warehouse, so it has no package count. Figure 15 shows the locations on a map. We use `googlemaps` and `gmaps` libraries in Python to plot the locations.

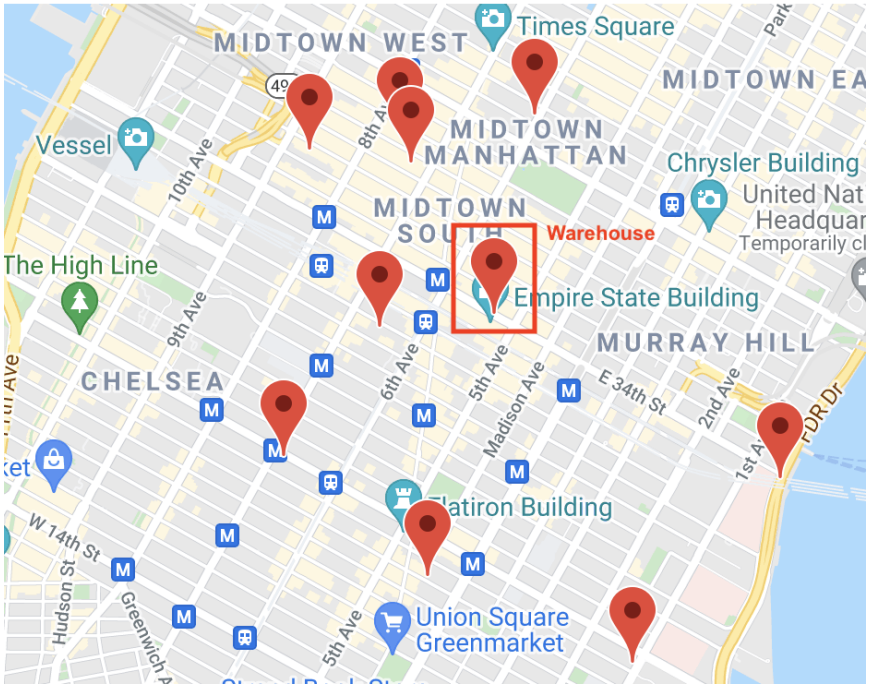


Figure 15: Locations of customers relative to the warehouse on Google Maps.

To solve the CVRP in terms of the objective function and the constraints, we use the `PULP` library in Python. After running the optimization problem, we see that the most optimal route requires only 3 vehicles, covering a distance of 8.7 miles. The route is shown in Figure 16, with the path taken by each vehicle color coded.

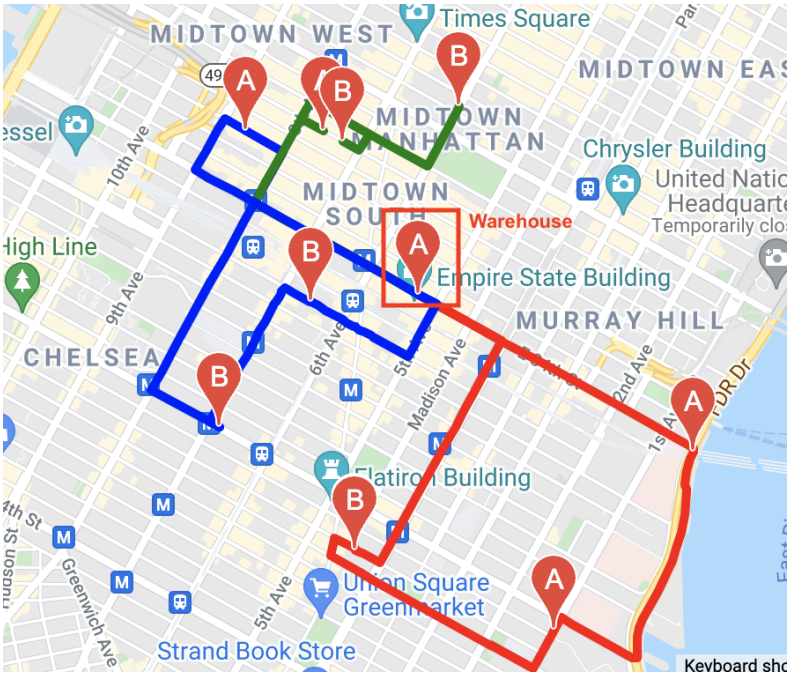


Figure 16: Most optimal route for the CVRP, using 3 out of 4 vehicles.

The induced graph is shown in Figure 17, with the warehouse colored green and customer locations colored yellow. The number of packages at each node is also shown in the plot.

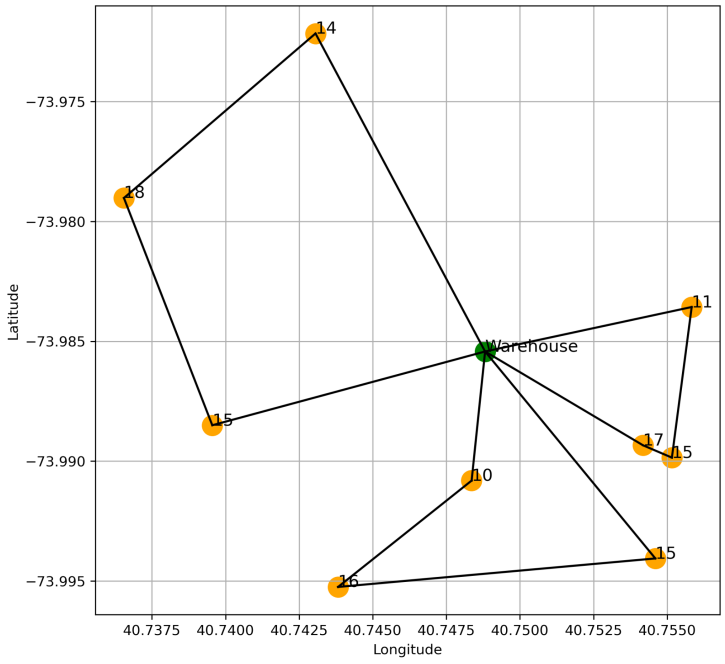


Figure 17: Induced graph showing customer locations and warehouse.

The main challenge in the CVRP is covering all corner cases and modelling the constraints properly. As we saw in the project, even the simplest classic CVRP formulation requires a lot of constraints to be modelled. In addition, CVRP is a NP-hard problem, so the scale of solution that can be found via combinatorial optimization or mathematical modeling is limited. In the real world, industries tend to use metaheuristics such as Genetic algorithms, Tabu search, Simulated annealing and Adaptive Large Neighborhood Search (ALNS) are normally used. There are several variants of VRP apart from CVRP, including VRP with profits, VRP with pickup and delivery, VRP with LIFO, VRP with time windows and VRP with multiple trips.

Saha_Young_Proj4_1-5

June 7, 2021

[1]:
```
library("igraph")
```

```
Attaching package: ‘igraph’


The following objects are masked from ‘package:stats’:

    decompose, spectrum


The following object is masked from ‘package:base’:

    union
```

0.1 Question 2

[2]:
```
tickers_sectors <- read.csv(file = 'finance_data/Name_sector.
 ↪csv',header=TRUE,stringsAsFactors=FALSE)
filenames = paste("finance_data/data", list.files("finance_data/data",␣
 ↪pattern="*.csv"), sep="/")

length_data<-c()
i<-1
log_norm_mat = matrix(0,length(filenames)-11,764) #omit files with NaN data
for(j in c(1:length(filenames))){
    df = read.csv(filenames[j],header=TRUE, stringsAsFactors=FALSE)
    length_data[j] = dim(df)[1]
    if(length_data[j]==765){
        p = df[,5]
        q = c()
        r = c()
        for(k in c(2:length(p))){
            q[k-1] = (p[k]-p[k-1])/p[k-1]
        }
```

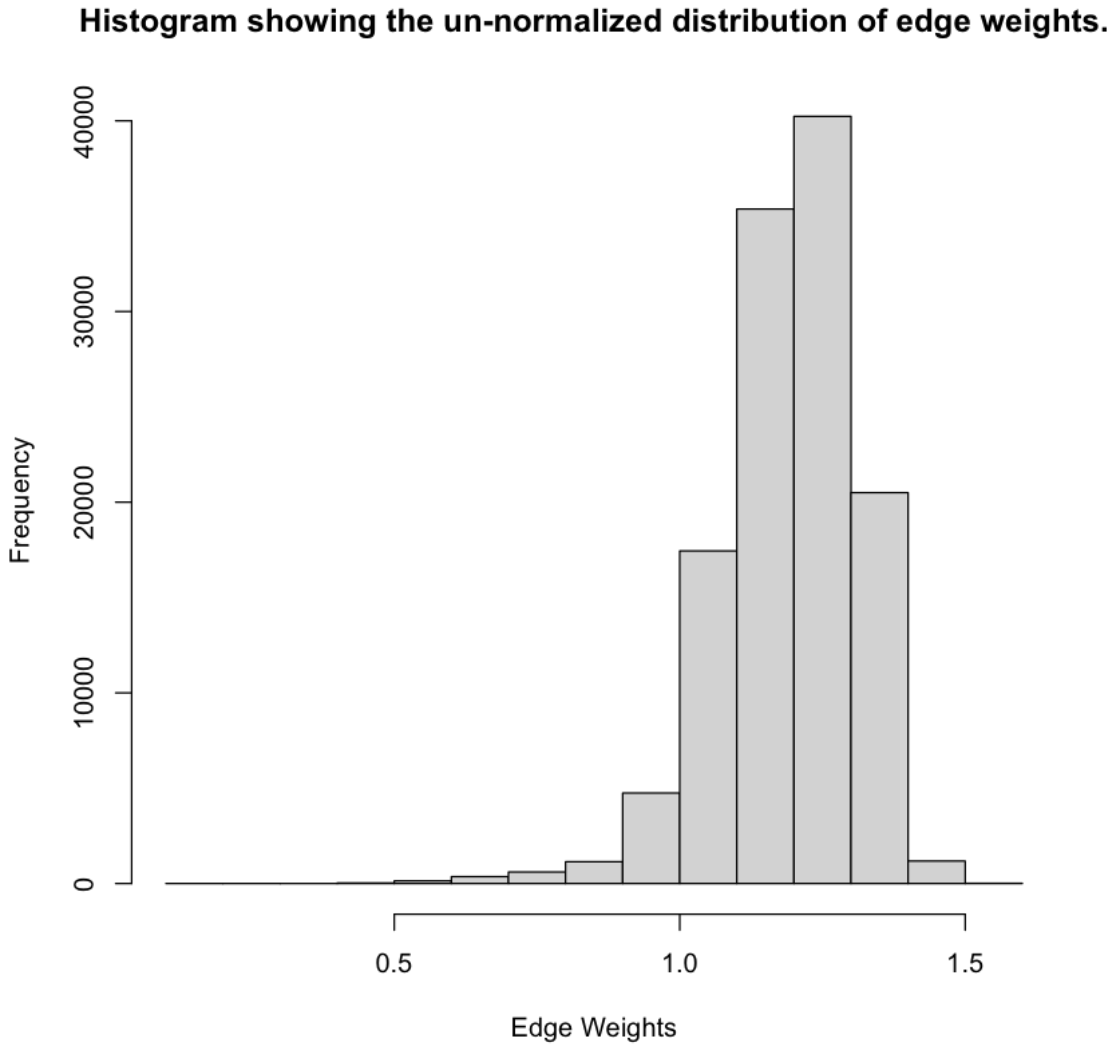
```
        r = log(1+q)
        log_norm_mat[i,] = r
        i = i+1
    }
}
```

[3]:
```
get_edges<- function(edge_weight_file,log_norm_mat,tickers_sectors){
  cat("Source","\t","Sink","\t","Weight",file=edge_weight_file)
  for(i in c(1:(dim(log_norm_mat)[1]-1))){
    for(j in c((i+1):dim(log_norm_mat)[1])){
      ri <- mean(log_norm_mat[i,])
      rj <- mean(log_norm_mat[j,])
      ri2 <- log_norm_mat[i,]^2
      rj2 <- log_norm_mat[j,]^2
      rhoij <- ((mean(log_norm_mat[i,]*log_norm_mat[j,]))-(ri*rj))/
↪(sqrt((mean(ri2)-(ri^2))*(mean(rj2)-(rj^2))))
      wij <- sqrt(2*(1-rhoij))
    ␣
↪cat('\n',tickers_sectors[i,1],'\t',tickers_sectors[j,1],'\t',wij,file=edge_weight_file)
    }
  }
}
```

[4]:
```
tickers_sectors=tickers_sectors[-which(length_data!=765),]
edge_weight_file <- file("finance_data/edge_weights.txt", "w")
get_edges(edge_weight_file,log_norm_mat,tickers_sectors)
close(edge_weight_file)

edge_list= read.delim("finance_data/edge_weights.txt",header=TRUE)
correlation_graph = graph.data.frame(edge_list, directed = FALSE)
E(correlation_graph)$weight = edge_list[,"Weight"]
```

[5]:
```
hist(edge_list[,"Weight"],main="Histogram showing the un-normalized␣
↪distribution of edge weights.",xlab="Edge Weights",ylab="Frequency")
dev.copy2eps(file='Q2.eps')
```

pdf: 2

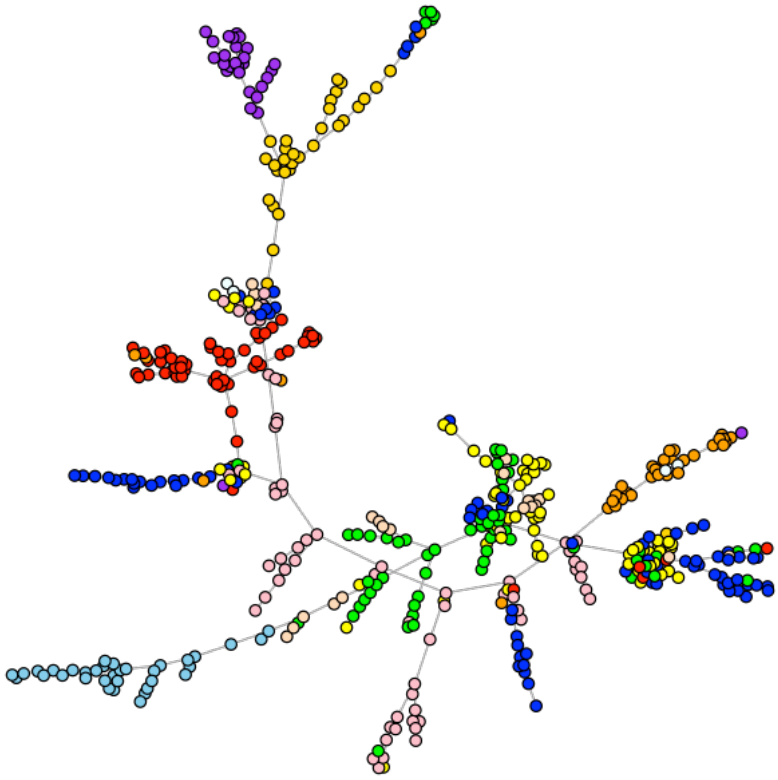


0.2 Question 3

[6]:
```
mst <- mst(correlation_graph,algorithm="prim")
sectors = unique(tickers_sectors[,2])

colors <- c()
  for(v in c(1:vcount(correlation_graph))){
    cur_sector <- tickers_sectors[v,2]
    i <- which(sectors==cur_sector)
    colors[v] <-␣
 ↪switch(i,"red","green","blue","yellow","orange","purple","pink","gold","peachpuff","skyblue
}
```

```
plot(mst,vertex.size=3, vertex.label=NA, vertex.color=colors)
dev.copy2eps(file='Q3.eps')
```

pdf: 2



0.3 Question 4

[7]:
```
Si <- c()
for(i in c(1:length(sectors))){
  Si[i] <- length(which(tickers_sectors[,2]==sectors[i]))
}
p1 <- c()
```

```
p2 <- c()
for(v in c(1:vcount(mst))){
  neighbors <- neighbors(mst,v)
  Ni <- length(neighbors)
  Qi<-0
  for(i in neighbors){
    if(tickers_sectors[i,2]==tickers_sectors[v,2])
      Qi<-Qi+1
  }
  p1[v] <- Qi/Ni
  p2[v] <- Si[which(sectors==tickers_sectors[v,2])]/vcount(mst)
}
alpha1 <- sum(p1)/vcount(mst)
alpha2 <- sum(p2)/vcount(mst)
print(sprintf("Values of alpha for the two cases: %f and %f",alpha1,alpha2))
```

```
[1] "Values of alpha for the two cases: 0.828930 and 0.114188"
```

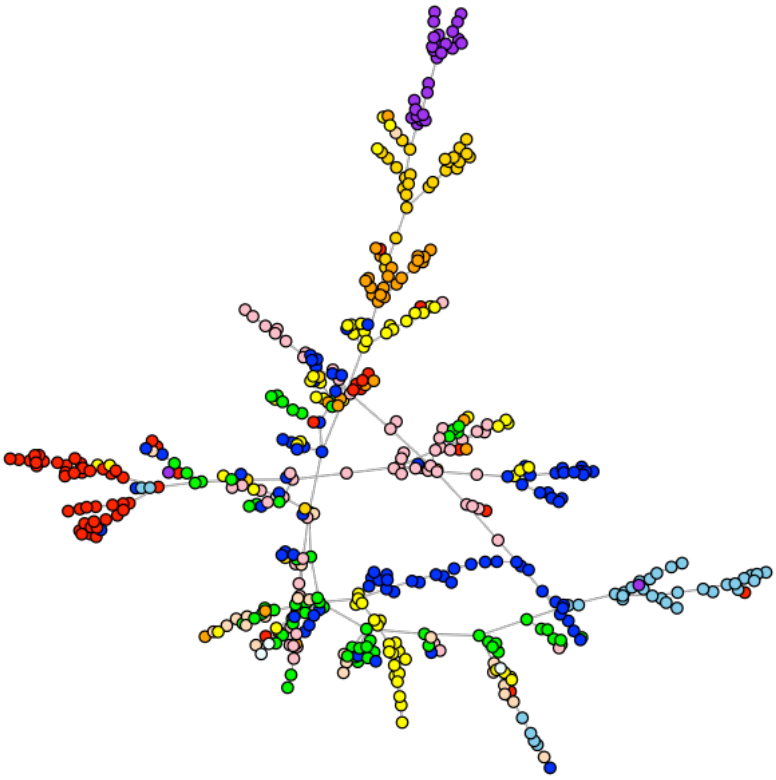
0.4 Question 5

[8]:
```
tickers_sectors_week <- read.csv(file = 'finance_data/Name_sector.
↪csv',header=TRUE,stringsAsFactors=FALSE)
filenames_week = paste("finance_data/data", list.files("finance_data/data",␣
↪pattern="*.csv"), sep="/")

length_data_week<-c()
i<-1
log_norm_mat_week = matrix(0,length(filenames_week)-13,142) #omit files with␣
↪NaN data
for(j in c(1:length(filenames_week))){
    df = read.csv(filenames_week[j],header=TRUE, stringsAsFactors=FALSE)
    df["Day"]=weekdays(as.Date(df[,1]))
    df =subset(df, Day=='Monday')
    length_data_week[j] = dim(df)[1]
    if(length_data_week[j]==143){
        p = df[,5]
        q = c()
        r = c()
        for(k in c(2:length(p))){
            q[k-1] = (p[k]-p[k-1])/p[k-1]
        }
        r = log(1+q)
        log_norm_mat_week[i,] = r
        i = i+1
    }
}
```

[9]:
```
tickers_sectors_week=tickers_sectors_week[-which(length_data_week!=143),]
edge_weight_file_week <- file("finance_data/edge_weights_week.txt", "w")
get_edges(edge_weight_file_week,log_norm_mat_week,tickers_sectors_week)
close(edge_weight_file_week)

edge_list_week= read.delim("finance_data/edge_weights_week.txt",header=TRUE)
correlation_graph_week = graph.data.frame(edge_list_week, directed = FALSE)
E(correlation_graph_week)$weight = edge_list_week[,"Weight"]
```

[14]:
```
mst_week <- mst(correlation_graph_week,algorithm="prim")
sectors_week = unique(tickers_sectors_week[,2])

colors_week <- c()
  for(v in c(1:vcount(correlation_graph_week))){
    cur_sector <- tickers_sectors_week[v,2]
    i <- which(sectors_week==cur_sector)
    colors_week[v] <-␣
 ↪switch(i,"red","green","blue","yellow","orange","purple","pink","gold","peachpuff","skyblue
}

plot(mst_week,vertex.size=3, vertex.label=NA, vertex.color=colors_week)
dev.copy2eps(file='Q5.eps')
```

pdf: 2



[ ]:

Saha_Young_Proj4_6-15

June 7, 2021

[2]:
```
import numpy as np
import pandas as pd
import igraph as ig
import json
import csv
import matplotlib.pyplot as plt
import cairocffi
import cairo
import networkx as nx
from scipy.spatial import Delaunay
```

0.1 Question 6

[2]:
```
df = pd.read_csv('los_angeles-censustracts-2019-4-All-MonthlyAggregate.csv',␣
 ↪usecols=['sourceid', 'dstid', 'mean_travel_time', 'month'])
df = df[df['month']==12][['sourceid','dstid','mean_travel_time']]
gd = df.values
graph_dict = {}
for sourceid,dstid,mean_travel_time in gd:
    key = tuple(np.sort([int(sourceid), int(dstid)]))
    if key in graph_dict: graph_dict[key].append(mean_travel_time)
    else: graph_dict[key] = [mean_travel_time]

with open('graph_data.txt','w') as f:
    for loc in graph_dict:
        f.write('{} {} {}\n'.format(loc[0], loc[1], np.mean(graph_dict[loc])))

g = ig.Graph.Read(f = 'graph_data.txt', format = 'ncol', directed = False)
gcc = g.components().giant()
print("Number of nodes and edges: ",len(gcc.vs),len(gcc.es))
```

```
Number of nodes and edges:  2649 1004955
```

0.2 Question 7

[3]:
```
mst = gcc.spanning_tree(weights = gcc.es["weight"])
visual_style = {}
visual_style["vertex_size"] = 3
ig.plot(mst,**visual_style)
```

[4]:
```
out = ig.plot(mst,**visual_style)
out.save('Q7.png')
```

[4]:
```
location_data = {}
with open('los_angeles_censustracts.json', 'r') as f:
    cur_data = json.loads(f.readline())
    for feature in cur_data['features']:
        coordinates = np.array(feature['geometry']['coordinates'][0][0])
        location_data[feature['properties']['MOVEMENT_ID']] = {'address':␣
 ↪feature['properties']['DISPLAY_NAME'],
                                                               'mean_coords': np.
 ↪mean(coordinates.reshape(-1,2), axis=0)}
```

[5]:
```
mst_edges = mst.es()
print("Coordinates near the two endpoints (the centroid locations) of a few␣
 ↪edges:")
for e in mst_edges[:10]:
    x, y = mst.vs(e.tuple[0])[0]['name'], mst.vs(e.tuple[1])[0]['name']
    print(location_data[str(x)]['address'],location_data[str(x)]['mean_coords'],
          location_data[str(y)]['address'],location_data[str(y)]['mean_coords'])
```

```
Coordinates near the two endpoints (the centroid locations) of a few edges:
Census Tract 554001 [-118.133298   33.904119] Census Tract 554002 [-118.141448
33.896526]
Census Tract 461700 [-118.159325   34.153604] Census Tract 460800 [-118.172425
34.180286]
Census Tract 302201 [-118.251349   34.146334] Census Tract 302202 [-118.248811
34.142665]
Census Tract 407101 [-117.967696   34.04101 ] Census Tract 407002 [-117.980572
34.051745]
Census Tract 433401 [-118.043316   34.058606] Census Tract 433402 [-118.038157
34.056957]
Census Tract 543603 [-118.28871    33.803659] Census Tract 294410 [-118.289861
33.797764]
Census Tract 482001 [-118.165951   34.062293] Census Tract 530700 [-118.176633
34.062274]
Census Tract 460800 [-118.172425   34.180286] Census Tract 463800 [-118.167761
34.141469]
Census Tract 269100 [-118.398461   34.057095] Census Tract 217001 [-118.385502
34.04886 ]
Census Tract 011000 [-117.950433   33.881015] Census Tract 001901 [-117.965225
```

```
33.868368]
```

0.3 Question 8

[7]:
```
triangles = []
while len(triangles)<1000:
    points = np.random.randint(1,high=len(gcc.vs),size=3)
    try:
        e1, e2, e3 = gcc.get_eid(points[0],points[1]), gcc.
 ↪get_eid(points[1],points[2]), gcc.get_eid(points[2],points[0])
        weights = [gcc.es['weight'][e1],gcc.es['weight'][e2],gcc.
 ↪es['weight'][e3]]
        triangles.append(weights)
    except: continue

counter = 0
for i in triangles:
    w1, w2, w3 = i[0], i[1], i[2]
    if w1+w2>w3 and w1+w3>w2 and w3+w2>w1:
        counter+=1
print(counter/len(triangles))
```

```
0.924
```

0.4 Question 9 and 10

[6]:
```
arr = gd
for i in range(0,len(arr)):
    if(arr[i][0]>arr[i][1]):
        t = arr[i][0]
        arr[i][0] = arr[i][1]
        arr[i][1] = t
newdf = pd.DataFrame(arr)
arr1 = newdf.groupby([0,1]).mean().reset_index()
arr1 = arr1.rename(columns={0: "source", 1: "sink", 2: "weight"})

g = nx.from_pandas_edgelist(arr1, 'source','sink', ['weight'])
gcc = g.subgraph(max(nx.connected_components(g), key=len))
mst = nx.minimum_spanning_tree(gcc)
mg = nx.MultiGraph()
mst_cost = 0
for i in mst.edges:
    w = mst.edges[i[0],i[1]]['weight']
    mst_cost += w
    mg.add_edge(i[0],i[1],weight=w)
    mg.add_edge(i[0],i[1],weight=w)
```

```
vertices, count = [], 0
for i in mg.nodes:
    vertices.append(i)
    count += 1
    if count>60:
        break

costs, cur_paths = [], []
for vertex in vertices:
    tour = [u for u,v in nx.eulerian_circuit(mg,source=vertex)]
    cur_path, visited_nodes = [], set()
    for i in tour:
        if i not in visited_nodes:
            cur_path.append(i)
            visited_nodes.add(i)
    cur_path.append(cur_path[0])
    cur_paths.append(cur_path)

    approx_cost = 0
    for i in range(len(cur_path)-1):
        s,t = cur_path[i], cur_path[i+1]
        w = 0
        if mst.has_edge(s,t):
            w = mst.edges[s,t]['weight']
        else:
            w = nx.dijkstra_path_length(gcc,s,t)
        approx_cost += w
    costs.append(approx_cost)

min_approx_cost = min(costs)
trajectory = cur_paths[np.argmin(costs)]
```

[7]:
```
print('MST cost:',mst_cost)
print('Approx. cost:',min_approx_cost)
print('Upper bound:',min_approx_cost/mst_cost)
```

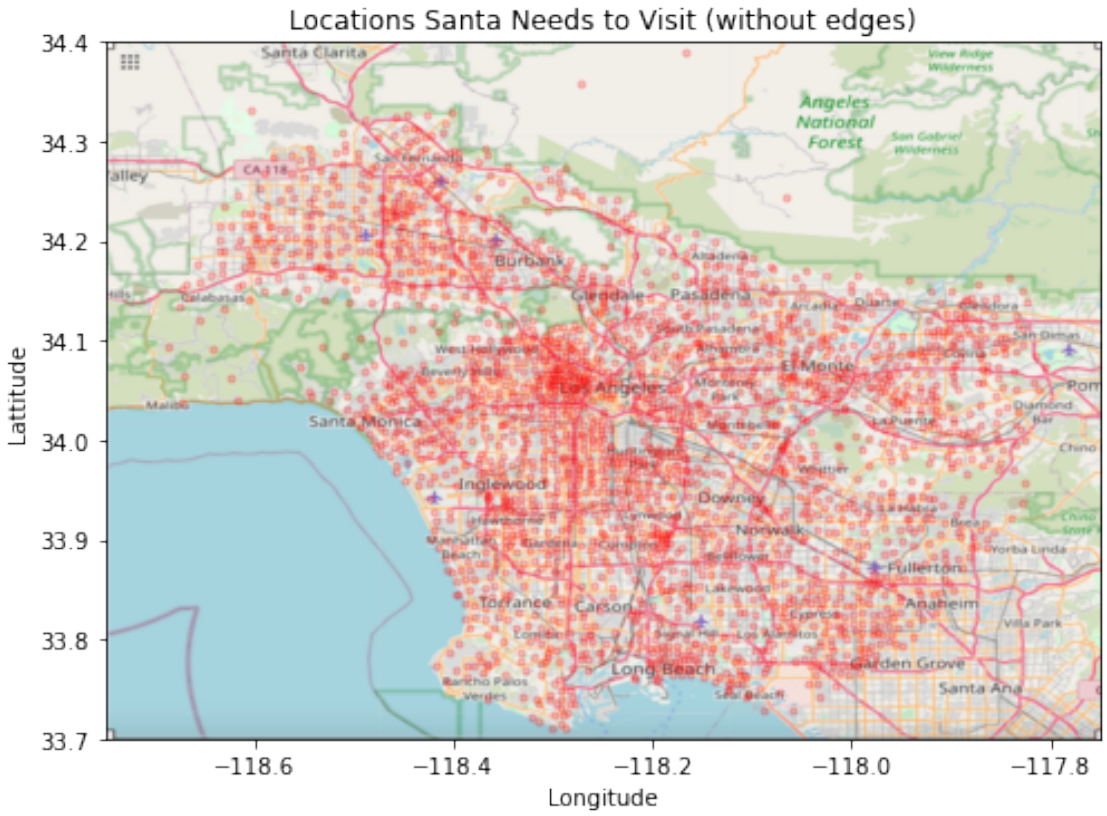
```
MST cost: 269084.54500000016
Approx. cost: 421489.3149999998
Upper bound: 1.5663824728395292
```

[8]:
```
data = json.load(open('los_angeles_censustracts.json'))
location_data = []
for i in trajectory:
    for j in range(len(data['features'])):
        if data['features'][j]['properties']['MOVEMENT_ID']==str(int(i)):
            cur_loc = data['features'][j]['geometry']['coordinates'][0]
```

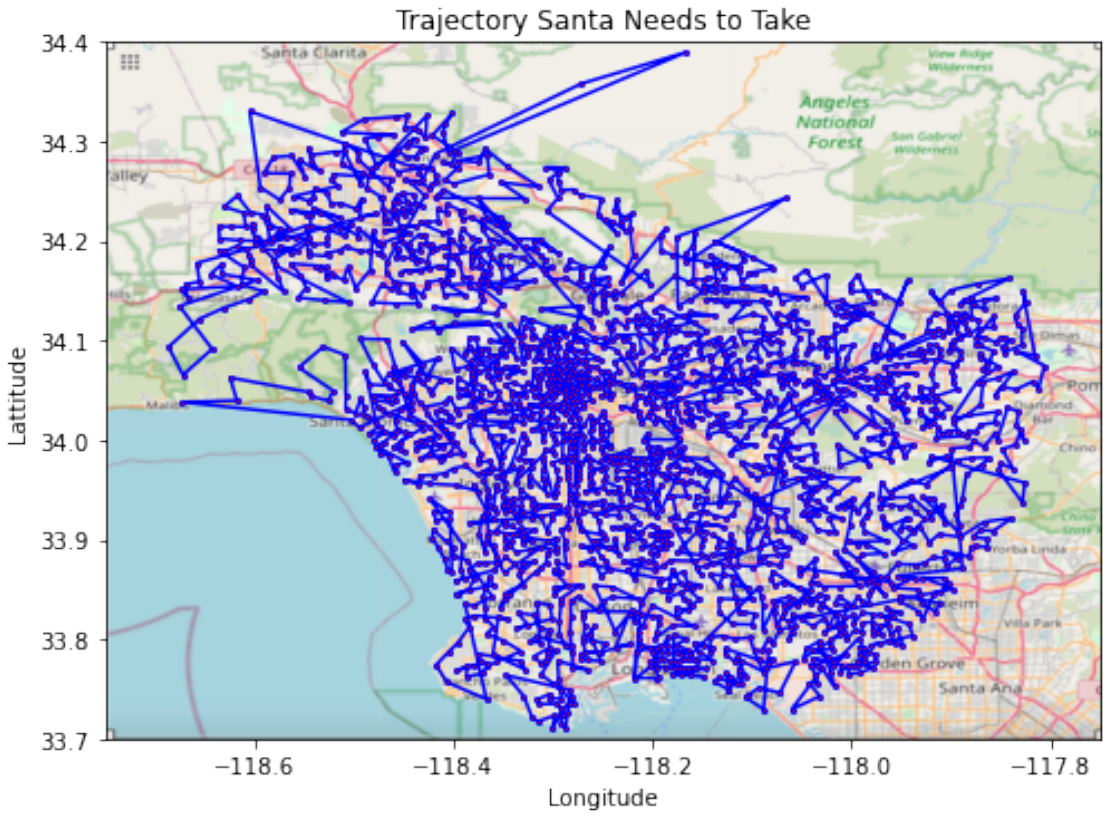
```
            if len(cur_loc)==1:
                t = np.asarray(cur_loc[0]).mean(axis=0)
                location_data.append(t)
            elif len(cur_loc)==2:
                t = np.asarray(cur_loc[0]+cur_loc[1]).mean(axis=0)
                location_data.append(t)
            elif i==1932.0:
                t = np.
↪asarray(cur_loc[0]+cur_loc[1]+cur_loc[2]+cur_loc[3]+cur_loc[4]+cur_loc[5]).
↪mean(axis=0)
                location_data.append(t)
            else:
                t = np.asarray(cur_loc).mean(axis=0)
                location_data.append(t)
x,y = [i[0] for i in location_data], [i[1] for i in location_data]
```

[9]:
```
BBox = ((-118.75,   -117.75,
         33.7, 34.4)) #use these to export map at https://www.openstreetmap.org/
```

[10]:
```
ruh_m = plt.imread('map_LA.png')
fig, ax = plt.subplots(figsize = (8,7))
ax.scatter(x, y, zorder=1, alpha= 0.2, c='r', s=10)
ax.set_title('Locations Santa Needs to Visit (without edges)')
plt.xlabel('Longitude')
plt.ylabel('Lattitude')
ax.set_xlim(BBox[0],BBox[1])
ax.set_ylim(BBox[2],BBox[3])
ax.imshow(ruh_m, zorder=0, extent = BBox, aspect= 'equal')
plt.savefig('Q10a.png',dpi=300,bbox_inches='tight')
plt.show()
```



[11]:
```
ruh_m = plt.imread('map_LA.png')
fig, ax = plt.subplots(figsize = (8,7))
ax.plot(x,y,color='blue',marker='o',markersize=2,markerfacecolor='red')
ax.set_title('Trajectory Santa Needs to Take')
plt.xlabel('Longitude')
plt.ylabel('Lattitude')
ax.set_xlim(BBox[0],BBox[1])
ax.set_ylim(BBox[2],BBox[3])
ax.imshow(ruh_m, zorder=0, extent = BBox, aspect= 'equal')
plt.savefig('Q10b.png',dpi=300,bbox_inches='tight')
plt.show()
```



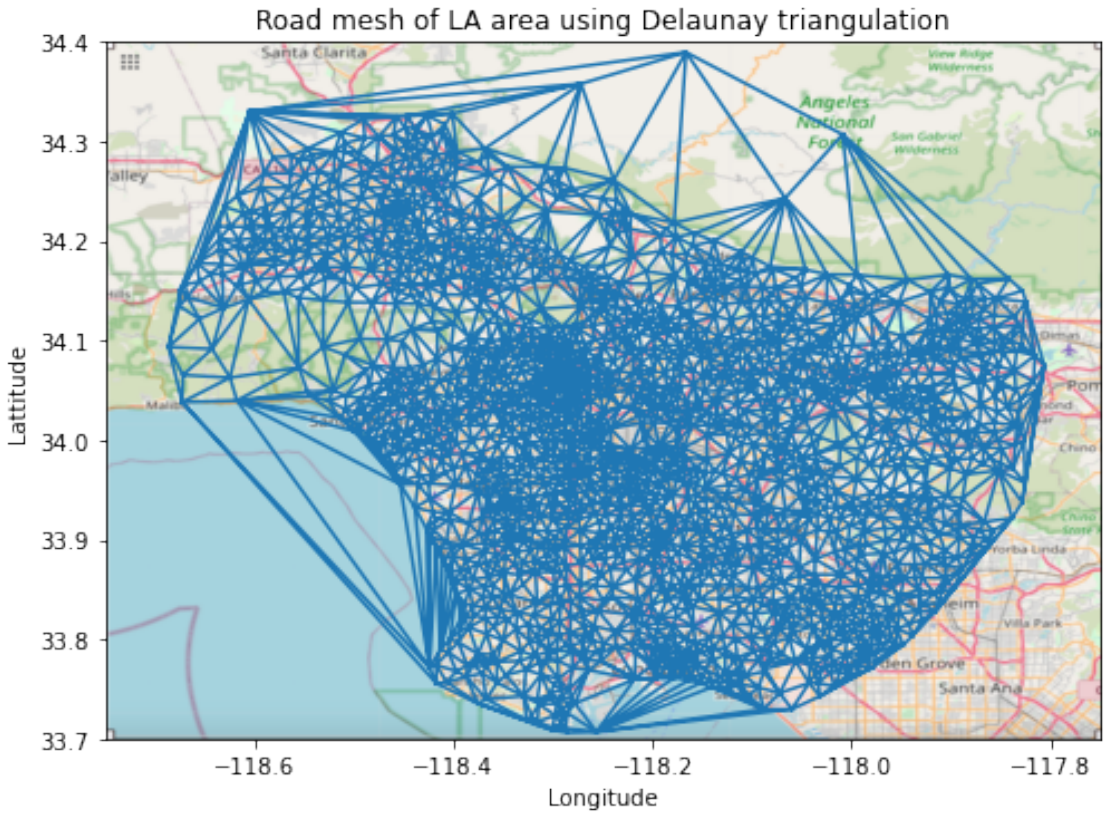
[19]:
```
for i in range(10):
    print('(',x[i],',',y[i],')')
```

```
( -118.12911933333332 , 34.08759475 )
( -118.13138209090911 , 34.09626386363636 )
( -118.13785063157897 , 34.09645121052631 )
( -118.13224544444446 , 34.10349303174603 )
( -118.14492316666666 , 34.098681500000005 )
( -118.15023891071432 , 34.09595766071429 )
( -118.15266638571427 , 34.09029572857144 )
( -118.15075123999998 , 34.083419626666675 )
( -118.15280849999998 , 34.098628 )
( -118.15508200990094 , 34.10732695049504 )
```

0.5 Question 11

[3]:
```
lat_long = {}

with open('los_angeles_censustracts.json', 'r') as f:
    cur_data = json.loads(f.readline())
    features = cur_data['features']
```

```
    for feature in features:
        latitude = 0.0
        longitude = 0.0
        if feature['geometry']['type']=='Polygon':
            coordinates = np.array(feature['geometry']['coordinates'][0])
            for coordinate in coordinates:
                latitude += coordinate[1]
                longitude += coordinate[0]
        if feature['geometry']['type']=='MultiPolygon':
            coordinates = np.array(feature['geometry']['coordinates'][0][0])
            for coordinate in coordinates:
                latitude += coordinate[1]
                longitude += coordinate[0]
        latitude /= len(coordinates)
        longitude /= len(coordinates)

        lat_long[feature['properties']['MOVEMENT_ID']] =␣
↪(feature['properties']['DISPLAY_NAME'], latitude, longitude)

f.close()

lat=[]
lon=[]
for i in range(1,len(lat_long)+1):
    lat.append(lat_long[str(i)][1])
    lon.append(lat_long[str(i)][2])
lat_lon = tuple(zip(lat, lon))
delaunay_out = Delaunay(lat_lon)

BBox = ((-118.75,   -117.75,
         33.7, 34.4))
ruh_m = plt.imread('map_LA.png')

fig, ax = plt.subplots(figsize = (8,7))
plt.triplot(lon, lat, delaunay_out.simplices)
ax.set_title('Road mesh of LA area using Delaunay triangulation')
plt.xlabel('Longitude')
plt.ylabel('Lattitude')
ax.set_xlim(BBox[0],BBox[1])
ax.set_ylim(BBox[2],BBox[3])
ax.imshow(ruh_m, zorder=0, extent = BBox, aspect= 'equal')
plt.savefig('Q11a.png',dpi=300,bbox_inches='tight')
plt.show()
```



[4]:
```
g_del=ig.Graph()
g_del.add_vertices(len(delaunay_out.points))
remove_duplicates=set()
for i in range(len(delaunay_out.simplices)):
    a=((delaunay_out.simplices[i][0], delaunay_out.simplices[i][1]))
    b=((delaunay_out.simplices[i][0], delaunay_out.simplices[i][2]))
    c=((delaunay_out.simplices[i][1], delaunay_out.simplices[i][2]))
    list(a).sort()
    list(b).sort()
    list(c).sort()
    if not a in remove_duplicates:
        remove_duplicates.add(a)
        g_del.add_edges([a])
    if not b in remove_duplicates:
        remove_duplicates.add(b)
        g_del.add_edges([b])
    if not c in remove_duplicates:
        remove_duplicates.add(c)
        g_del.add_edges([c])
```

[5]:
```
visual_style = {}
visual_style["vertex_size"] = 3
ig.plot(g_del,**visual_style)
```

[7]:
```
out = ig.plot(g_del,**visual_style)
out.save('Q11b.png')
```

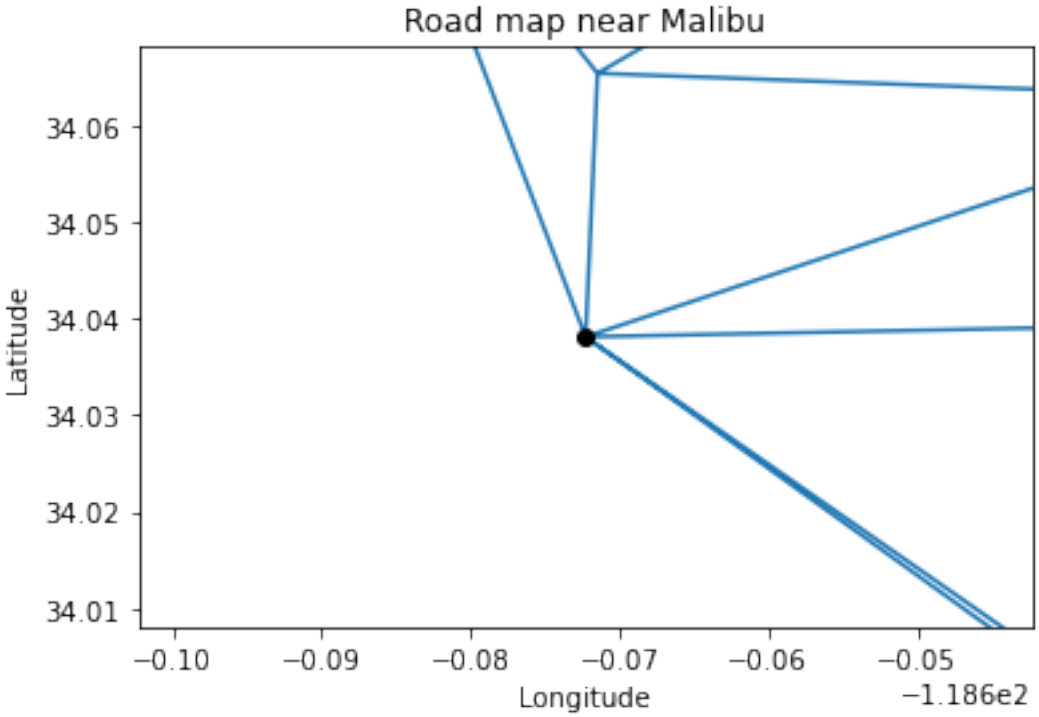
0.6 Question 13

[28]:
```
malibu= [34.026, -118.78]
long_beach =  [33.77, -118.18]
vcar=(69*np.sqrt((malibu[0]-long_beach[0])**2+(malibu[1]-long_beach[1])**2))/1.
 ↪05
max_car_num=(3600*vcar)/(5.4 + vcar)

min_long_beach=np.inf
min_malibu=np.inf
long_beach_node=0
malibu_node=0
for i in range(1,len(lat_lon)):
    long_beach_closest=np.
 ↪sqrt(((lat_lon[i][0])-long_beach[0])**2+((lat_lon[i][1])-long_beach[1])**2)
    malibu_closest=np.
 ↪sqrt((malibu[0]-lat_lon[i][0])**2+(malibu[1]-lat_lon[i][1])**2)
    if long_beach_closest<min_long_beach:
        min_long_beach=long_beach_closest
        long_beach_node=i
    if malibu_closest<min_malibu:
        min_malibu=malibu_closest
        malibu_node=i

print('Number of edge-disjoint paths: ',g_del.
 ↪adhesion(long_beach_node,malibu_node)-1)
print('Degree Distribution of nodes (Malibu, Long Beach): ', g_del.
 ↪degree(malibu_node,mode='out')-1,g_del.degree(long_beach_node,mode='in')-1)
```

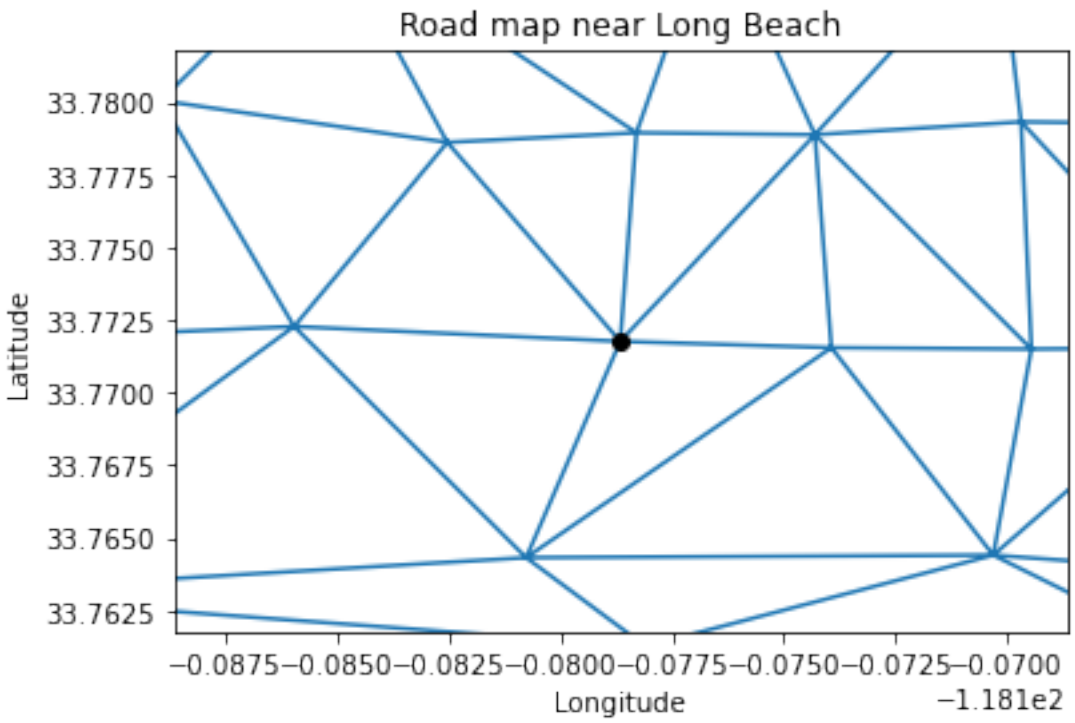
```
Number of edge-disjoint paths:  6
Degree Distribution of nodes (Malibu, Long Beach):  6 8
```

[23]:
```
plt.triplot(lon, lat, delaunay_out.simplices)
plt.ylim(lat_lon[malibu_node][0]-0.03,lat_lon[malibu_node][0]+0.03)
plt.xlim(lat_lon[malibu_node][1]-0.03,lat_lon[malibu_node][1]+0.03)
plt.xlabel('Longitude')
plt.ylabel('Latitude')
plt.title('Road map near Malibu')
plt.plot(lon[malibu_node], lat[malibu_node],'o',color='black')
```

```
plt.savefig('Q13a.png',dpi=300,bbox_inches='tight')
plt.show()
```



[37]:
```
plt.triplot(lon, lat, delaunay_out.simplices)
plt.ylim(lat_lon[long_beach_node][0]-0.01,lat_lon[long_beach_node][0]+0.01)
plt.xlim(lat_lon[long_beach_node][1]-0.01,lat_lon[long_beach_node][1]+0.01)
plt.xlabel('Longitude')
plt.ylabel('Latitude')
plt.title('Road map near Long Beach')
plt.plot(lon[long_beach_node], lat[long_beach_node],'o',color='black')
plt.savefig('Q13b.png',dpi=300,bbox_inches='tight')
plt.show()
```



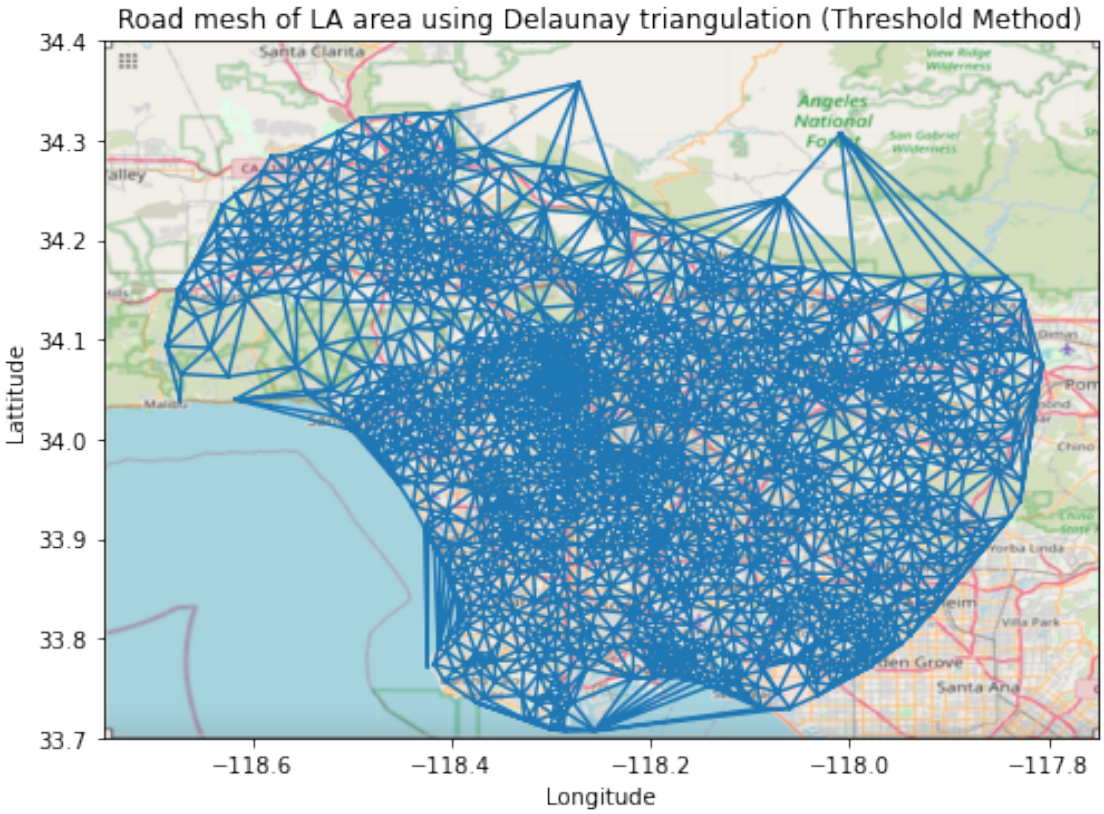
0.7 Question 14

[23]:
```
g_del_prune=ig.Graph()
g_del_prune.add_vertices(len(delaunay_out.points))
nodes=g_del_prune.vs()
i=0
for node in nodes:
    node['latitude']=lat[i]
    node['longitude']=lon[i]
    i+=1

threshold=(19.2/69)

duplicate_remove=set()
edge_cut=set()
for i in range(len(delaunay_out.simplices)):
    a=((delaunay_out.simplices[i][0], delaunay_out.simplices[i][1]))
    b=((delaunay_out.simplices[i][0], delaunay_out.simplices[i][2]))
    c=((delaunay_out.simplices[i][1], delaunay_out.simplices[i][2]))
    list(a).sort()
    list(b).sort()
    list(c).sort()
    if not a in duplicate_remove:
```

```
        duplicate_remove.add(a)
        a_dist_check=np.
↪sqrt((nodes[a[0]]['latitude']-nodes[a[1]]['latitude'])**2+(nodes[a[0]]['longitude']-nodes[a
        if a_dist_check<threshold:
            g_del_prune.add_edges([a])
        else:
            edge_cut.add(a)
    if not b in duplicate_remove:
        duplicate_remove.add(b)
        b_dist_check=np.
↪sqrt((nodes[b[0]]['latitude']-nodes[b[1]]['latitude'])**2+(nodes[b[0]]['longitude']-nodes[b
        if b_dist_check<threshold:
            g_del_prune.add_edges([b])
        else:
            edge_cut.add(b)
    if not c in duplicate_remove:
        duplicate_remove.add(c)
        c_dist_check=np.
↪sqrt((nodes[c[0]]['latitude']-nodes[c[1]]['latitude'])**2+(nodes[c[0]]['longitude']-nodes[c
        if c_dist_check<threshold:
            g_del_prune.add_edges([c])
        else:
            edge_cut.add(c)

edge_cut_list=list(edge_cut)
simplices_list=[]
for simplex in delaunay_out.simplices:
    for edge in edge_cut_list:
        if edge[0] in simplex and edge[1] in simplex:
            simplices_list.append(list(simplex))

new_delaunay_out_desimplices = [i for i in delaunay_out.simplices if i not in␣
↪np.array(simplices_list)]
```

[24]:
```
BBox = ((-118.75,   -117.75,
         33.7, 34.4))
ruh_m = plt.imread('map_LA.png')

fig, ax = plt.subplots(figsize = (8,7))
plt.triplot(lon, lat, new_delaunay_out_desimplices)
ax.set_title('Road mesh of LA area using Delaunay triangulation (Threshold␣
↪Method)')
plt.xlabel('Longitude')
plt.ylabel('Lattitude')
ax.set_xlim(BBox[0],BBox[1])
ax.set_ylim(BBox[2],BBox[3])
ax.imshow(ruh_m, zorder=0, extent = BBox, aspect= 'equal')
```

```
plt.savefig('Q14a.png',dpi=300,bbox_inches='tight')
plt.show()
```



[25]:
```
visual_style = {}
visual_style["vertex_size"] = 3
ig.plot(g_del_prune,**visual_style)
```

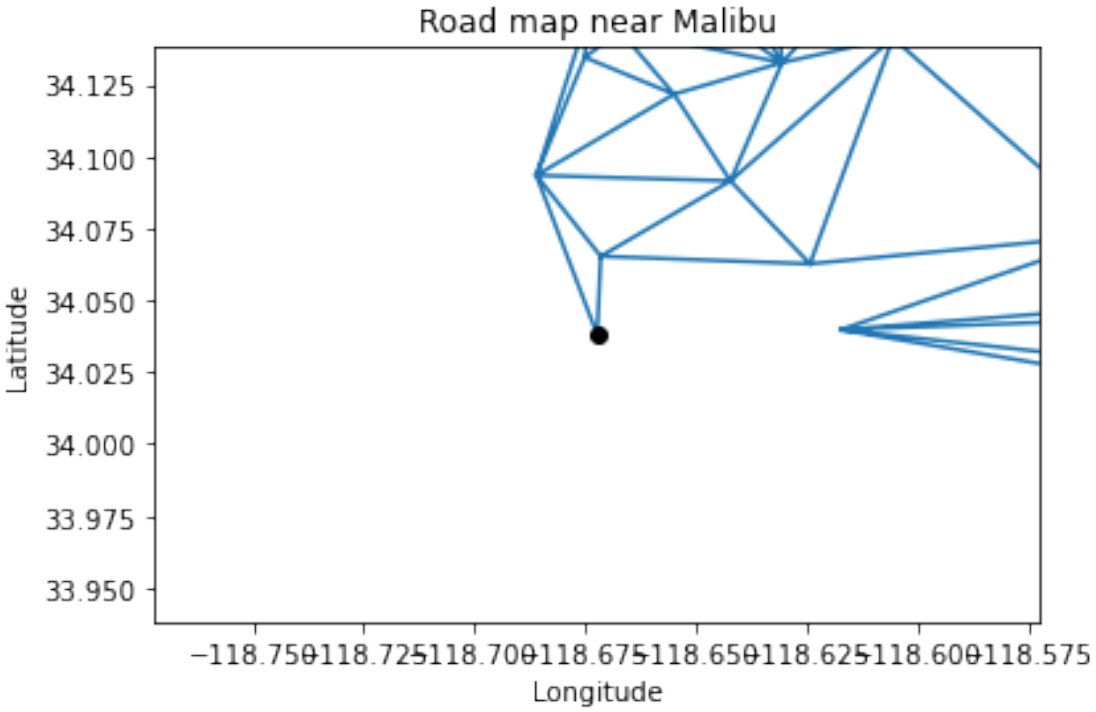
[29]:
```
out = ig.plot(g_del,**visual_style)
out.save('Q14b.png')
```

0.8 Question 15

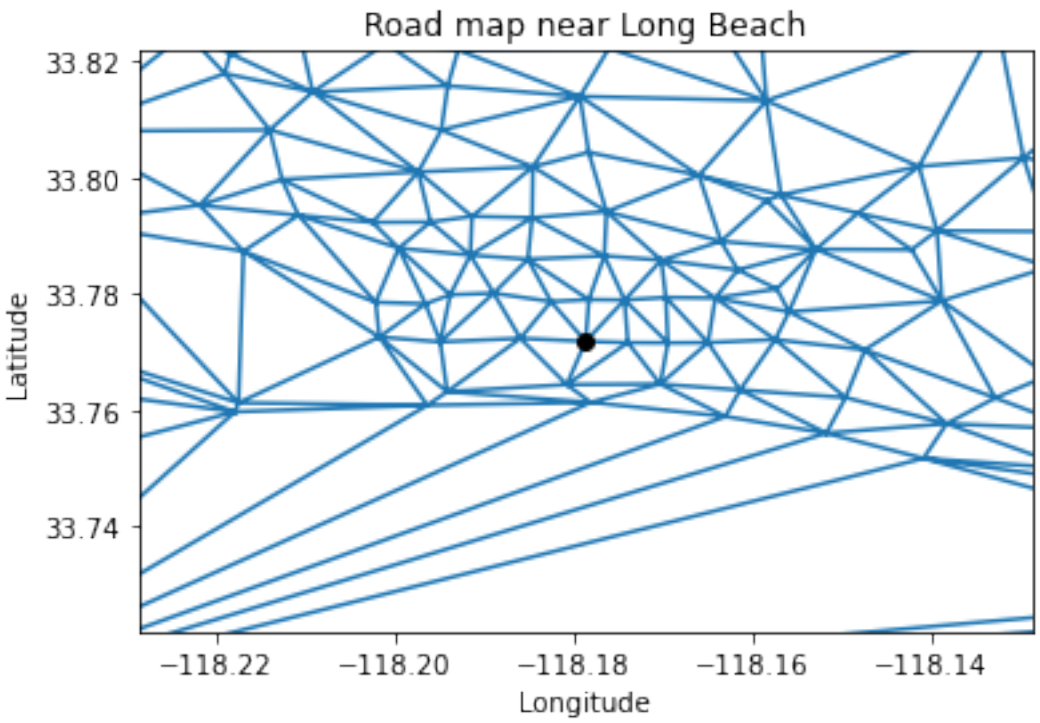
[26]:
```
print('Number of edge-disjoint paths: ',g_del_prune.
 ↪adhesion(long_beach_node,malibu_node)-1)
print('Degree Distribution of nodes (Malibu, Long Beach): ', g_del_prune.
 ↪degree(malibu_node,mode='out')-1,g_del_prune.
 ↪degree(long_beach_node,mode='in')-1)
```

```
Number of edge-disjoint paths:  3
Degree Distribution of nodes (Malibu, Long Beach):  3 8
```

[32]:
```
plt.triplot(lon, lat, new_delaunay_out_desimplices)
plt.ylim(lat_lon[malibu_node][0]-0.1,lat_lon[malibu_node][0]+0.1)
plt.xlim(lat_lon[malibu_node][1]-0.1,lat_lon[malibu_node][1]+0.1)
plt.xlabel('Longitude')
plt.ylabel('Latitude')
plt.title('Road map near Malibu')
plt.plot(lon[malibu_node], lat[malibu_node],'o',color='black')
plt.savefig('Q15a.png',dpi=300,bbox_inches='tight')
plt.show()
```



[33]:
```
plt.triplot(lon, lat, new_delaunay_out_desimplices)
plt.ylim(lat_lon[long_beach_node][0]-0.05,lat_lon[long_beach_node][0]+0.05)
plt.xlim(lat_lon[long_beach_node][1]-0.05,lat_lon[long_beach_node][1]+0.05)
plt.xlabel('Longitude')
plt.ylabel('Latitude')
plt.title('Road map near Long Beach')
plt.plot(lon[long_beach_node], lat[long_beach_node],'o',color='black')
plt.savefig('Q15b.png',dpi=300,bbox_inches='tight')
plt.show()
```


Road map near Long Beach
33.82
33.80
33.78
33.76
33.74
Latitude
−118.22
−118.20
−118.18
−118.16
−118.14
Longitude

[ ]:

Own_Task

June 7, 2021

[1]:
```
import numpy as np
import pandas as pd
import pulp
import itertools
import gmaps
import googlemaps
import warnings
warnings.filterwarnings('ignore')
import matplotlib.pyplot as plt
```

[2]:
```
API_KEY = 'AIzaSyCahBFOfV0cEukC4sPZNPHcVfZTkedoBUw'
gmaps.configure(api_key=API_KEY)
googlemaps = googlemaps.Client(key=API_KEY)
```

[17]:
```
customer_count = 10
vehicle_count = 4
vehicle_capacity = 50
np.random.seed(seed=777)
depot_latitude = 40.748817
depot_longitude = -73.985428

df = pd.DataFrame({"latitude":np.random.normal(depot_latitude, 0.007,␣
 ↪customer_count),
                   "longitude":np.random.normal(depot_longitude, 0.007,␣
 ↪customer_count),
                   "package_count":np.random.randint(10, 20, customer_count)})
```

[18]:
```
df['package_count'][0] = 0
df['latitude'][0] = depot_latitude
df['longitude'][0] = depot_longitude
```

[19]:
```
print(df)
```

```
    latitude  longitude  package_count
0  40.748817 -73.985428              0
1  40.743057 -73.972162             14
2  40.748359 -73.990814             10
3  40.743823 -73.995250             16
```

```
4  40.755161 -73.989855             15
5  40.754181 -73.989340             17
6  40.754599 -73.994061             15
7  40.739551 -73.988505             15
8  40.736550 -73.979024             18
9  40.755834 -73.983573             11
```

[6]:
```
def _plot_on_gmaps(_df):

    _marker_locations = []
    for i in range(len(_df)):
        _marker_locations.append((_df['latitude'].iloc[i],_df['longitude'].
↪iloc[i]))

    _fig = gmaps.figure()
    _markers = gmaps.marker_layer(_marker_locations)
    _fig.add_layer(_markers)

    return _fig

def _distance_calculator(_df):

    _distance_result = np.zeros((len(_df),len(_df)))
    _df['latitude-longitude'] = '0'
    for i in range(len(_df)):
        _df['latitude-longitude'].iloc[i] = str(_df.latitude[i]) + ',' +␣
↪str(_df.longitude[i])

    for i in range(len(_df)):
        for j in range(len(_df)):
            _google_maps_api_result = googlemaps.
↪directions(_df['latitude-longitude'].iloc[i],
                                                                    ␣
↪_df['latitude-longitude'].iloc[j],
                                                                     mode = 'driving')
            _distance_result[i][j] =␣
↪_google_maps_api_result[0]['legs'][0]['distance']['value']

    return _distance_result
```

[7]:
```
distance = _distance_calculator(df)
plot_result = _plot_on_gmaps(df)
plot_result
```

```
Figure(layout=FigureLayout(height='420px'))
```

[8]:
```
for vehicle_count in range(1,vehicle_count+1):

    problem = pulp.LpProblem("CVRP", pulp.LpMinimize)
    x = [[[pulp.LpVariable("x%s_%s,%s"%(i,j,k), cat="Binary") if i != j else␣
↪None for k in range(vehicle_count)]for j in range(customer_count)] for i in␣
↪range(customer_count)]
    problem += pulp.lpSum(distance[i][j] * x[i][j][k] if i != j else 0
                          for k in range(vehicle_count)
                          for j in range(customer_count)
                          for i in range (customer_count))
    for j in range(1, customer_count):
        problem += pulp.lpSum(x[i][j][k] if i != j else 0
                              for i in range(customer_count)
                              for k in range(vehicle_count)) == 1

    for k in range(vehicle_count):
        problem += pulp.lpSum(x[0][j][k] for j in range(1,customer_count)) == 1
        problem += pulp.lpSum(x[i][0][k] for i in range(1,customer_count)) == 1

    for k in range(vehicle_count):
        for j in range(customer_count):
            problem += pulp.lpSum(x[i][j][k] if i != j else 0
                                  for i in range(customer_count)) -  pulp.
↪lpSum(x[j][i][k] for i in range(customer_count)) == 0
    for k in range(vehicle_count):
        problem += pulp.lpSum(df.no[j] * x[i][j][k] if i != j else 0 for i in␣
↪range(customer_count) for j in range (1,customer_count)) <= vehicle_capacity

    subtours = []
    for i in range(2,customer_count):
         subtours += itertools.combinations(range(1,customer_count), i)

    for s in subtours:
        problem += pulp.lpSum(x[i][j][k] if i !=j else 0 for i, j in itertools.
↪permutations(s,2) for k in range(vehicle_count)) <= len(s) - 1

    if problem.solve() == 1:
        print('Vehicle Requirements:', vehicle_count)
        print('Moving Distance:', pulp.value(problem.objective))
        break
```

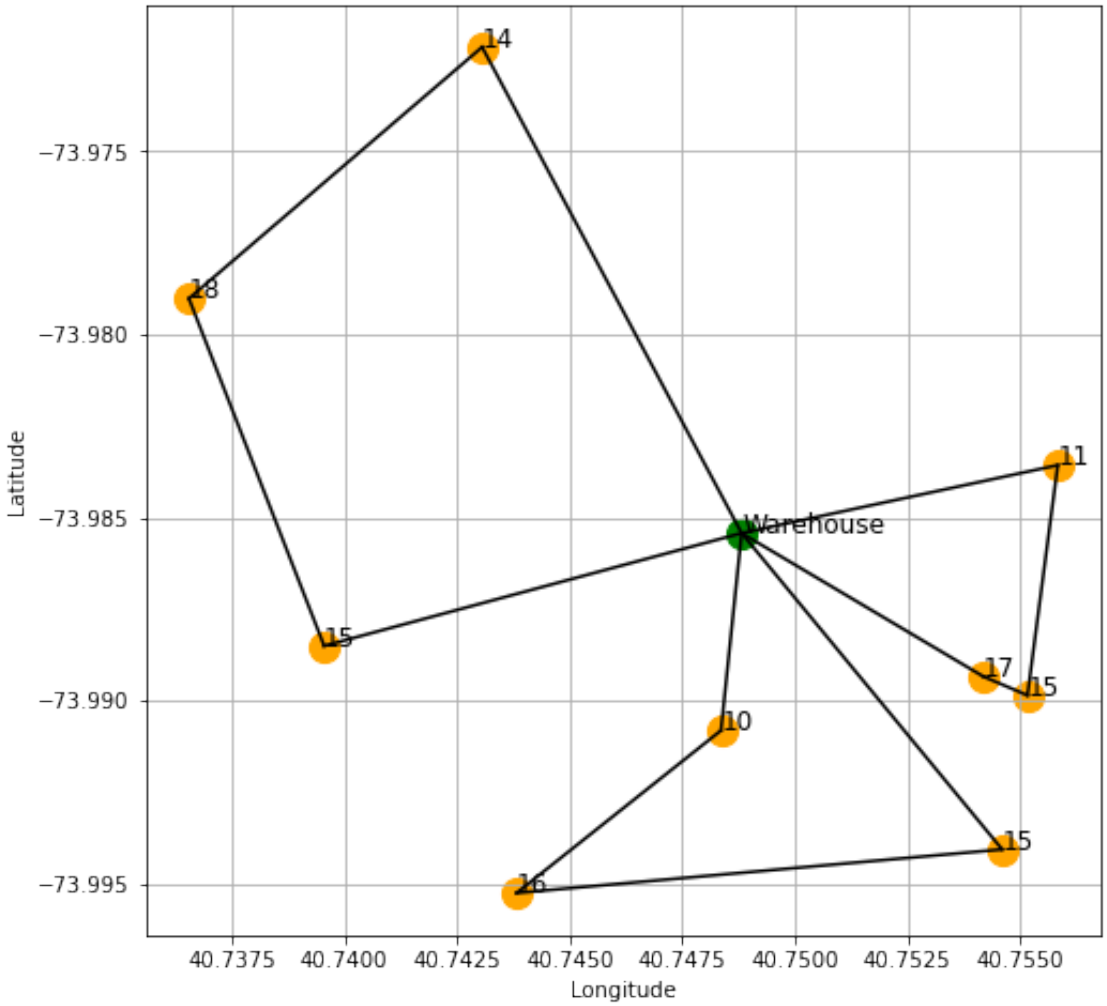
```
Vehicle Requirements: 3
Moving Distance: 14075.0
```

[9]:
```
fig = gmaps.figure()
layer = []
color_list = ["red","blue","green"]
```

```
for k in range(vehicle_count):
    for i in range(customer_count):
        for j in range(customer_count):
            if i != j and pulp.value(x[i][j][k]) == 1:
                layer.append(gmaps.directions.Directions(
                    (df.latitude[i],df.longitude[i]),
                    (df.latitude[j],df.longitude[j]),
                    mode='car',stroke_color=color_list[k],stroke_opacity=1.0,␣
 ↪stroke_weight=5.0))

for i in range(len(layer)):
    fig.add_layer(layer[i])

fig
```

```
Figure(layout=FigureLayout(height='420px'))
```

[12]:
```
plt.figure(figsize=(8,8))
for i in range(customer_count):
    if i == 0:
        plt.scatter(df.latitude[i], df.longitude[i], c='green', s=200)
        plt.text(df.latitude[i], df.longitude[i], "Warehouse", fontsize=12)
    else:
        plt.scatter(df.latitude[i], df.longitude[i], c='orange', s=200)
        plt.text(df.latitude[i], df.longitude[i], str(df.package_count[i]),␣
 ↪fontsize=12)

for k in range(vehicle_count):
    for i in range(customer_count):
        for j in range(customer_count):
            if i != j and pulp.value(x[i][j][k]) == 1:
                plt.plot([df.latitude[i], df.latitude[j]], [df.longitude[i], df.
 ↪longitude[j]], c="black")

plt.xlabel('Longitude')
plt.ylabel('Latitude')
plt.grid()
plt.savefig('Own_task.png',dpi=300,bbox_inches='tight')
plt.show()
```


14
18
11
Warehouse
15
17
15
10
15
16
−73.975
−73.980
−73.985
−73.990
−73.995
Latitude
40.7375
40.7400
40.7425
40.7450
40.7475
40.7500
40.7525
40.7550
Longitude

[ ]: