University of California, Los Angeles
Department of Electrical and Computer Engineering
ECE 232E Large Scale Social and Complex Networks: Design and Algorithms

Project 4: Graph Algorithms

505430686 Viacheslav Inderiakin

904627828 Mia Levy

805626088 Connor Roberts

804737257 Tameez Latib

June 9, 2021

1. Stock Market

Question 1: What are upper and lower bounds on ρ_{ij}? Provide a justification for using log-normalized return ($r_i(t)$) instead of regular return ($q_i(t)$).

Answer: Note that we can reformulate the expression in probability terms. Let X_i be a random variable taking the value $r_i(t)$ with probability $1/N$ given N samples. Then the temporal average is the expectation. Further, we get that the correlation

$$Cor(X_i, X_j) = p_{ij} = \frac{E(X_i X_j) - E(X_i)E(X_j)}{\sqrt{V(X_i)V(X_j)}}$$

and by Cauchy Schwarz we have that

$$|E(X_i X_j) - E(X_i)E(X_j)|^2 = |E[(X_i - E(X_i))(X_j - E(X_j))]|^2$$

$$\leq E[(X_i - E(X_i))^2]E[(X_j - E(X_j))^2] = V(X_i)V(X_j)$$

And therefore we have that

$p_{ij}^2 \leq \frac{V(X_i)V(X_j)}{V(X_i)V(X_j)} = 1$

I.e. p_{ij} is bounded in magnitude by 1.

It makes sense to use the log normalized return because we can reduce numerical/computation errors by using normalized numbers

Question 2: Plot a histogram showing the un-normalized distribution of edge weights.

Answer: See figure 1

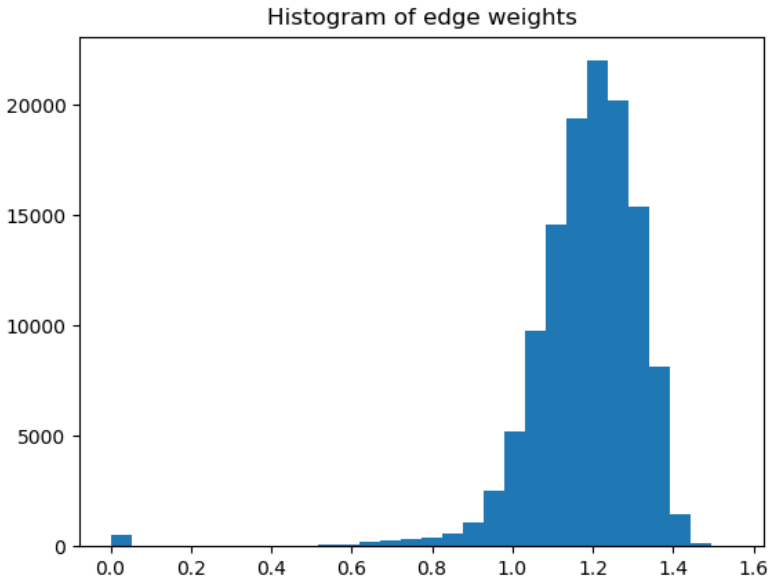


Figure 1: Histogram of edge weights.

Question 3: Extract the MST of the correlation graph. Each stock can be categorized into a sector, which can be found in `Name_sector.csv` file. Plot the MST and color-code the nodes based on sectors. Do you see any pattern in the MST? The structures that you find in MST are called Vine clusters. Provide a detailed explanation about the pattern you observe.

Answer: See figure 2. Note that in the MST, neighbors are more likely to be similarly colored. This is because there should be a higher correlation between stocks in the same sector. That is, if there is a medical breakthrough, most of the healthcare sector stocks should see increase (whereas other stocks may not be as affected by medical breakthroughs). Therefore, they will be highly correlated if they are in the same sector and thus have lower edge weight. So in the MST they will be neighbors. This is why we get the "Vine Clusters"

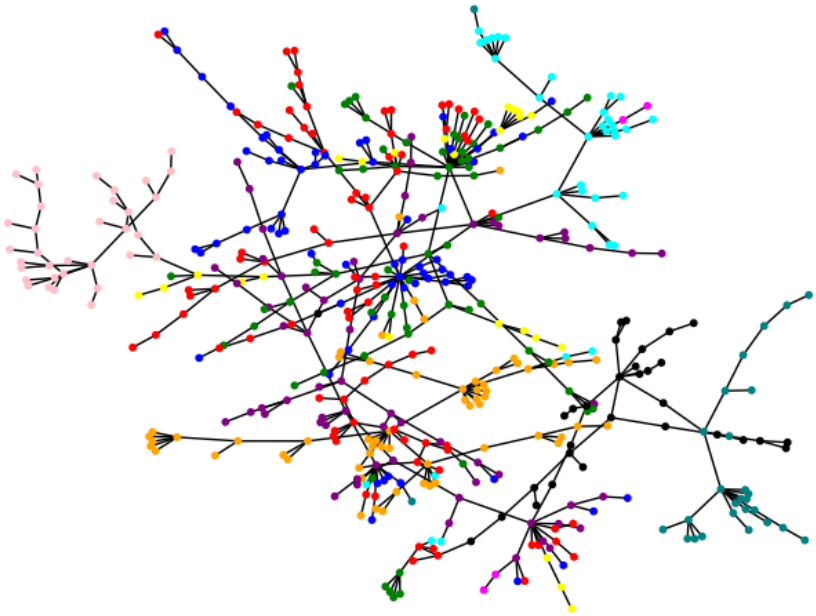


Figure 2: MST with color coded nodes based on sectors.

Question 4: Report the value of α for the above two cases and provide an interpretation for the difference.

Answer: $\alpha_1 = 0.8289300775306759, \alpha_2 = 0.11435042289654039$

Note that the P in α_1 formula represents the ratio of neighbors in your (the nodes) sector. Therefore if we average over all nodes to get α_1, we should see a very high value for this MST because of the vine clusters. That is, on average, most node neighbors are in the same sector. And we do see a α_1 close to 1. So if we needed to predict an unknown node, we can check the sectors of its neighbors!

For the P in α_2, this represents the ratio of how many nodes are in your sector to how many nodes are in the graph. This is not as good of an estimate or predictor because it deals with global data and does not take into account a node's local connections/neighbors. Note that $\alpha_2 = \frac{|S_1|^2}{|V|^2} + \frac{|S_2|^2}{|V|^2} + ...$ So if all our sectors had roughly equal amounts of nodes, and we have M sectors, then roughly $\alpha_2 \approx 1/M$. However, let's say that S_1 has substantially more nodes than the other sectors. That is, $|S_1|$ is close to $|V|$, then

$\alpha_2 \approx \frac{|S_1|^2}{|V|^2}$ will be large. So α_2 being somewhat small in our case is telling us that most of our sectors have roughly an equal number of nodes.

Question 5: Extract the MST from the correlation graph based on weekly data. Compare the pattern of this MST with the pattern of the MST found in Question 3.

Answer: See figure 3

The vine clusters in the weekly data are still present, but are less pronounced. Since we are only using weekly data it is possible that stocks from different sectors have higher correlation than previously, since it could be that the entire market is going up or down. Indeed, we can check the α_1 value and see that it is less than that of Q4. Here $\alpha_1 = 0.7429696034959192$, which is less than that of the daily data. Note that it is still high, and so the vine clusters still appear, but because it is lower than before there are now many points in the graph where nodes of different color appear in the vine clusters.

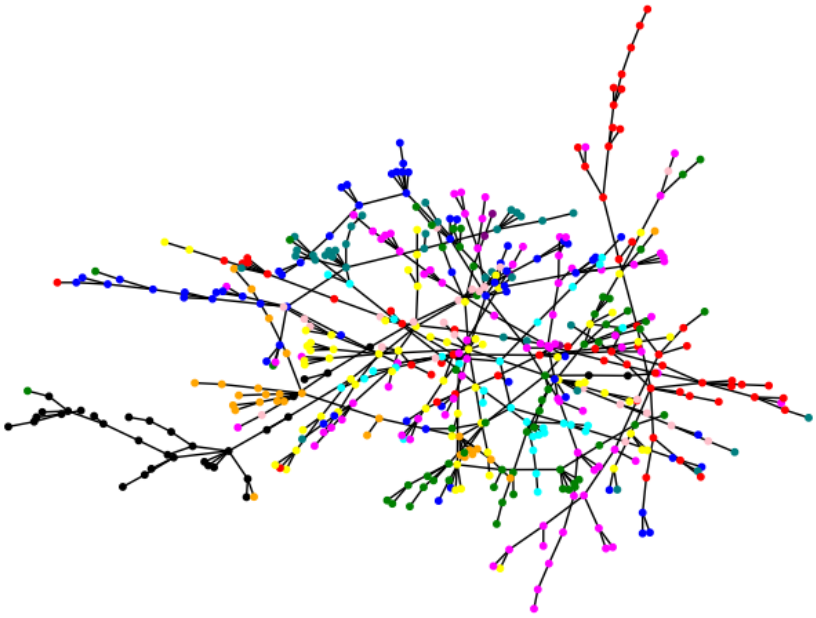


Figure 3: MST with color coded nodes based on sectors (Weekly data).

2. Uber Dataset

Question 6: Read the dataset at hand, and build a graph in which nodes correspond to locations, and undirected weighted edges correspond to the mean traveling times between each pair of locations (only December). Add the centroid coordinates of each polygon region (a 2-D vector) as an attribute to the corresponding vertex. The graph will contain some isolated nodes (extra nodes existing in the Geo Boundaries JSON file) and a few small connected components. Remove such nodes and just keep the largest connected component of the graph. In addition, merge duplicate edges by averaging their weights. We will refer to this cleaned graph as G afterwards. Report the number of nodes and edges in G.

Answer: The graph G created according above-mentioned instructions is shown in figure 4. The total number of nodes and edges is provided below:

$$
\begin{aligned}
N_{nodes} &= 2649 \\
N_{edges} &= 1003858
\end{aligned}
$$

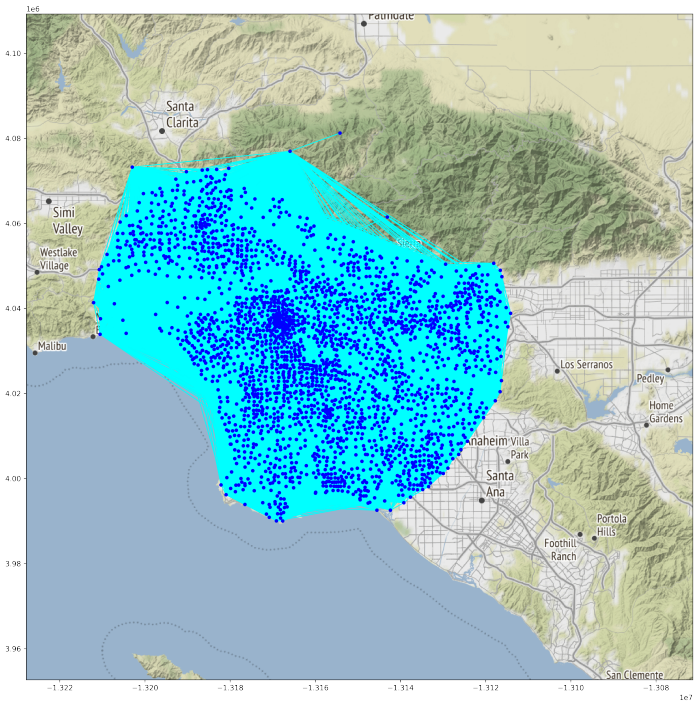


Figure 4: Uber Movement graph G.

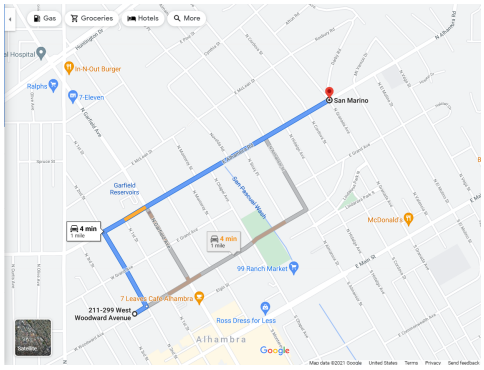
Question 7: Build a minimum spanning tree (MST) of graph G. Report the street addresses near the two endpoints (the centroid locations) of a few edges. Are the results intuitive?

Answer: The minimum spanning tree of graph G is provided in figure 6.

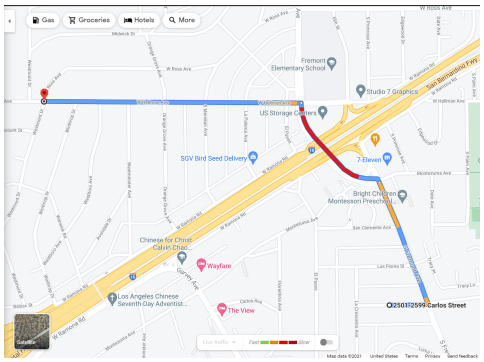
To check whether this result makes sense we can take a look at a few MST's edges. For example, edges 0, 25 and 100 are connecting nodes with uber-ids 1 and 3, 21 and 22, 78 and 79 correspondingly. We retrieved coordinates of the centroids of those edges, found the addresses closest to said coordinates and plotted the routes connecting them using google maps. The results are provided in figure 5, and address are provided in table 1. This figure shows that the locations connected by MST are placed very close to each other, and are less than 5 minute drive apart. From this we can conclude that MST was found successfully.

Edge id	Address 1	Address 2
0	823 E. Grand Ave, Alhambra	217 W Woodward Ave, Alhambra
25	3025 W Hellman Ave, Alhambra	2529 Carlos St, Alhambra
100	6615 Bear Ave, Bell	661 Riverside Ave, Bell

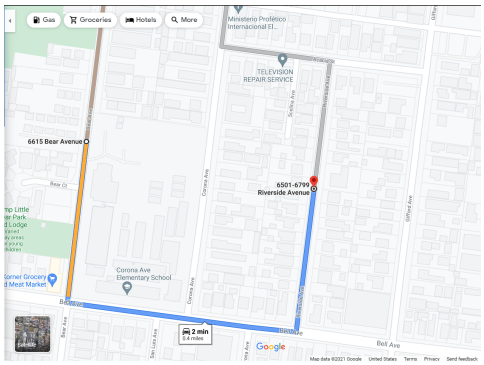
Table 1: Street addresses closest to the centroid locations.



(a) Edge 0.



(b) Edge 25.



(c) Edge 100.

Figure 5: Edges plotted as routes on google map.

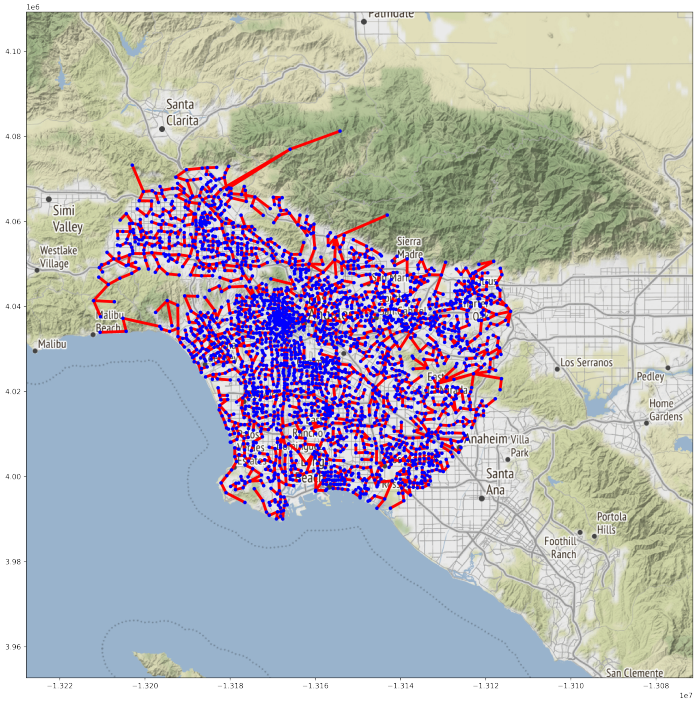


Figure 6: MST of G.

Question 8: Determine what percentage of triangles in the graph (sets of 3 points on the map) satisfy the triangle inequality. You do not need to inspect all triangles, you can just estimate by random sampling of 1000 triangles.

Answer: Approximately ≈ 0.923 of the triangles satisfy triangle inequality. We can see that for the most of the triangles triangle inequality holds.

Note: as per TA's advice, we assumed that a triangle is a set of 3 vertices directly connected by edges.

Question 9: Now, we want to find an approximation solution for the traveling salesman problem (TSP) on G. Apply the 1-approximate algorithm described in the class. Inspect the sequence of street addresses visited on the map and see if the results are intuitive. Find an upper bound on the empirical performance of the approximate algorithm:

$$\rho = \frac{Approximate\ \ TSP\ \ Cost}{Optimal\ \ TSP\ \ Cost}$$

Answer: The TSP problem solution in a form of edge / vertex id sequences was found in jupyter notebook (see code on CCLE). To check whether solution makes sense, we inspected 5 consecutive destinations on the path. Full addresses of destinations we inspected are the follows:

1. 1023 Fortune Way

2. 6169 Meridian St
3. 301 Neva Pl
4. 301 Neva Pl
5. 376 North Ave 57

A trajectory trough these points retrieved from google maps is provided in figure 7. As we can see from the figure, the destinations are placed close to each other and approximate TSP algorithm performed well.

The optimal TSP solution is bounded from the bottom by the total length of MST. Simultaneously, by construction approximate solution of TSP is an Eulerian cycle on directed graph induced from MST, and has maximal length of 2 MST-s. Therefore, $\rho \leq 2$.

Note: the discussion above holds true if all triangles in the graph satisfy triangle inequality. While this is not precisely true for G, this condition is satisfied for most of the graph (see question 9), so we can assume that bounds on approximate and optimal solution remain in place.

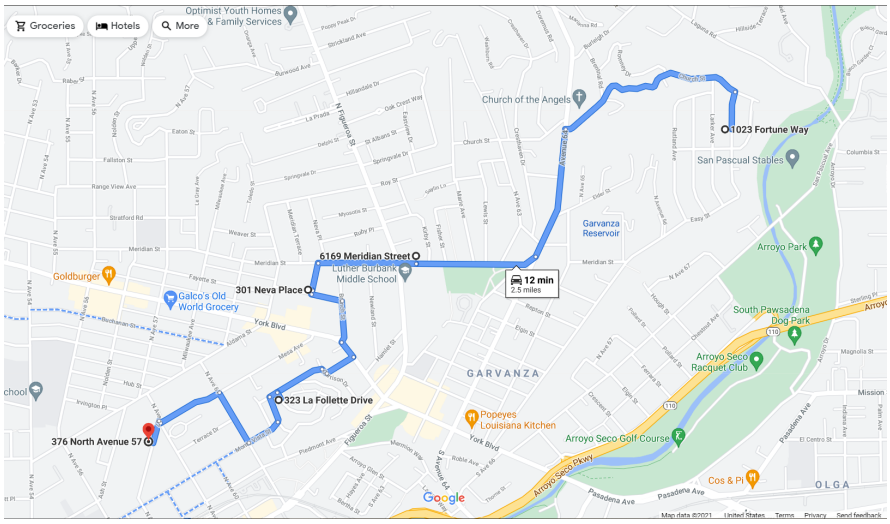


Figure 7: A sequence of addresses from approximate TSP problem solution.

Question 10: Plot the trajectory that Santa has to travel!

Answer: A solution of TSP in a form of trajectory to follow is plotted in figure 8. Consecutive edges are close in terms of colors, with first roads to take having cyan color and last having purple color. From that figure we can see that the algorithm successfully covers each location on the graph.

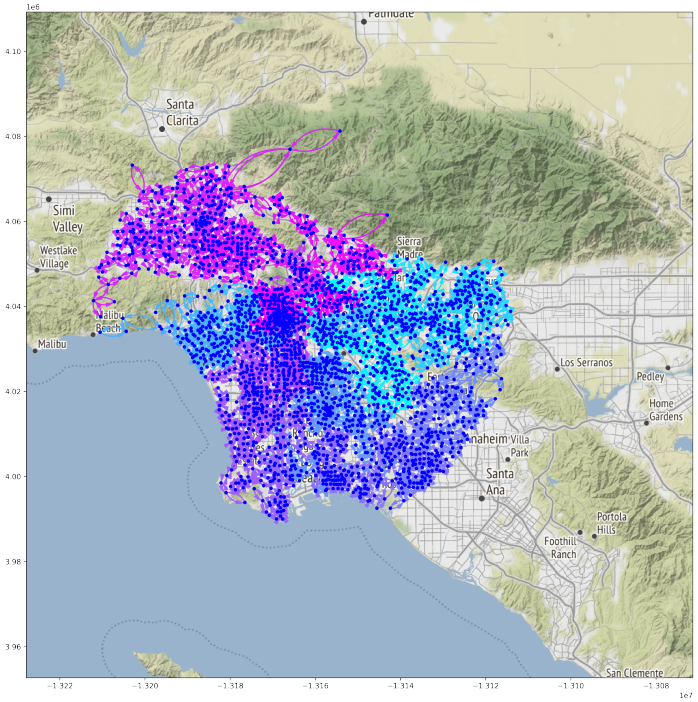


Figure 8: The trajectory Snata needs to follow.

Question 11: Educate yourself about Delaunay triangulation algorithm and then apply it to the nodes coordinates. Plot the road mesh that you obtain and explain the result. Create a graph G_Δ whose nodes are different locations and its edges are produced by triangulation.

Answer: Delaunay triangulation is an algorithm for creating a mesh between a given set of points in a way such that for any triangle in a mesh no other point from a set lies inside the circumcircle of said triangle. By running this algorithm on a collection of source and destination point coordinates we can estimate coordinates of the roads in real life. The graph G_Δ obtained by following Delaunay triangulation algorithm is shown in figure 9.

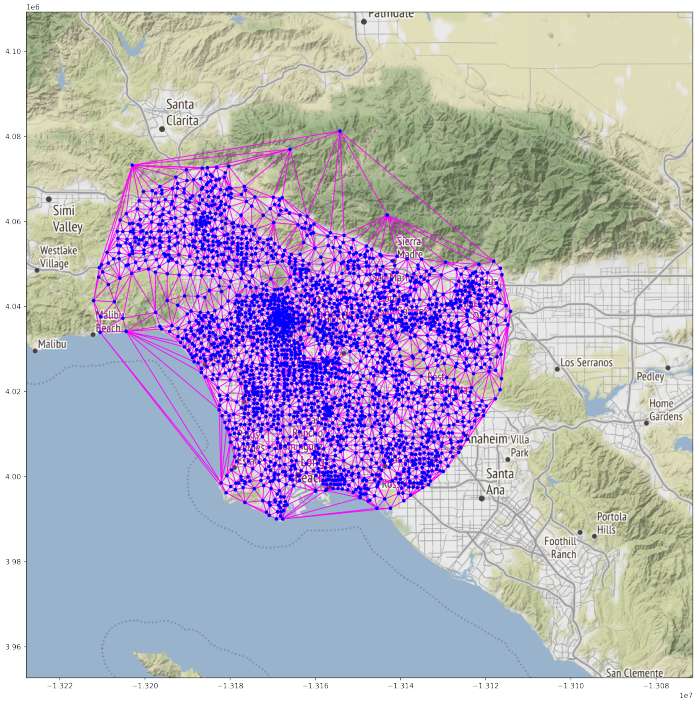


Figure 9: Graph G_Δ.

Question 12: Using simple math, calculate the traffic flow for each road in terms of cars/hour. Report your derivation. Assuming no traffic jam, consider the calculated traffic flow as the max capacity of each road.

Answer: To calculate flow between any 2 points connected by an edge in G_Δ, we followed an algorithm which steps are described below:

1. Calculate the distance between points as $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$
2. Multiply the obtained value by 69 to get the distance in miles.
3. Get an average time needed to travel between vertices. If an edge connecting these points exists in graph G, its weight (or mean travel time) is the time we need. If it does not, compute shortest path (in terms of mean travel time) for the points in G and get its total time.
4. Divide distance obtained in 2 by travel time obtained in 3. The result will be an average speed in miles per seconds.
5. Multiply average speed by 2 seconds to get safety distance between cars
6. Add safety distance to car length to get a total space a car takes on the road
7. Multiply speed by 3600 and divide total space from 6 to get the flow of one lane
8. Replace the undirected edge with 2 edges going in opposite directions, each one with flow equal to 2 flows of a lane (since each road has 2 lanes in each direction)

Question 13: Calculate the maximum number of cars that can commute per hour from Malibu to Long Beach. Also calculate the number of edge-disjoint paths between the two spots. Does the number of edge-disjoint paths match what you see on your road map?

Answer: The maximal number of cars that can commute from Malibu to Long beach and the number of edge-disjoint path is provided below

$$N_{G_\Delta}^{cars} \approx 11074$$
$$N_{G_\Delta}^{edge-disjoint\ \ paths} = 4$$

Edge disjoint paths and one of the possible road sets that can provide maximal flow of cars are plotted in figure 10. As we can see from that figure the number of edge-disjoint path is limited by the out-degree of Malibu node (red dot closer to the top of the map). However, we can also see that both max flow and edge-disjoint paths utilize roads that go over the water and therefore do not exist in real life.

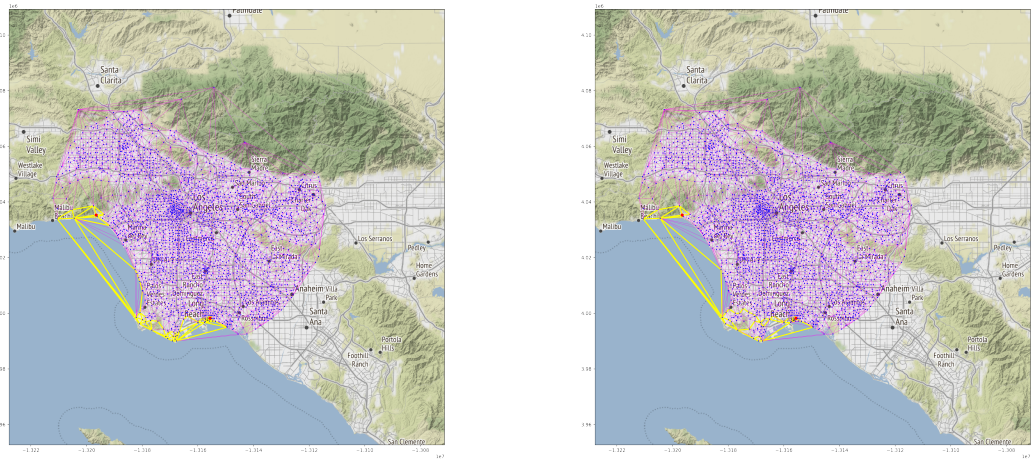


Figure 10: An example of a set of edges that can provide for maximal flow of cars (left) and edge-disjoint paths between Malibu and Long Beach in G_Δ.

Question 14: In G_Δ, there are a number of unreal roads that could be removed. For instance, you might notice some unreal links along the concavities of the beach, as well as in the hills of Topanga. Apply a threshold on the travel time of the roads in G_Δ to remove the fake edges. Call the resulting graph $\tilde{G}_\Delta$. Plot $\tilde{G}_\Delta$ on actual coordinates. Do you think the thresholding method worked?

Answer: Assuming that travel time through an unrealistic road is longer than through a real road (because it is a composition of travel times through several edges), we applied a threshold for travel times. All roads that have travel times beyond 600 sec = 10 min were removed. The result (graph $\tilde{G}_\Delta$) is plotted in figure 11. We can see that thresholding travel times successfully removes all roads that go through the ocean and most of the roads that go through the hills.

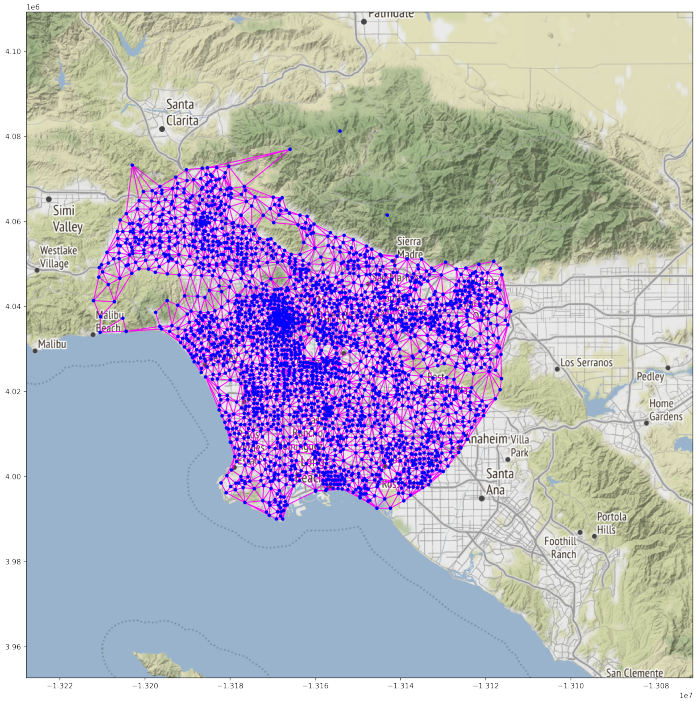


Figure 11: Graph $\tilde{G}_\Delta$.

Question 15: Now, repeat question 13 for $\tilde{G}_\Delta$ and report the results. Do you see any changes? Why?

Answer: The max flow and number of edge-disjoint paths for the cropped graph were recomputed as below:

$$N^{cars}_{\tilde{G}_\Delta} \approx 11074$$
$$N^{edge-disjoint\ \ paths}_{\tilde{G}_\Delta} = 4$$

We can see that neither maxflow nor the number of edge-disjoint paths has changed in comparison with G_Δ. The reason behind this is the fact that the bottleneck for both the car flow and the number of edge-disjoint paths is Malibu node, which has only 4 output edges which together can let 11074 cars through them. This can be also visually confirmed in figure 12. From that figure we can see that deletion of the unrealistic roads re-routed middle part of the trajectories for max flow and edge-disjoint paths, but selection of the edges near the source and destination remained the same, which tells us that the flow has not changed.

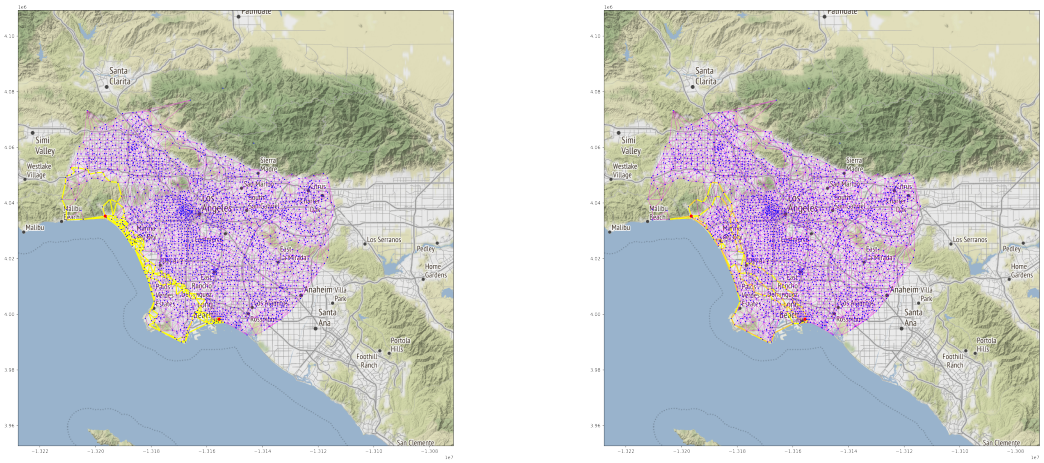


Figure 12: An example of a set of edges that can provide for maximal flow of cars (left) and edge-disjoint paths between Malibu and Long Beach in $\tilde{G}_{\Delta}$.

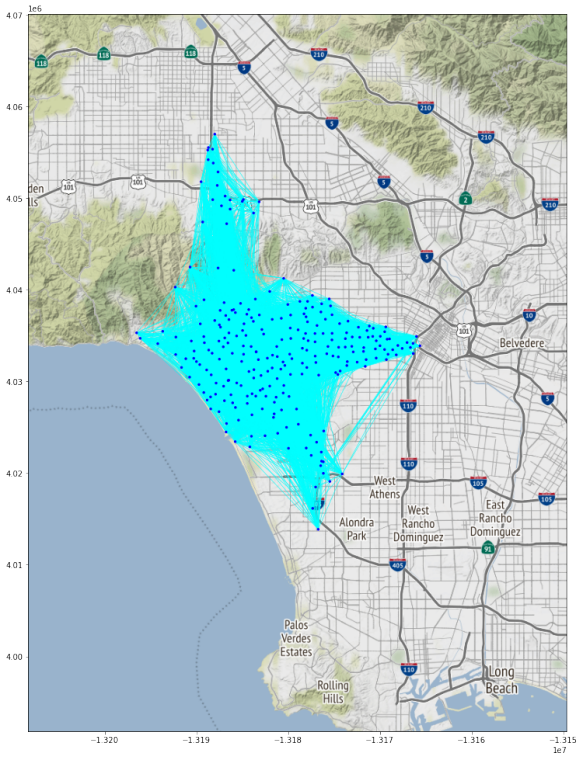
3. Define your own task

For this section, we use the Monthly Aggregated Uber data and compare the months of January and March to analyze changes in traffic between these two months. We expect to see that due to the pandemic, there will be less traffic in March, and therefore people should be able to travel longer distances in shorter amounts of time.

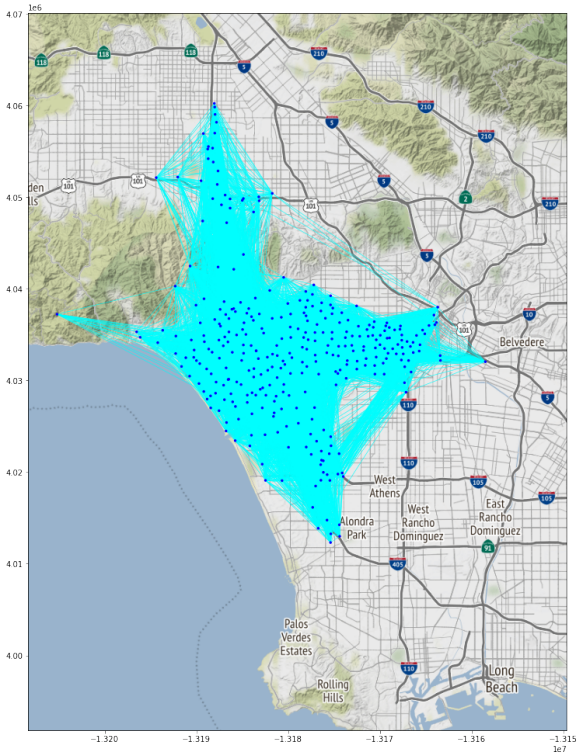
For this task, we use data from four of the columns in the monthly aggregated Uber data, for quarter 1 of 2020. The columns are source ID, destination ID, month, and mean travel time. Each line represents aggregated travel data between the source and the destination locations, and the mean travel time is the average travel time over the specified month.

The task can be defined as follows: First, we go through the files and pick out entries in which the travel time is less than 10 minutes. Then, a graph is constructed from these entries in which each node is a location found in the data. An edge is constructed between two nodes if the mean travel time between them is less than 10 minutes. Finally, an ego graph (personalized network) is constructed for the location that is visited the most in the data.

The above procedure is completed for January's entries, and then with March's entries, and analysis is performed on the two resulting personalized networks. We would expect to see more edges and a degree distribution that is skewed higher in the March graph, since there is less traffic and therefore it is faster to visit more locations from the ego node. Note that in 13, the ego graph in March appears more dense and wider reaching than in January.



(a) Ego graph in January.



(b) Ego graph in March

Figure 13: Ego graphs for 10 minute travel time from the most visited location

The same process is then repeated for travel times limited by 5 minutes and 30 minutes. As seen in 2 and 3, nodes, edges, and average degree of the ego graphs all increase from January to March.

	# Nodes	# Edges	Av. Degree
5 min	29	206	14.20
10 min	110	2403	43.69
30 min	1073	274041	510.79

Table 2: Statistics for January graphs

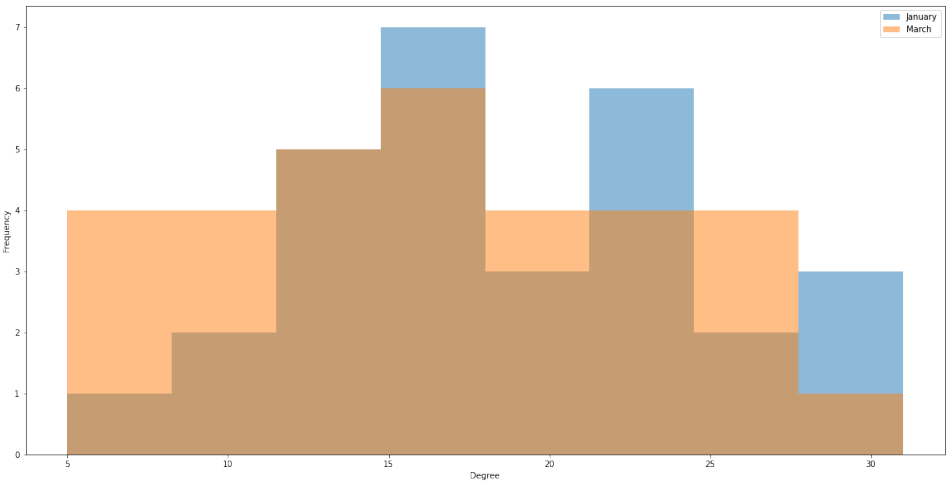
	# Nodes	# Edges	Av. Degree
5 min	32	265	16.56
10 min	127	3140	49.44
30 min	1115	293671	526.76

Table 3: Statistics for March graphs

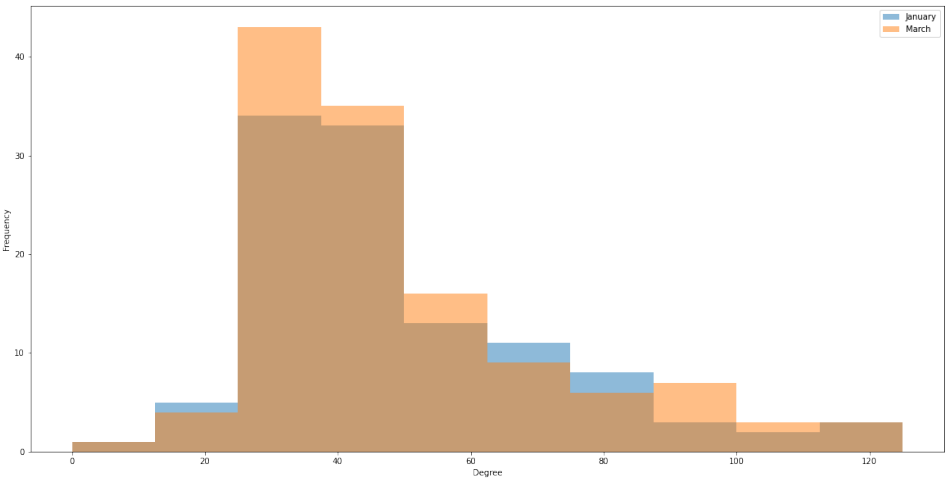
	Pre-Pandemic	Post-Pandemic
Average Distance Traveled	7.61 mi.	7.98 mi.
Max Distance Traveled	30.53 mi.	32.60 mi.

Table 4: Comparison of average distance traveled and max distance traveled in 30 minutes for both pre-pandemic and post-pandemic

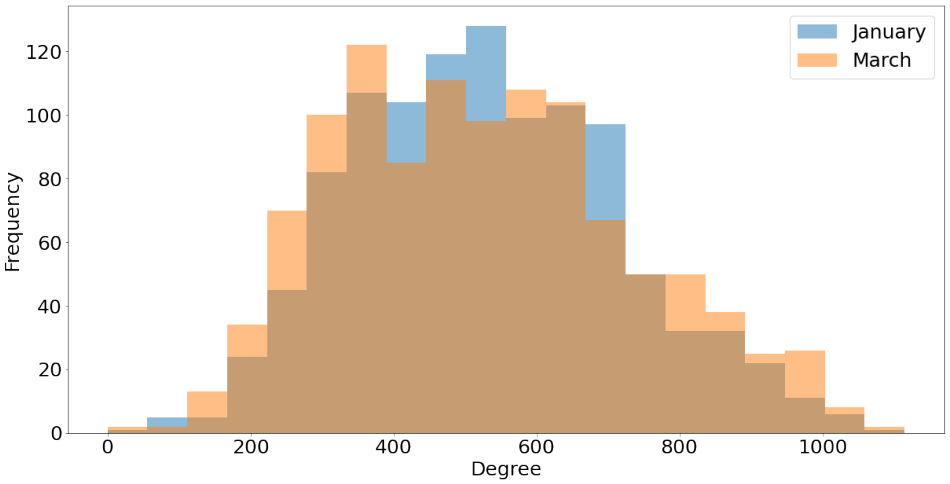
Next, the degree distributions are shown for each case. We expected to see a distribution that was skewed higher for March. As we can see in all three histograms below, March has more entries in almost every single bin compared to January. This shows us that there is correlation between the pandemic and what locations were able to be accessed in under 30 minutes. Even in the case where there are some bins in which January outperformed March, the number of bins where the opposite occurred greatly outweighed those favoring January.



(a) Degree Distribution 5 minute limit



(b) Degree Distribution 10 minute limit



(c) Degree Distribution 30 minute limit

Figure 14: Degree distributions for each ego graph.

```
 1 TESTING=False #only do first 100
 2 Q5 = True #weekly on mondays
 3 data_size = 765 #size of good data
 4 if Q5:
 5     data_size = 143
 6 print("+++ SETTINGS +++\n Q5={}, TEST={}\n
   +++ ------- +++".format(Q5,TESTING))
 7 import datetime
 8 import os
 9 import numpy as np
10 from csv import reader
11 import matplotlib.pyplot as plt
12 cols = "Date,Open,High,Low,Close,Volume,Adj Close"
13 data_path = "./finance_data/finance_data/data" #
   change depending on where data located
14
15 cwd = os.getcwd()
16 if "project 4" not in cwd:
17     print("Error: Must be in project 4 directory")
18     exit(1)
19
20 files = os.listdir(data_path)
21 #check to see if right dir.
22 print(files[:10], flush=True)
23
24 ##### Get all the data from csv files and store in
   arrays
25 stock_names = []
26 stock_data = {}
27 for fname in files:
28     #only take csv files
29     #print("fname: " + fname)
30     if "csv" in fname:
31         stockname = fname[:-4]
32         #Open the csv data file
33         with open(data_path + "/" + fname) as f:
34             csv_reader = reader(f)
35             csv_data = []
36             #first row is column field names
37             first_row = True
38             for row in csv_reader:
39                 if first_row:
40                     first_row = False
41                 else:
```

```
42                      if Q5: #only on mondays
43                          if "-" in row[0]: #check date
    format
44                              yyyy,mm,dd = row[0].split
   ("-")
45                          else:
46                              mm,dd,yyyy = row[0].split
   ("/")
47
48                          day = datetime.datetime(int(
   yyyy), int(mm), int(dd))
49                          if day.weekday() == 0: #
   monday
50                              csv_data.append(float(row
   [4]))
51
52                      else:
53                          csv_data.append(float(row[4
   ])) #Only closing price
54              #print("Closes: ", len(csv_data))
55              #Make sure that all data has same length
56              #print("len", len(csv_data))
57              if len(csv_data) == data_size:
58                  stock_names.append(stockname)
59                  stock_data[stockname] = csv_data
60                  if TESTING:
61                      if len(stock_names) >= 100:
62                          break
63
64
65 #deal with Name sector.csv (can't find it??)
66 stock_sectors = {}
67 with open("./finance_data/finance_data/Name_sector.
   csv") as f:
68     csv_reader = reader(f)
69     csv_data = []
70     #first row is column field names
71     first_row = True
72     for row in csv_reader:
73         if first_row:
74             first_row = False
75         else:
76             stock_sectors[row[0]] = row[1]
77
```

```
 78 #create color map
 79 color_from_sector = {}
 80 sectors = list(set(stock_sectors.values())) #unique
 81 print("sectors: ", sectors, flush=True)
 82 print(len(sectors)) #11
 83 colors = ['red', 'blue', 'green', 'yellow', 'orange'
    , 'black', 'purple', 'cyan', 'magenta', 'pink', '
    teal']
 84 for i in range(len(sectors)):
 85     color_from_sector[sectors[i]] = colors[i]
 86
 87 colormap = []
 88 for stock in stock_names:
 89     sector = stock_sectors[stock]
 90     color = color_from_sector[sector]
 91     colormap.append(color)
 92
 93 #print("=== check ===")
 94 #for i in range(10):
 95 #    print("stock: {}, sector: {}, color: {},{}".
    format(stock_names[i], stock_sectors[stock_names[i
    ]], color_from_sector[stock_sectors[stock_names[i
    ]]], colormap[i]))
 96 #print("=== ... ===")
 97
 98
 99 #Now convert closing prices to returns, and
    normalised returns
100 stock_returns = {}
101 stock_normalised_returns = {}
102 for stock in stock_data.keys():
103     returns = []
104     normalised_returns = []
105     closing_prices = stock_data[stock]
106     for i in range(len(closing_prices)-1):
107         ret = closing_prices[i+1] / closing_prices[i
    ] - 1 # ( pi+1 - pi ) / pi
108         returns.append( ret )
109         normalised_returns.append(np.log(1 + ret))
110
111     stock_returns[stock] = returns
112     stock_normalised_returns[stock] =
    normalised_returns
113
```

```
114
115 # get correlation data. Takes a while.
116 N_stocks = len(stock_names)
117 correlations = np.zeros((N_stocks,N_stocks)) # i =>
    stock_names[i]
118 for i in range(N_stocks):
119     if i % 5 == 0:
120         print("i: ", i, flush=True)
121     for j in range(i+1):
122         #Check this(?)
123         #print(stock_names[i], stock_names[j])
124         #print(len(stock_normalised_returns[
    stock_names[i]]), len(stock_normalised_returns[
    stock_names[j]]))
125         corr =  np.corrcoef(stock_normalised_returns
    [stock_names[i]], stock_normalised_returns[
    stock_names[j]])
126         #print("corr: ", corr[0,1])
127         correlations[i,j] = corr[0,1]
128         correlations[j,i] = corr[0,1]
129
130 #print("CORR")
131 #print(correlations[:10,:10], flush=True)
132
133 #w_ij
134 edge_weights = np.sqrt(2-2*correlations)
135 #print("Corr vs edge weight size!", edge_weights.
    shape, np.array(correlations).shape)
136 edge_weights_flat = []
137 for i in range(N_stocks):
138     for j in range(i+1):
139         edge_weights_flat.append(edge_weights[i,j])
140
141 #print("wij")
142 #print(edge_weights[:10,:10], flush=True)
143
144
145 plt.hist(edge_weights_flat, bins=30)
146 title = "Histogram of edge weights"
147 if Q5:
148     title = title + " (Weekly data)"
149 plt.title(title)
150 plt.savefig("results/" + title + ".png")
151 plt.show()
```

```
152
153 #get minimal spanning tree
154 import scipy
155 from scipy.sparse import csr_matrix
156 from scipy.sparse.csgraph import
    minimum_spanning_tree
157
158 print("Constructing MST")
159 MST = minimum_spanning_tree(csr_matrix(edge_weights
    ))
160
161 #test
162 #print("MST")
163 #print(MST[:10], flush=True)
164 MST = MST.todense()
165
166 #graph
167 import networkx as nx
168
169 gr = nx.Graph()
170 #get edges
171 print("Getting edges")
172 edges = []
173 for i in range(N_stocks):
174     for j in range(N_stocks):
175         if MST[i,j] > 0:
176             edges.append((i,j))
177             gr.add_edge(i,j)
178 ##gr.add_edges_from(edges) #edges
179 node_colormap = [colormap[node] for node in gr] #
    Since gr has nodes in wrong order :/
180 nx.draw(gr, node_size=10, node_color=node_colormap)
    #, labels=stock_names, with_labels=True)
181 title = "MST with nodes colored by sector"
182 if Q5:
183     title = title + " (Weekly data)"
184 plt.title(title)
185 plt.savefig("results/" + title + ".png")
186 plt.show()
187
188 #prediction, case 1:
189 alpha = 0
190 for i in range(N_stocks):
191     stock = stock_names[i]
```

```
192     # |Qi| / |Ni| , Qi = set of neighbors in the
    same sector as node i. Ni is set of neighbors
193     Ni = 0
194     Qi = 0
195     for j in range(N_stocks):
196         #If j a neighbor:
197         if MST[i,j] > 0 or MST[j,i] > 0:
198             Ni += 1
199             #If i and neighbor j in same sector:
200             if stock_sectors[stock_names[i]] ==
    stock_sectors[stock_names[j]]:
201                 Qi += 1
202     #print("i, Ni, Qi", i, Ni, Qi)
203     #print(MST[i,:])
204     #print(MST[:,i])
205     P = Qi/Ni
206     alpha += P #P vi in Si
207
208 alpha = alpha /N_stocks
209
210 #prediction, case 2:
211 alpha2 = 0
212 for i in range(N_stocks):
213     stock = stock_names[i]
214     # |Si| / |V| , Si = sector i, V = graph
215     Si = 0
216     V = len(stock_sectors)
217     #check how many stocks in same sector
218     for val in stock_sectors.values():
219         if (stock_sectors[stock_names[i]] == val):
220             Si += 1
221
222     P = Si/V
223     alpha2 += P #P vi in Si
224
225 alpha2 = alpha2 /N_stocks
226 print("alpha 1 vs 2:", alpha, alpha2)
```

part 2 python

June 9, 2021

1 Notes

Questions to ask on Friday:

1. What to do with strangely formated regions: Some of them hold not a set of points, but a set of sets of points. In particular, those are 220, 388, 1931, 2627. Do we just treat all points the same or there is some trick?

Yes 2. What to do with the rides that start and end at the same geo point? Am I right to assume we just ignore them?

Yes. 3. Question 9: the book says that if triangle inequality holds TSP is equivalent to shortest Euclidean path. Without this assumption (as in the uber data), can we assume them to be the same? Won't this affect upper bound of $\frac{approx\ TSP}{opt\ TSP} = 2$?

It holds for high percentage of triangles, and others are basically placed on the line. SO we can say that the dercvation approximately holds. 4. Can't find the algorithm implemented in any package. Do you have any advice?

Implement yourself. 5. We have created Delaunay triangulation mesh, and we have orginal graph. Which one should we use in question, to find flow? It seems that we need the original graph, but the question is formulated ambigously so I want to double check.

No. Use mesh G_Δ. Transfer times where possible from original graph. Where impossible, estiamate using shortest time. 6. Same for question 14.

Same as question

Useful links:

- plotting: https://towardsdatascience.com/easy-steps-to-plot-geographic-data-on-a-map-python-11217859a2db
- plotting: https://www.qgis.org/en/site/forusers/index.html
- plotting: https://gis.stackexchange.com/questions/247871/convert-gps-coordinates-to-web-mercator-epsg3857-using-python-pyproj
- plotting: https://pyproj4.github.io/pyproj/stable/gotchas.html#upgrading-to-pyproj-2-from-pyproj-1

2 Import libs

[594]:
```
import os
import time
import json
import cairo
import random
import itertools

import numpy as np
import pandas as pd
import igraph as ig
import contextily as cx
import matplotlib
import matplotlib.cm as cm
import matplotlib.pyplot as plt
import matplotlib.patches as patches
from matplotlib.collections import LineCollection
from pyproj import Transformer
from scipy.spatial import Delaunay
```

3 Import graph

load data from csv and preprocess it

TA confirmed that we can discard loops that lead from a vertex to itself

[2]:
```
TRAVEL_TIMES = os.path.join("uber data",␣
 ↪"los_angeles-censustracts-2019-4-All-MonthlyAggregate.csv")

# import pandas dataframe
time_data = pd.read_csv(TRAVEL_TIMES)
# choose only the data gathered during december
time_data = time_data.loc[time_data.month == 12]
# get rid of everything except start/end points and meat travel time
time_data = time_data.drop(columns=["month",
                                    "standard_deviation_travel_time",
                                    "geometric_mean_travel_time",
                                    "geometric_standard_deviation_travel_time"])
# reorder dataframe in a way that source id is always smaller than destination␣
 ↪id
time_data[['sourceid','dstid']] = time_data[['sourceid','dstid']].
 ↪mask(time_data.sourceid > time_data.dstid,
                                                                  ␣
 ↪time_data[['dstid','sourceid']].values)
# calculate means of repeating rows
time_data = time_data.groupby(by=['sourceid','dstid']).mean().reset_index()
```

```
# delete all rows that go to themselves
time_data = time_data.loc[time_data.sourceid != time_data.dstid]
```

save the result

[3]:
```
TRAVEL_TIMES_CLEAR = os.path.join("uber data", "clear.csv")
# save the data if needed
time_data.to_csv(TRAVEL_TIMES_CLEAR, index=False)
```

get mean travel time for reference

[522]:
```
print(time_data["mean_travel_time"].mean())
```

```
1496.788687707823
```

create graph

[523]:
```
uber = ig.Graph.DataFrame(time_data, directed=False)
```

check if graph is connected, and if not, select GCC

[524]:
```
print(uber.is_connected())
```

```
True
```

load geo locations from json

[7]:
```
GEO_BOUNDARIES = os.path.join('uber data', "los_angeles_censustracts.json")

# load geographical data of node id-s
with open(GEO_BOUNDARIES) as f:
    geo = json.load(f)


def find_centroid(points):
    # this function finds centroid of a set of points
    # init center coordinates
    center = [0, 0]
    # check if there are nested lists
    if not type(points[0][0]) is float:
        # if there is only one list nested
        if len(points) == 1:
            # just access it
            points = points[0]
        # if not
        else:
            # flatten the whole structure
            points = list(itertools.chain.from_iterable(points))
    # sweep through all coordinates
    for j in range(len(points)):
```

```
        # update coordinates with corresponding values
        center[0] += points[j][0]
        center[1] += points[j][1]
    # normilize coordinates
    center[0] /= len(points)
    center[1] /= len(points)

    return center


# init a dictionary to hold coordinates
centroids = {}

# sweep through the file
for i in range(len(geo["features"])):
    try:
        centroids[int(geo['features'][i]['properties']['MOVEMENT_ID'])] = \
            find_centroid(geo['features'][i]['geometry']['coordinates'][0])
    except TypeError:
        print("error in", i)
        pass
```

Either append or choose 1

Sourceid and Movementid are te same

Assign a property to each vertex

[525]:
```
# sweep through all vertices
for i in range(len(uber.vs)):
    # get the name of the vertex
    vertex_name = int(uber.vs[i]["name"])
    # assign a centroid property to the vertex
    uber.vs[i]["centroid"] = centroids[vertex_name]
    # get the string of the vertex and store it in a separate attribute for␣
 ↪question 9; to
    # remove ambiguity, add "a" letter to the string
    uber.vs[i]["vetrex_name_string"] = str(uber.vs[i]["name"]) + 'a'
```

[526]:
```
print(uber.vs[0])
print(uber.vs[1])
```

```
igraph.Vertex(<igraph.Graph object at 0x7fcdf98f7c70>, 0, {'name': 1.0,
'centroid': [-118.12053321311474, 34.103095573770496], 'vetrex_name_string':
'1.0a'})
igraph.Vertex(<igraph.Graph object at 0x7fcdf98f7c70>, 1, {'name': 2.0,
'centroid': [-118.13785063157897, 34.09645121052631], 'vetrex_name_string':
'2.0a'})
```

[527]:
```
uber.vs.find(vetrex_name_string='1.0a')
```

[527]:
```
igraph.Vertex(<igraph.Graph object at 0x7fcdf98f7c70>, 0, {'name': 1.0,
'centroid': [-118.12053321311474, 34.103095573770496], 'vetrex_name_string':
'1.0a'})
```

Assign a weight to each edge

[528]:
```
# sweep through all edges
for i in range(len(uber.es)):
    # assign a weight property to the edge
    uber.es[i]['weight'] = uber.es[i]['mean_travel_time']
```

4 Question 6

Report the number of nodes and edges in G.

[529]:
```
print("Number of nodes:", uber.vcount())
print("Number of edges:", uber.ecount())
print("Total number of locations", len(centroids))
```

```
Number of nodes: 2649
Number of edges: 1003858
Total number of locations 2716
```

Plot G

[13]:
```
def plot_graph(g, path, margin = 0.1):
    """
    a function that allows to plot specified graph over a map
    Args:
        g (igraph.Grah):        graph to plot. Each vertex needs to have␣
↪attributes:
                                    centroid    latitude and longditude of the␣
↪point
                                    size        a multipler of the preset␣
↪diameter; size = 1 corresponds to
                                                1 / 10000 of the figure size
                                    color       color of the vertex
                                Each edge needs to have attributes:
                                    width       a multipler of the preset width;
↪ width = 1 corresponds to
                                                1 / 2000000 of the figure size
                                    color       color of the edge
        path (str):             path to save graph to
        margin (float):         how much of additional map space is added to␣
↪the bounding box of vertex
```

```
                                coordinates
    """
    # init output figure
    fig, ax = plt.subplots(figsize=(17, 17))

    # init coordinates of the snapshot
    west, south, east, north = centroids[1][0], centroids[1][1],␣
↪centroids[1][0], centroids[1][1]
    # sweep through all centroids to crorect coordinates of the bounding box
    for key in centroids.keys():
        if centroids[key][0] < west:
            west = centroids[key][0]
        if centroids[key][0] > east:
            east = centroids[key][0]
        if centroids[key][1] > north:
            north = centroids[key][1]
        if centroids[key][1] < south:
            south = centroids[key][1]
    # get the center of snapshot
    x_center_degrees = (west - east) / 2 + east
    y_center_degrees = (north - south) / 2 + south
    # choose the higher dimensions, and make the dimension of the other␣
↪coordinate match with it, with 5% addition
    # to both dimesions
    margin = 0.1
    # choose an arbitrary dimension
    dim = abs(west - east)
    # check if it is the largest, and if it is not, reset it
    if abs(north - south) > dim: dim = abs(north - south)
    # reset the coordinates
    west = x_center_degrees - dim * (1 + margin) / 2
    east = x_center_degrees + dim * (1 + margin) / 2
    north = y_center_degrees + dim * (1 + margin) / 2
    south = y_center_degrees - dim * (1 + margin) / 2

    # download a snapshot of the map
    ghent_img, ghent_ext = cx.bounds2img(west,
                                         south,
                                         east,
                                         north,
                                         ll=True,
                                         source=cx.providers.Stamen.Terrain)
    # add it to the figure
    ax.imshow(ghent_img, extent=ghent_ext)

    # at this point a picture is guaranteed to have "approximately" square form;
↪ if precision is not sufficient
```

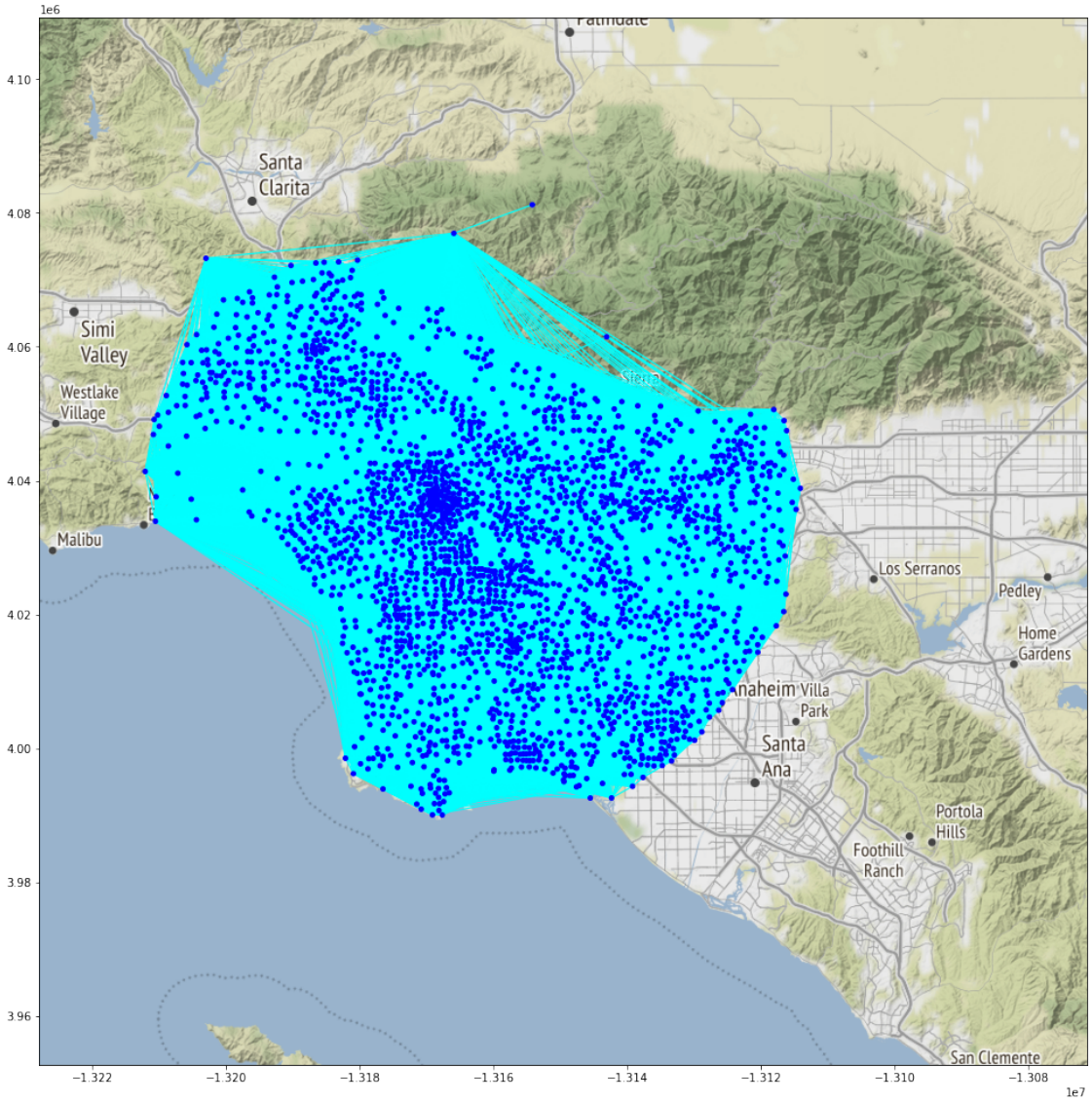
```
    # this can be relaced by separate x and y scales
    scale_contextily = abs(ghent_ext[0] - ghent_ext[1])
    # calculate unit diameter of points and width of lines
    unit_vertex = scale_contextily / 10000
    unit_edge = scale_contextily / 2000000
    # initialize transformer to convert coordinates to web merkator
    transformer = Transformer.from_crs("epsg:4326", "epsg:3857")

    # init arrays for vertices' coordinates, colors and sizes
    vertex_xs = np.zeros((g.vcount()))
    vertex_ys = np.zeros((g.vcount()))
    vertex_colors = np.zeros((g.vcount(), 3))
    vertex_sizes = np.zeros((g.vcount()))
    # init array for edges coordinates: (num_lines, num_point_in_line,␣
↪dots_dim), colors and sizes
    edges_coords = np.zeros((g.ecount(), 2, 2))
    edges_colors = np.zeros((g.ecount(), 3))
    edges_sizes = np.zeros((g.ecount()))
    # for each vertex
    for i in range(g.vcount()):
        # convert coordinates to web merkator
        g.vs[i]['x'], g.vs[i]['y'] = transformer.transform(g.
↪vs[i]["centroid"][1], g.vs[i]["centroid"][0])
        # store coordinates in the array for plotting
        vertex_xs[i] = g.vs[i]['x']
        vertex_ys[i] = g.vs[i]['y']
        # store rgb colors for plotting, if color is present in attributes
        if "color" in g.vs[i].attributes().keys():
            vertex_colors[i] = g.vs[i]["color"]
        else:
            # else set default blue
            vertex_colors[i] = (0, 0, 1)
        # store relative sizes for plotting, if size is present in attributes
        if "size" in g.vs[i].attributes().keys():
            vertex_sizes[i] = g.vs[i]["size"]
        else:
            # else set default 1
            vertex_sizes[i] = 1
    # for each edge
    for i in range(g.ecount()):
        # get the edge
        ed = g.es[i]
        # get the id-s of a start and end points
        v_source_id, v_target_id = ed.source, ed.target
        # fill the coordinates for the source
        edges_coords[i, 0, 0], edges_coords[i, 0, 1] = g.vs[v_source_id]["x"],␣
↪g.vs[v_source_id]["y"]
```

```
        # fill the coordinates for the target
        edges_coords[i, 1, 0], edges_coords[i, 1, 1] = g.vs[v_target_id]["x"],␣
↪g.vs[v_target_id]["y"]
        # store rgb colors for plotting, if color is present in attributes
        if "color" in g.es[i].attributes().keys():
            edges_colors[i] = g.es[i]["color"]
        else:
            # else set default cyan
            edges_colors[i] = (0, 1, 1)
        # store relative sizes for plotting, if size is present in attributes
        if "size" in g.es[i].attributes().keys():
            edges_sizes[i] = g.es[i]["size"]
        else:
            # else set default 1
            edges_sizes[i] = 1


    # Make a sequence of (x, y) pairs.
    line_segments = LineCollection(edges_coords,
                                   linewidths=edges_sizes * unit_edge,
                                   linestyles='solid',
                                   colors=edges_colors)
    ax.add_collection(line_segments)
    # create a scatter plot for vertices
    ax.scatter(vertex_xs,
               vertex_ys,
               s=vertex_sizes * unit_vertex,
               c=vertex_colors,
               marker='o',
               zorder=2)
    # save the figure
    plt.savefig(path, dpi=300)
```

[289]:
```
for i in range(uber.ecount()):
    uber.es[i]["size"] = 15
```

[290]:
```
start = time.time()
plot_graph(uber, "results/q6.png")
end = time.time()
print(end - start)
```

```
80.17182445526123
```



As we can see, almost each location was either a destination or initial point at least once.

5 Question 7

Build a minimum spanning tree (MST) of graph G. Report the street addresses near the two endpoints (the centroid locations) of a few edges. Are the results intuitive?

[530]:
```
uber_mst = uber.spanning_tree(weights=uber.es['mean_travel_time'])
```

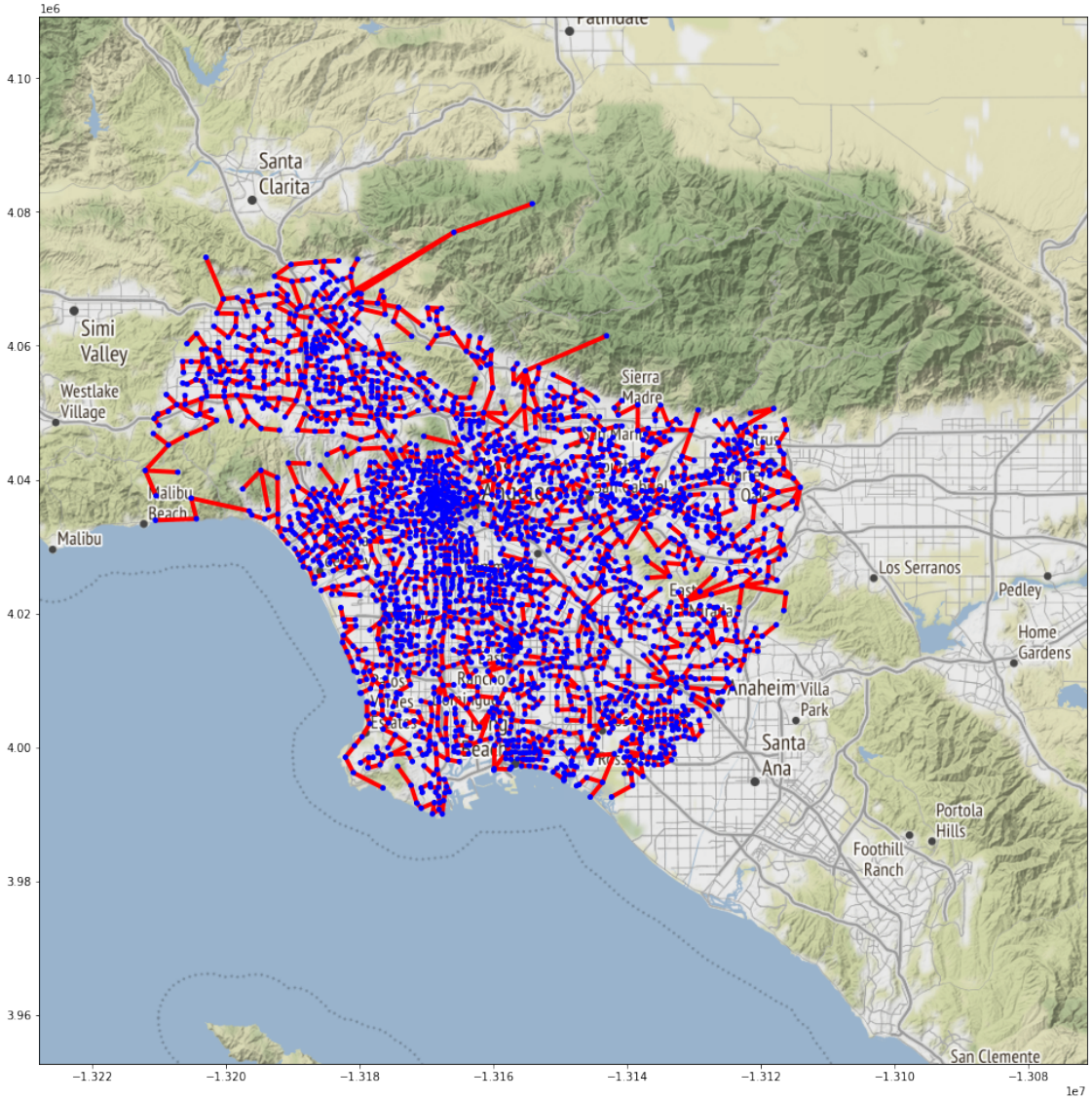
Plot MST

[16]:
```
for i in range(uber_mst.vcount()):
    uber_mst.vs[i]["color"] = (0, 0, 1)
```

```
    uber_mst.vs[i]["size"] = 1

for i in range(uber_mst.ecount()):
    uber_mst.es[i]["color"] = (1, 0, 0)
    uber_mst.es[i]["size"] = 50
```

[17]:
```
start = time.time()
plot_graph(uber_mst, "results/q7.png")
end = time.time()
print(end - start)
```

```
2.423133373260498
```



To find street addresses, chose some edge i, get vertices that are connected by this edge a and b, retreive their coordinates, and past in google such that link looks like

https://www.google.com/maps/@x,y,Mz

where x is a second coordinate of the centroid, y is the first, M is a multiplication factor for scale. Choose $M = 20$ for convenience. Click 2 times on the street closest to the center of the screen. This will be your street.

[532]:
```
def retreive_coord(idx, graph):
    # retreive an edge with this index
    e = graph.es[idx]
    # find igraph id-s (not uber id-s!) of incident vertices
    v1_id, v2_id = e.source, e.target
    # retreive vertices
    v1, v2 = graph.vs[v1_id], graph.vs[v2_id]
    # print their centroid in REVERSED order (see explanation in text cell␣
 ↪above)
    print("first point's uber id", v1["name"], "and coordinates",␣
 ↪v1["centroid"][1], v1["centroid"][0])
    print("google link:")
    print("https://www.google.com/maps/@{x},{y},20z".format(x =␣
 ↪v1["centroid"][1], y=v1["centroid"][0]))
    print("second point's uber id", v2["name"], "and coordinates",␣
 ↪v2["centroid"][1], v2["centroid"][0])
    print("google link:")
    print("https://www.google.com/maps/@{x},{y},20z".format(x =␣
 ↪v2["centroid"][1], y=v2["centroid"][0]))
```

[535]:
```
# choose index to look into
index = 0
# get links
retreive_coord(index, uber_mst)
```

```
first point's uber id 1.0 and coordinates 34.103095573770496 -118.12053321311474
google link:
https://www.google.com/maps/@34.103095573770496,-118.12053321311474,20z
second point's uber id 3.0 and coordinates 34.09626386363636 -118.13138209090911
google link:
https://www.google.com/maps/@34.09626386363636,-118.13138209090911,20z
```

[536]:
```
# choose index to look into
index = 25
# check coordinates
retreive_coord(index, uber_mst)
```

```
first point's uber id 21.0 and coordinates 34.06994802040815 -118.15898950000003
google link:
https://www.google.com/maps/@34.06994802040815,-118.15898950000003,20z
```

```
second point's uber id 22.0 and coordinates 34.064586108910895
-118.1490713069307
google link:
https://www.google.com/maps/@34.064586108910895,-118.1490713069307,20z
```

[537]:
```
# choose index to look into
index = 100
# check coordinates
retreive_coord(index, uber_mst)
```

```
first point's uber id 78.0 and coordinates 33.98065181395349 -118.19460306976742
google link:
https://www.google.com/maps/@33.98065181395349,-118.19460306976742,20z
second point's uber id 79.0 and coordinates 33.97718283333333
-118.19844175925924
google link:
https://www.google.com/maps/@33.97718283333333,-118.19844175925924,20z
```

6 Question 8

Determine what percentage of triangles in the graph (sets of 3 points on the map) satisfy the triangle inequality. You do not need to inspect all triangles, you can just estimate by random sampling of 1000 triangles.

Let's find the percentage by using the definition of the triangle is "3 vertices directly connected by edges". If we use this definition, a triangle is basically a clique (fully connected susbset) of size of 3.

[292]:
```
def check_type_1_inequality(graph, vertex_ids):
    # this function checks whether the triangle inequality works for specified␣
 ↪triangle
    # vertex_id-s are assumed to be connected
    # collect id-s
    id1, id2, id3 = vertex_ids
    # get id-s of the edges connecting them
    edges = graph.get_eids(pairs=[(id1, id2), (id2, id3), (id3, id1)],␣
 ↪directed=False)
    # get lengths
    side_a = graph.es[edges[0]]['mean_travel_time']
    side_b = graph.es[edges[1]]['mean_travel_time']
    side_c = graph.es[edges[2]]['mean_travel_time']
    # check if triangle inequality is satisfied for all sides
    satisfy = (side_a + side_b > side_c) and (side_b + side_c > side_a) and␣
 ↪(side_c + side_a > side_b)
    return satisfy
```

[345]:
```
# set up the number of triangles to test on
SAMPLE_SIZE = 1000
```

```
# select the size of the subgraph we will consider for efficiency
buffer_size = 40
# select offset, i.e. how strongly we shift the window of size "buffer_size" of␣
 ↪vertices we will look into
offset = np.random.randint(0, high=uber.vcount() - buffer_size - 1)
# get a list of vertices to select for the subgraph
sub_vertices = list(offset + np.arange(buffer_size))
# derive the subgraph
triangles_uber = uber.subgraph(sub_vertices)
# find all triangles in the subgraph
traingles_type_1 = triangles_uber.cliques(min=3, max=3)
print("total number of triangles in the window:", len(traingles_type_1))
if len(traingles_type_1) < SAMPLE_SIZE:
    print("Not enough trianlges to work with! Select larger beuffer size")
else:
    # randomly sample enough triangles, remove all others
    traingles_type_1 = random.sample(traingles_type_1, SAMPLE_SIZE)
    print("Sampled", len(traingles_type_1), "for evaluation")
    # init counter to check how many traingles satisfy the equality
    good_triangles_counter = 0
    # sweer through seleted triangles and check if they are good
    for i in range(len(traingles_type_1)):
        if check_type_1_inequality(triangles_uber, traingles_type_1[i]):
            good_triangles_counter += 1
    # normilize and print the percentage of good triangles
    good_triangles_counter /= SAMPLE_SIZE
    print(good_triangles_counter)
```

```
total number of triangles in the window: 9880
Sampled 1000 for evaluation
0.923
```

7 Question 9

Now, we want to find an approximation solution for the traveling salesman problem (TSP) on G. Apply the 1-approximate algorithm described in the class. Inspect the sequence of street addresses visited on the map and see if the results are intuitive. Find an upper bound on the empirical performance of the approximate algorithm:

Obviously, the upper bound on approximate solution is 2 lengths of MST

The algorithm to find an approximate solution for travelling salesman problem:

1. Find minimum spanning tree
2. Create a directed graph from it
3. Find an Euler cycle in multi-graph

- Choose a vertex, and follow any of its output edges at random until you return back

- Sweep through all vertices of the obtained graph. If there exist a vertex v that is connected to an output edge that does not belong to the obtained graph, repeat in cycle
- until a vertex described above exist, start another trial from this vertext that can use only the edges that have not been visited before. Join new path with already existing path.

[250]:
```python
def get_euleriean_cycle(start_id, graph):
    # initialize current vertex id
    v_current_id = start_id
    # initialize vertex and edge paths
    v_path = [v_current_id]
    e_path = []

    # get unvisited edges connected to it
    edges = graph.vs[v_current_id].out_edges()
    # find their id-s
    edge_ids = [edge.index for edge in edges if edge["visited"] is False]
    # choose a random edge
    next_edge_id = random.choice(edge_ids)
    # mark it as visited
    graph.es[next_edge_id]["visited"] = True
    # record it into memory
    e_path.append(next_edge_id)
    # get the next pont
    v_current_id = graph.es[next_edge_id].target
    # record it into memory
    v_path.append(v_current_id)

    # until the end point is not the same as the start point
    while v_current_id != start_id:
        # get unvisited edges connected to it
        edges = graph.vs[v_current_id].out_edges()
        # find their id-s
        edge_ids = [edge.index for edge in edges if edge["visited"] is False]
        # choose a random edge
        next_edge_id = random.choice(edge_ids)
        # mark it as visited
        graph.es[next_edge_id]["visited"] = True
        # record it into memory
        e_path.append(next_edge_id)
        # get the next pont
        v_current_id = graph.es[next_edge_id].target
        # record it into memory
        v_path.append(v_current_id)

    return v_path, e_path
```

[274]:
```
# copy minimum spanning tree
uber_mst_directed = uber_mst.copy()
# convert graph to directed
uber_mst_directed.to_directed(mode=True)
# add a "visited" property to each edge of the graph. By default, it is false
for i in range(uber_mst_directed.ecount()):
    uber_mst_directed.es[i]["visited"] = False


# choose a random vertex to start from, for example, zero
v_start_id = 0

# get some path
v_path, e_path = get_euleriean_cycle(v_start_id, uber_mst_directed)

# now we check for edges that have not yet been visited that appear on our path
# initialize an index of the vertex we are currently surveying for unvisited␣
 ↪edges
survey_id = 0

# until we have not sweeped through all vertices on our path
while survey_id < len(v_path) - 1:
    # get unvisited edges connected to it
    edges = uber_mst_directed.vs[v_path[survey_id]].out_edges()
    # find their id-s
    edge_ids = [edge.index for edge in edges if edge["visited"] is False]
    # if there is an edge that satisfies our criteria
    if len(edge_ids) > 0:
        # get one more eulerian cycle
        v_sub_path, e_sub_path = get_euleriean_cycle(v_path[survey_id],␣
 ↪uber_mst_directed)
        # divide the list in 2 sublists
        before_v, after_v = v_path[0:survey_id], v_path[survey_id + 1:]
        before_e, after_e = e_path[0:survey_id], e_path[survey_id:]
        # concatenate them with new input part
        v_path = []
        v_path.extend(before_v)
        v_path.extend(v_sub_path)
        v_path.extend(after_v)
        e_path = []
        e_path.extend(before_e)
        e_path.extend(e_sub_path)
        e_path.extend(after_e)

    # move to the next veretex
    survey_id += 1
```

[281]:
```
correct = True
for i in range(uber_mst_directed.ecount()):
    if uber_mst_directed.es[i].index not in e_path:
        print(uber_mst_directed.es[i].index)
        correct = False

print(correct)
```

```
True
```

[541]:
```
offset = 20

for i in range(4):
    print(e_path[offset + i])
    retreive_coord(e_path[offset + i], uber_mst_directed)
```

```
3813
first point's uber id 1048.0 and coordinates 34.12479304651163
-118.16984624418603
google link:
https://www.google.com/maps/@34.12479304651163,-118.16984624418603,20z
second point's uber id 1046.0 and coordinates 34.12044833333333
-118.1829551884058
google link:
https://www.google.com/maps/@34.12044833333333,-118.1829551884058,20z
1166
first point's uber id 1046.0 and coordinates 34.12044833333333
-118.1829551884058
google link:
https://www.google.com/maps/@34.12044833333333,-118.1829551884058,20z
second point's uber id 1049.0 and coordinates 34.11932847499999
-118.18758254999997
google link:
https://www.google.com/maps/@34.11932847499999,-118.18758254999997,20z
1171
first point's uber id 1049.0 and coordinates 34.11932847499999
-118.18758254999997
google link:
https://www.google.com/maps/@34.11932847499999,-118.18758254999997,20z
second point's uber id 1058.0 and coordinates 34.11539226086957
-118.18893304347824
google link:
https://www.google.com/maps/@34.11539226086957,-118.18893304347824,20z
3827
first point's uber id 1058.0 and coordinates 34.11539226086957
-118.18893304347824
google link:
https://www.google.com/maps/@34.11539226086957,-118.18893304347824,20z
```

```
second point's uber id 1057.0 and coordinates 34.11402644444444
-118.19431949999999
google link:
https://www.google.com/maps/@34.11402644444444,-118.19431949999999,20z
```

8 Question 10

Plot the trajectory that Santa has to travel!

[596]:
```
def plot_trajectory(g, trajectory, path, margin = 0.1):
    """
    a function that allows to plot specified graph over a map
    Args:
        g (igraph.Grah):        graph to plot. Each vertex needs to have␣
↪attributes:
                                    centroid    latitude and longditude of the␣
↪point
                                    size        a multipler of the preset␣
↪diameter; size = 1 corresponds to
                                                1 / 10000 of the figure size
                                    color       color of the vertex
                                Each edge needs to have attributes:
                                    width       a multipler of the preset width;
↪ width = 1 corresponds to
                                                1 / 2000000 of the figure size
                                    color       color of the edge
        trajectory (list):      list with edges that are included in the␣
↪trajectory
        path (str):             path to save graph to
        margin (float):         how much of additional map space is added to␣
↪the bounding box of vertex
                                coordinates
    """
    # init output figure
    fig, ax = plt.subplots(figsize=(17, 17))

    # init coordinates of the snapshot
    west, south, east, north = centroids[1][0], centroids[1][1],␣
↪centroids[1][0], centroids[1][1]
    # sweep through all centroids to crorect coordinates of the bounding box
    for key in centroids.keys():
        if centroids[key][0] < west:
            west = centroids[key][0]
        if centroids[key][0] > east:
            east = centroids[key][0]
        if centroids[key][1] > north:
            north = centroids[key][1]
```

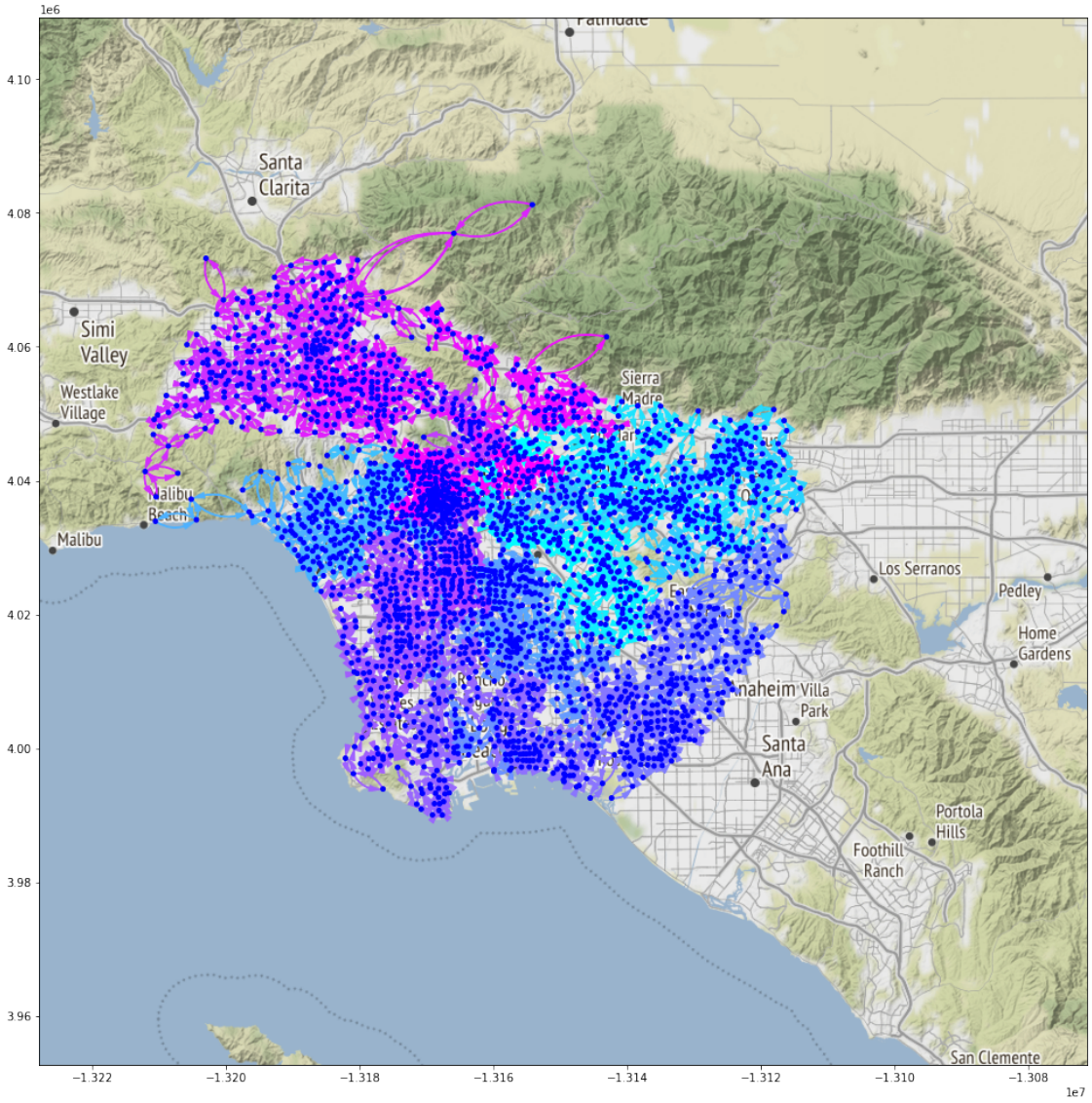
```
        if centroids[key][1] < south:
            south = centroids[key][1]
    # get the center of snapshot
    x_center_degrees = (west - east) / 2 + east
    y_center_degrees = (north - south) / 2 + south
    # choose the higher dimensions, and make the dimension of the other␣
↪coordinate match with it, with 5% addition
    # to both dimesions
    margin = 0.1
    # choose an arbitrary dimension
    dim = abs(west - east)
    # check if it is the largest, and if it is not, reset it
    if abs(north - south) > dim: dim = abs(north - south)
    # reset the coordinates
    west = x_center_degrees - dim * (1 + margin) / 2
    east = x_center_degrees + dim * (1 + margin) / 2
    north = y_center_degrees + dim * (1 + margin) / 2
    south = y_center_degrees - dim * (1 + margin) / 2

    # download a snapshot of the map
    ghent_img, ghent_ext = cx.bounds2img(west,
                                         south,
                                         east,
                                         north,
                                         ll=True,
                                         source=cx.providers.Stamen.Terrain)
    # add it to the figure
    ax.imshow(ghent_img, extent=ghent_ext)

    # at this point a picture is guaranteed to have "approximately" square form;
↪ if precision is not sufficient
    # this can be relaced by separate x and y scales
    scale_contextily = abs(ghent_ext[0] - ghent_ext[1])
    # calculate unit diameter of points and width of lines
    unit_vertex = scale_contextily / 10000
    unit_edge = scale_contextily / 2000000
    # initialize transformer to convert coordinates to web merkator
    transformer = Transformer.from_crs("epsg:4326", "epsg:3857")

    # init arrays for vertices' coordinates, colors and sizes
    vertex_xs = np.zeros((g.vcount()))
    vertex_ys = np.zeros((g.vcount()))
    vertex_colors = np.zeros((g.vcount(), 3))
    vertex_sizes = np.zeros((g.vcount()))
    # init array for edges coordinates: (num_lines, num_point_in_line,␣
↪dots_dim), colors and sizes
    edges_coords = np.zeros((g.ecount(), 2, 2))
```

```
    edges_colors = np.zeros((g.ecount(), 3))
    edges_sizes = np.zeros((g.ecount()))
    # for each vertex
    for i in range(g.vcount()):
        # convert coordinates to web merkator
        g.vs[i]['x'], g.vs[i]['y'] = transformer.transform(g.
↪vs[i]["centroid"][1], g.vs[i]["centroid"][0])
        # store coordinates in the array for plotting
        vertex_xs[i] = g.vs[i]['x']
        vertex_ys[i] = g.vs[i]['y']
        # store rgb colors for plotting, if color is present in attributes
        if "color" in g.vs[i].attributes().keys():
            vertex_colors[i] = g.vs[i]["color"]
        else:
            # else set default blue
            vertex_colors[i] = (0, 0, 1)
        # store relative sizes for plotting, if size is present in attributes
        if "size" in g.vs[i].attributes().keys():
            vertex_sizes[i] = g.vs[i]["size"]
        else:
            # else set default 1
            vertex_sizes[i] = 0.1

    style = "Simple, tail_width=0.75, head_width=5, head_length=10"

    minima = 0
    maxima = len(trajectory)

    norm = matplotlib.colors.Normalize(vmin=minima, vmax=maxima, clip=True)
    mapper = cm.ScalarMappable(norm=norm, cmap="cool")

    # for edge in the trajectory
    for i in range(len(trajectory)):
        # get the edge
        ed = g.es[trajectory[i]]
        # get the id-s of a start and end points
        v_source_id, v_target_id = ed.source, ed.target
        # fill the coordinates for the source
        edges_coords[i, 0, 0], edges_coords[i, 0, 1] = g.vs[v_source_id]["x"],␣
↪g.vs[v_source_id]["y"]
        # fill the coordinates for the target
        edges_coords[i, 1, 0], edges_coords[i, 1, 1] = g.vs[v_target_id]["x"],␣
↪g.vs[v_target_id]["y"]
        # store rgb colors for plotting, if color is present in attributes
        if "color" in g.es[i].attributes().keys():
            edges_colors[i] = g.es[i]["color"]
        else:
```

```
            # else set default cyan
            edges_colors[i] = (0, 1, 1)
        # store relative sizes for plotting, if size is present in attributes
        if "size" in g.es[i].attributes().keys():
            edges_sizes[i] = g.es[i]["size"]
        else:
            # else set default 1
            edges_sizes[i] = 1

        # get a color
        col = mapper.to_rgba(i)

        kw = dict(arrowstyle=style, color=col)
        # create a new path
        arrow = patches.FancyArrowPatch((edges_coords[i, 0, 0], edges_coords[i,␣
↪0, 1]),
                                        (edges_coords[i, 1, 0], edges_coords[i,␣
↪1, 1]),
                                        connectionstyle="arc3,rad=.35", **kw)
        ax.add_patch(arrow)

    # create a scatter plot for vertices
    ax.scatter(vertex_xs,
               vertex_ys,
               s=vertex_sizes * unit_vertex,
               c=vertex_colors,
               marker='o',
               zorder=2)
    # save the figure
    plt.savefig(path, dpi=300)
```

[598]:
```
start = time.time()
plot_trajectory(uber_mst_directed, e_path, "results/q10.png")
end = time.time()
print(end - start)
```

```
24.385759592056274
```



9 Question 11

Plot the road mesh that you obtain and explain the result. Create a graph G_Δ whose nodes are different locations and its edges are produced by triangulation.

[18]:
```
# initialize a matrix to store coordinates
coord_triang = np.zeros((uber.vcount(), 2))
# fill the coordinates from each vertex
for i in range(uber.vcount()):
    # remember that x is the second variable
    coord_triang[i, 0], coord_triang[i, 1] = uber.vs[i]["centroid"][1], uber.
↪vs[i]["centroid"][0]
```

```
# find triangluation
triang = Delaunay(coord_triang)
```

[19]:
```
# triang.simplices store the id-s of vertices that form triangles
print(triang.simplices)
```

```
[[2421 2419 2418]
 [ 983 2415 2419]
 [1699 1783 1860]
 …
 [ 199 2357  196]
 [ 196 2357 1696]
 [2357  198 1696]]
```

[20]:
```
# create a new graph as a copy of the original
uber_triangle = uber.copy()
# delete all edges of said traingle
uber_triangle.delete_edges(np.arange(uber_triangle.ecount()))
# verify that no edge from the original graph remained
print("deleted all edges from the graph. Number of edges remained is",␣
 ↪uber_triangle.ecount())
# init a numpy list with edges we need to add to the graph
triangle_edges = np.zeros((len(triang.simplices) * 3, 2))
# for each traingele in a mesh
for i in range(len(triang.simplices)):
    # record to add 3 new edges to the graph
    triangle_edges[i * 3]     = triang.simplices[i, 0], triang.simplices[i, 1]
    triangle_edges[i * 3 + 1] = triang.simplices[i, 1], triang.simplices[i, 2]
    triangle_edges[i * 3 + 2] = triang.simplices[i, 2], triang.simplices[i, 0]
# convert to appropriate type
triangle_edges = triangle_edges.astype(int)
# add said edges to the graph
uber_triangle.add_edges(triangle_edges)
# delete repeating edges
uber_triangle.simplify()
print("Now the graph has", uber_triangle.ecount(), "edges")
```

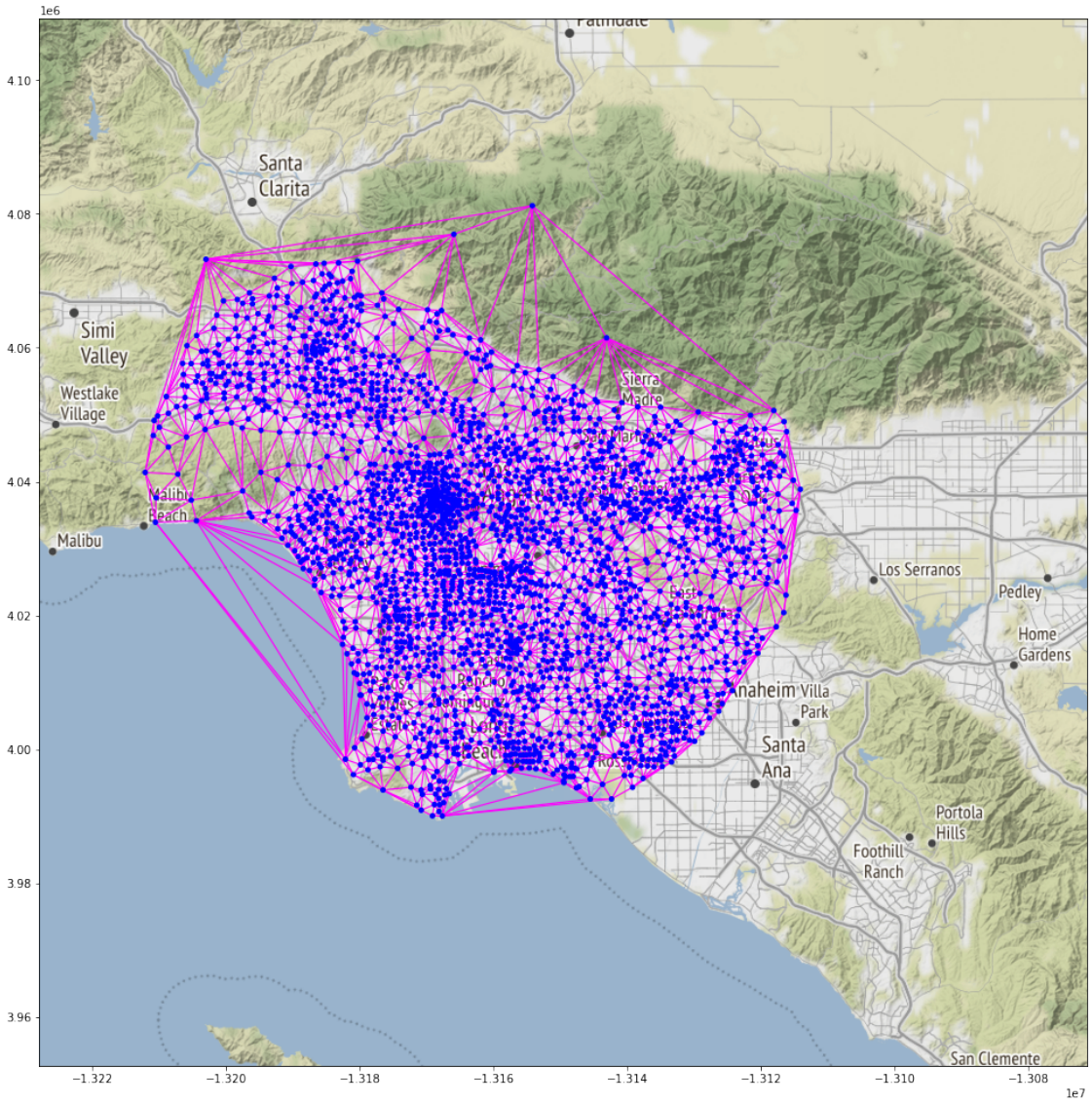
```
deleted all edges from the graph. Number of edges remained is 0
Now the graph has 7923 edges
```

[21]:
```
# increase size of the edges in the graph
for i in range(uber_triangle.ecount()):
    uber_triangle.es[i]["color"] = (1, 0, 1)
    uber_triangle.es[i]["size"] = 15
```

[546]:
```
start = time.time()
plot_graph(uber_triangle, "results/temp.png")
```

```
end = time.time()
print(end - start)
```

```
3.071528196334839
```



10 Question 12

Using simple math, calculate the traffic flow for each road in terms of cars/hour. Report your derivation.

To find flow between 2 connected points in the graph:

1. Calculate distance as a square root of sum of squares of delta coordinates.

2. Multiply that by 69 miles.
3. Divide by the average time of commuting - get an average speed.
4. Multiply that by 2 seconds to get the safety distance, than add it to the car length, to get the overall space that a single car takes.
5. Now multiply speed by an hour and 2, and divide by the total space, to get the total number of cars that can go per hour through the edge.
6. Create a new graph where each undirected edge is doubled, and directed edges have capacity as above.

TA advice: for edges that exist you transfer speed from graph G. For edges that do not exist you compute speed as shortest path.

[23]:
```
# first, add commute time for each of the edges
for i in range(uber_triangle.ecount()):
    # get the source and the destination id of the vertices connected by this␣
↪edge
    v_source_id, v_target_id = uber_triangle.es[i].source, uber_triangle.es[i].
↪target
    # get their names as strings
    v_source_name = uber_triangle.vs[v_source_id]["vetrex_name_string"]
    v_target_name = uber_triangle.vs[v_target_id]["vetrex_name_string"]
    # find them in the original graph
    v_source_id_uber = uber.vs.find(vetrex_name_string=v_source_name)
    v_target_id_uber = uber.vs.find(vetrex_name_string=v_target_name)
    # check if they are connected
    edge_id_uber = uber.get_eid(v_source_id_uber, v_target_id_uber,␣
↪directed=False, error=False)
    if edge_id_uber == -1:
        # if they are not, find the shortest path
        uber_triangle.es[i]["mean_travel_time"] = uber.
↪shortest_paths_dijkstra(source=v_source_id_uber,
                                                                    ␣
↪target=v_target_id_uber,
                                                                    ␣
↪weights="mean_travel_time",
                                                                    ␣
↪mode='all')[0][0]
    else:
        # if they are, assing a mean travel time
        uber_triangle.es[i]["mean_travel_time"] = uber.
↪es[edge_id_uber]["mean_travel_time"]
```

[24]:
```
# second, compute flow for each of the edges in ONE direction
for i in range(uber_triangle.ecount()):
    # get the source and the destination id of the vertices connected by this␣
↪edge
```

```
    v_source_id, v_target_id = uber_triangle.es[i].source, uber_triangle.es[i].
↪target
    # get the coordinates of source and destination in degrees
    x_source_deg, y_source_deg = uber_triangle.vs[v_source_id]["centroid"]
    x_target_deg, y_target_deg = uber_triangle.vs[v_target_id]["centroid"]
    # compute the distance in degrees between those 2 points
    dist_degrees = ((x_target_deg - x_source_deg) ** 2 + (y_target_deg -␣
↪y_source_deg) ** 2) ** (1/2)
    # rescale it into miles
    dist_miles = dist_degrees * 69
    # get average speed per second
    avg_speed = dist_miles / (uber_triangle.es[i]["mean_travel_time"])
    # get the total space a car takes by adding a safety distance with car␣
↪length
    total_space = avg_speed * 2 + 0.003
    # get the total number of cars that can flow in one direction per hour
    total_cars = avg_speed * 3600 * 2 / total_space
    # save it as a parameter
    uber_triangle.es[i]["max_flow"] = total_cars
```

[25]:
```
# thrid, create a copy of the current graph
uber_triangle_directed = uber_triangle.copy()
# convert graph to directed
uber_triangle_directed.to_directed(mode=True)
```

[26]:
```
# check that back-and-forth edges have the same properties
uber_triangle_directed.is_directed()
edge1 = uber_triangle_directed.get_eid(0, 2, directed=True, error=False)
edge2 = uber_triangle_directed.get_eid(2, 0, directed=True, error=False)
print(edge1)
print(edge2)
print(uber_triangle_directed.es[edge1])
print(uber_triangle_directed.es[edge2])
```

```
0
7923
igraph.Edge(<igraph.Graph object at 0x7fcdf98f7220>, 0, {'color': (1, 0, 1),
'size': 15, 'mean_travel_time': 129.76500000000001, 'max_flow':
2950.7397388860118})
igraph.Edge(<igraph.Graph object at 0x7fcdf98f7220>, 7923, {'color': (1, 0, 1),
'size': 15, 'mean_travel_time': 129.76500000000001, 'max_flow':
2950.7397388860118})
```

11 Question 13

Calculate the maximum number of cars that can commute per hour from Malibu to Long Beach. Also calculate the number of edge-disjoint paths between the two spots. Does the number of

edge-disjoint paths match what you see on your road map?

[27]:
```
def find_closest_vertex(cooridnate, ctrds):
    # returns the id of the region which centroid is closest to the specified␣
 ↪coordinate
    # choose some initial id at random
    opt_id = list(ctrds.keys())[0]
    # calculate the distance between the target and random id
    opt_distance = ((cooridnate[0] - ctrds[opt_id][1]) ** 2 + (cooridnate[1] -␣
 ↪ctrds[opt_id][0]) ** 2) ** (1 / 2)
    # sweep through all locations
    for key in ctrds.keys():
        # if new id is closer to the target
        if ((cooridnate[0] - ctrds[key][1]) ** 2 + (cooridnate[1] -␣
 ↪ctrds[key][0]) ** 2) ** (1 / 2) < opt_distance:
            # than select it as optimal
            opt_id = key
            opt_distance = ((cooridnate[0] - ctrds[key][1]) ** 2 +␣
 ↪(cooridnate[1] - ctrds[key][0]) ** 2) ** (1 / 2)
    return opt_id
```

[28]:
```
# find the id of a vertex with a coordinate closest to malibu
malibu_deg = [34.04, -118.56]
malibu_id = find_closest_vertex(malibu_deg, centroids)
print(malibu_id)
# find the string identificator of the vertex closest to malibu
malibu_vertex_string = str(malibu_id) + ".0a"
# find the vertex id
malibu_vertex = uber_triangle_directed.vs.
 ↪find(vetrex_name_string=malibu_vertex_string)
print(malibu_vertex)
malibu_vertex_id = malibu_vertex.index
print(malibu_vertex_id)
# find the id of a vertex with a coordinate closest to Long Beach
long_beach_deg = [33.77, -118.18]
long_beach_id = find_closest_vertex(long_beach_deg, centroids)
print(long_beach_id)
# find the string identificator of the vertex closest to Long Beach
long_beach_vertex_string = str(long_beach_id) + ".0a"
# find the vertex id
long_beach_vertex = uber_triangle_directed.vs.
 ↪find(vetrex_name_string=long_beach_vertex_string)
print(long_beach_vertex)
long_beach_vertex_id = long_beach_vertex.index
print(long_beach_vertex_id)
```

```
1523
igraph.Vertex(<igraph.Graph object at 0x7fcdf98f7220>, 1511, {'name': 1523.0,
'centroid': [-118.54625915050165, 34.048403046822735], 'vetrex_name_string':
'1523.0a', 'x': -13196509.204081291, 'y': 4035303.2259358317})
1511
672
igraph.Vertex(<igraph.Graph object at 0x7fcdf98f7220>, 660, {'name': 672.0,
'centroid': [-118.17865950000001, 33.771767700000005], 'vetrex_name_string':
'672.0a', 'x': -13155588.198171662, 'y': 3998197.020574805})
660
```

[62]:
```
# find the vertex with a coordinate closest to malibu
M_to_LB_flow = uber_triangle_directed.maxflow(malibu_vertex_id,
                                              long_beach_vertex_id,
                                              capacity='max_flow')
print(M_to_LB_flow)
```

```
Graph flow (4 edges, 1 vs 2648 vertices, value=11074.1447)
```

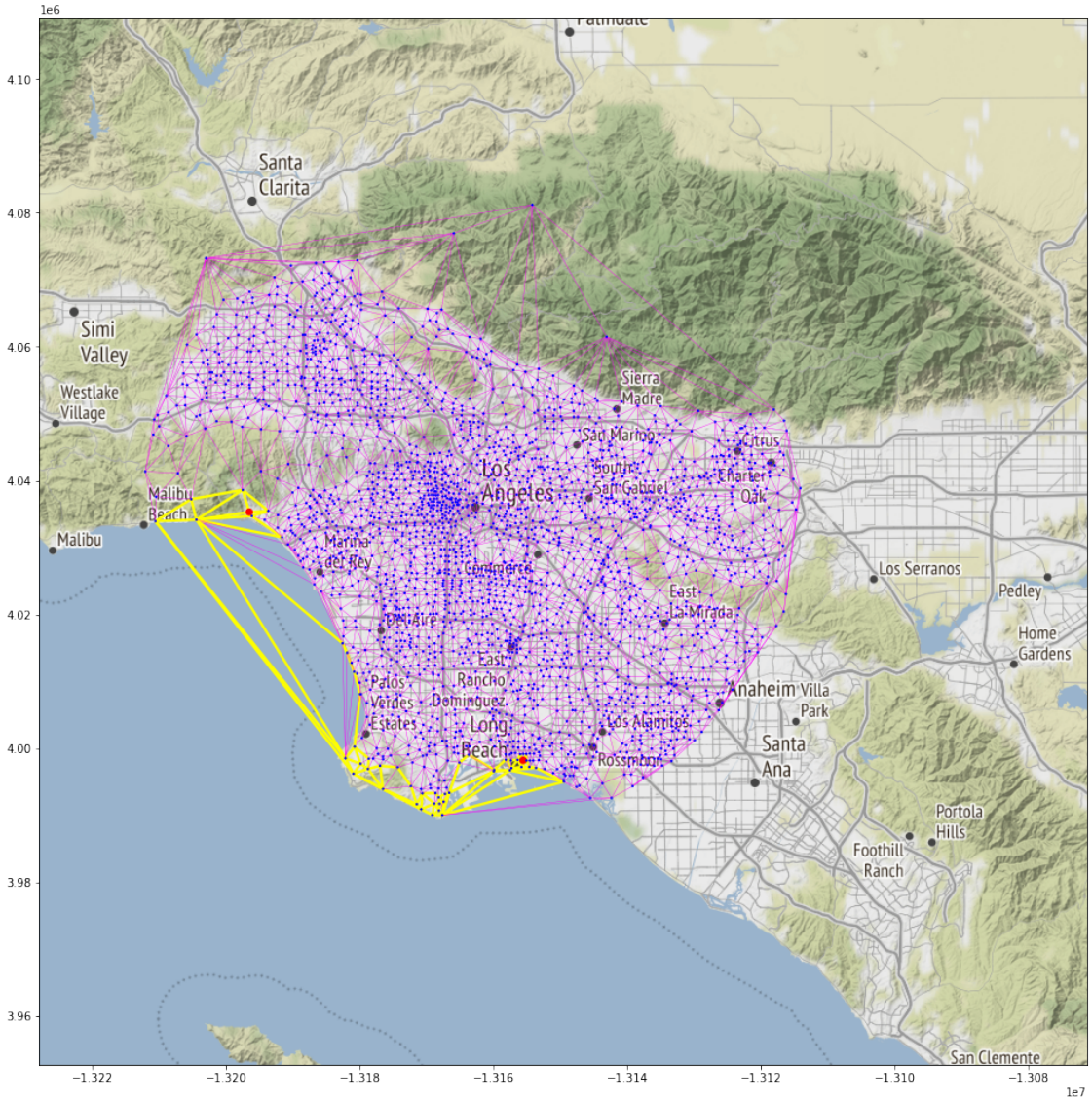
[73]:
```
# let's copy the original triangle graph
uber_triangle_plotting_copy = uber_triangle.copy()

# let's reset default vertex parameters for the triangle graph
for i in range(uber_triangle_plotting_copy.vcount()):
    uber_triangle_plotting_copy.vs[i]["color"] = (0, 0, 1)
    uber_triangle_plotting_copy.vs[i]["size"] = 0.1

# we need to specify source and destination points with big red dots
uber_triangle_plotting_copy.vs.
 ↪find(vetrex_name_string=malibu_vertex_string)["color"] = (1, 0, 0)
uber_triangle_plotting_copy.vs.
 ↪find(vetrex_name_string=malibu_vertex_string)["size"] = 2
uber_triangle_plotting_copy.vs.
 ↪find(vetrex_name_string=long_beach_vertex_string)["color"] = (1, 0, 0)
uber_triangle_plotting_copy.vs.
 ↪find(vetrex_name_string=long_beach_vertex_string)["size"] = 2

# increase size of the edges in the graph
for i in range(uber_triangle.ecount()):
    uber_triangle_plotting_copy.es[i]["color"] = (1, 0, 1)
    uber_triangle_plotting_copy.es[i]["size"] = 5
```

[75]:
```
# for each edge
for i in range(uber_triangle_directed.ecount()):
    # which has a positive flow
    if M_to_LB_flow.flow[i] > 1:
        # find the id-s of the vertices it connects
```

```
        source_id, target_id = uber_triangle_directed.es[i].source,␣
↪uber_triangle_directed.es[i].target
        # find their string names
        source_name = uber_triangle_directed.vs[source_id]["vetrex_name_string"]
        target_name = uber_triangle_directed.vs[target_id]["vetrex_name_string"]
        # find their id-s in plotting copy
        source_id = uber_triangle_plotting_copy.vs.
↪find(vetrex_name_string=source_name).index
        target_id = uber_triangle_plotting_copy.vs.
↪find(vetrex_name_string=target_name).index
        # find the edge they are connected by
        edge_id = uber_triangle_plotting_copy.get_eid(source_id, target_id,␣
↪directed=False, error=False)
        # find the eastpoint
        # if the id is valid
        if edge_id > 0:
            # color it and increase size
            uber_triangle_plotting_copy.es[edge_id]["color"] = (1, 1, 0)
            uber_triangle_plotting_copy.es[edge_id]["size"] = 30
        # else pop up an error
        else:
            print("no edge is found!")
```

[78]:
```
start = time.time()
plot_graph(uber_triangle_plotting_copy, "results/q13-maxflow.png")
end = time.time()
print(end - start)
```

```
2.6441140174865723
```



Obviously, those are not the correct paths. The reason is that from the plot of traingulation we can see that Malibu Beach and Long Beach are connected with roads that go through the ocean. These roads do not exist in real life, and flow caclualated through them is some composition of flows thriugh other vertices.

On the other hand, we can check the number of all edge-disjoint paths

[79]:
```
M_to_LB_paths = uber_triangle_directed.edge_disjoint_paths(source =␣
↪malibu_vertex_id,
                                                           target =␣
↪long_beach_vertex_id)
```

[81]:
```
print(M_to_LB_paths)
```

4

The number of path is 4, because the out degree of the source vertex is only 4:

[32]:
```
print(malibu_vertex.indegree())
```

4

Let's first collect the names of the edges in string disjoint paths. We will do that by creating a copy of the graph, and then iteratively finding a shortest then delete all edges from that path, until it becomes disconnected

[100]:
```
# let's init paths list
paths = []
# get a copy of traingle graph
uber_triangle_directed_copy = uber_triangle_directed.copy()
# for each path in edge disjoint paths:
for i in range(M_to_LB_paths):
    # append a vertex path to the list
    paths.append(uber_triangle_directed_copy.
 ↪get_shortest_paths(malibu_vertex_id,
                                                                 ␣
 ↪to=long_beach_vertex_id,
                                                                  mode='out',
                                                                 ␣
 ↪output='vpath')[0])
    # get an edge path
    edges = uber_triangle_directed_copy.get_shortest_paths(malibu_vertex_id,
                                                           ␣
 ↪to=long_beach_vertex_id,
                                                            mode='out',
                                                            output='epath')[0]
    # delete those edges
    uber_triangle_directed_copy.delete_edges(np.array(edges))

print(paths)
```

```
[[1511, 1700, 1699, 1860, 1658, 1660, 1661, 2373, 663, 660], [1511, 1510, 1700,
1783, 1860, 1861, 1658, 1657, 1662, 1661, 2375, 665, 664, 661, 660], [1511,
1512, 1700, 1701, 1783, 1859, 1882, 1861, 1863, 1862, 1643, 1642, 1641, 2423,
2373, 653, 654, 660], [1511, 1509, 1510, 2417, 1699, 1783, 1782, 1781, 1858,
1879, 2043, 2042, 568, 570, 1643, 1644, 1641, 1638, 684, 651, 652, 654, 656,
660]]
```

[101]:
```
# let's copy the original triangle graph
uber_triangle_plotting_copy = uber_triangle.copy()

# let's reset default vertex parameters for the triangle graph
for i in range(uber_triangle_plotting_copy.vcount()):
```

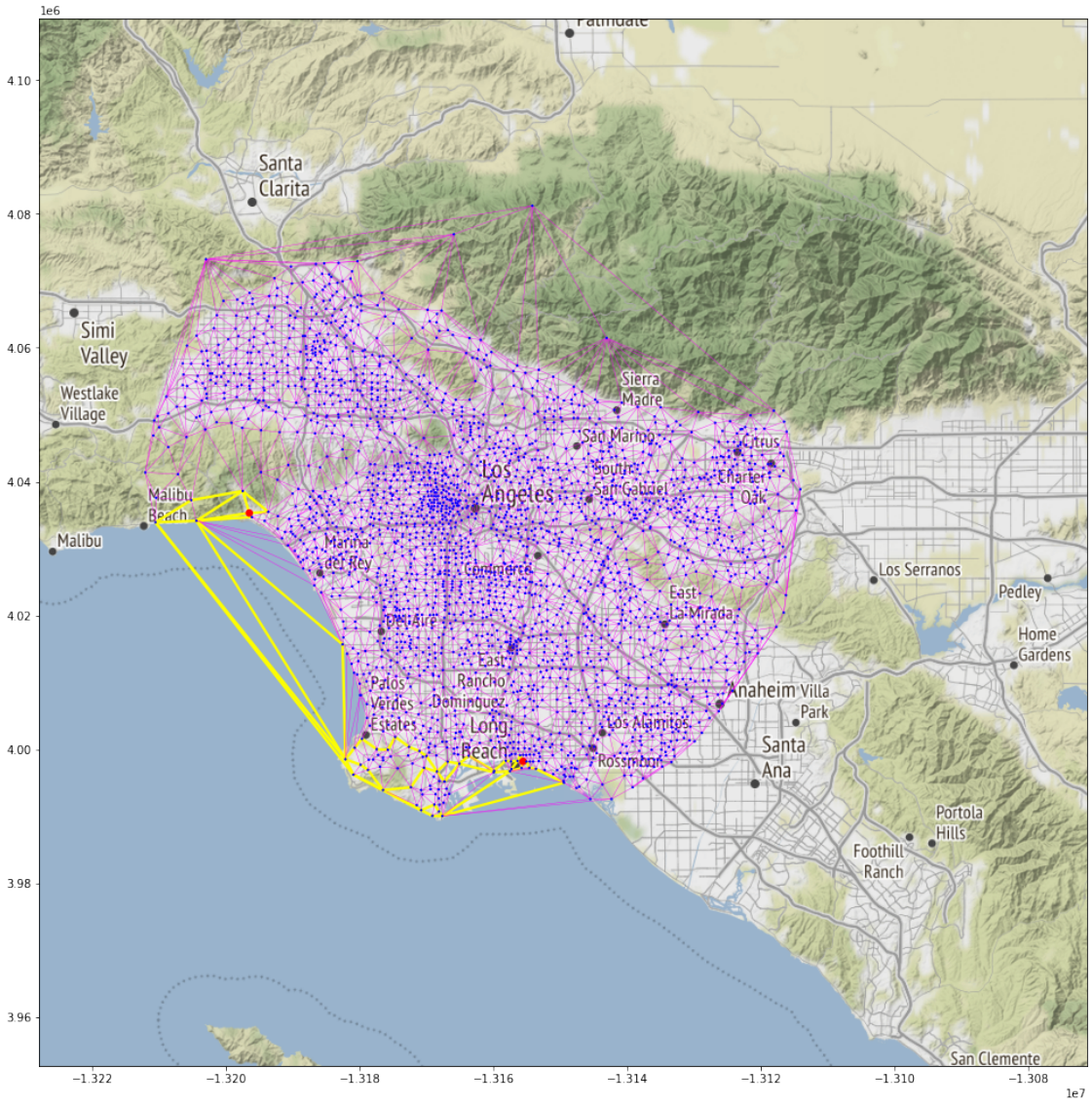
```
    uber_triangle_plotting_copy.vs[i]["color"] = (0, 0, 1)
    uber_triangle_plotting_copy.vs[i]["size"] = 0.1

# we need to specify source and destination points with big red dots
uber_triangle_plotting_copy.vs.
 ↪find(vetrex_name_string=malibu_vertex_string)["color"] = (1, 0, 0)
uber_triangle_plotting_copy.vs.
 ↪find(vetrex_name_string=malibu_vertex_string)["size"] = 2
uber_triangle_plotting_copy.vs.
 ↪find(vetrex_name_string=long_beach_vertex_string)["color"] = (1, 0, 0)
uber_triangle_plotting_copy.vs.
 ↪find(vetrex_name_string=long_beach_vertex_string)["size"] = 2

# increase size of the edges in the graph
for i in range(uber_triangle.ecount()):
    uber_triangle_plotting_copy.es[i]["color"] = (1, 0, 1)
    uber_triangle_plotting_copy.es[i]["size"] = 5
```

[102]:
```
# for each path from the paths
for i in range(len(paths)):
    for j in range(len(paths[i]) - 1):
        # find the string names of the vertices
        source_name = uber_triangle_directed_copy.
 ↪vs[paths[i][j]]["vetrex_name_string"]
        target_name = uber_triangle_directed_copy.
 ↪vs[paths[i][j+1]]["vetrex_name_string"]
        # find their id-s in plotting copy
        source_id = uber_triangle_plotting_copy.vs.
 ↪find(vetrex_name_string=source_name).index
        target_id = uber_triangle_plotting_copy.vs.
 ↪find(vetrex_name_string=target_name).index
        # find the edge they are connected by
        edge_id = uber_triangle_plotting_copy.get_eid(source_id, target_id,␣
 ↪directed=False, error=False)
        # if the id is valid
        if edge_id > 0:
            # color it and increase size
            uber_triangle_plotting_copy.es[edge_id]["color"] = (1, 1, 0)
            uber_triangle_plotting_copy.es[edge_id]["size"] = 30
        # else pop up an error
        else:
            print("no edge is found!")
```

[103]:
```
start = time.time()
plot_graph(uber_triangle_plotting_copy, "results/q13-edge-disjoint.png")
end = time.time()
```

```
print(end - start)
```

```
2.6548235416412354
```



12 Question 14

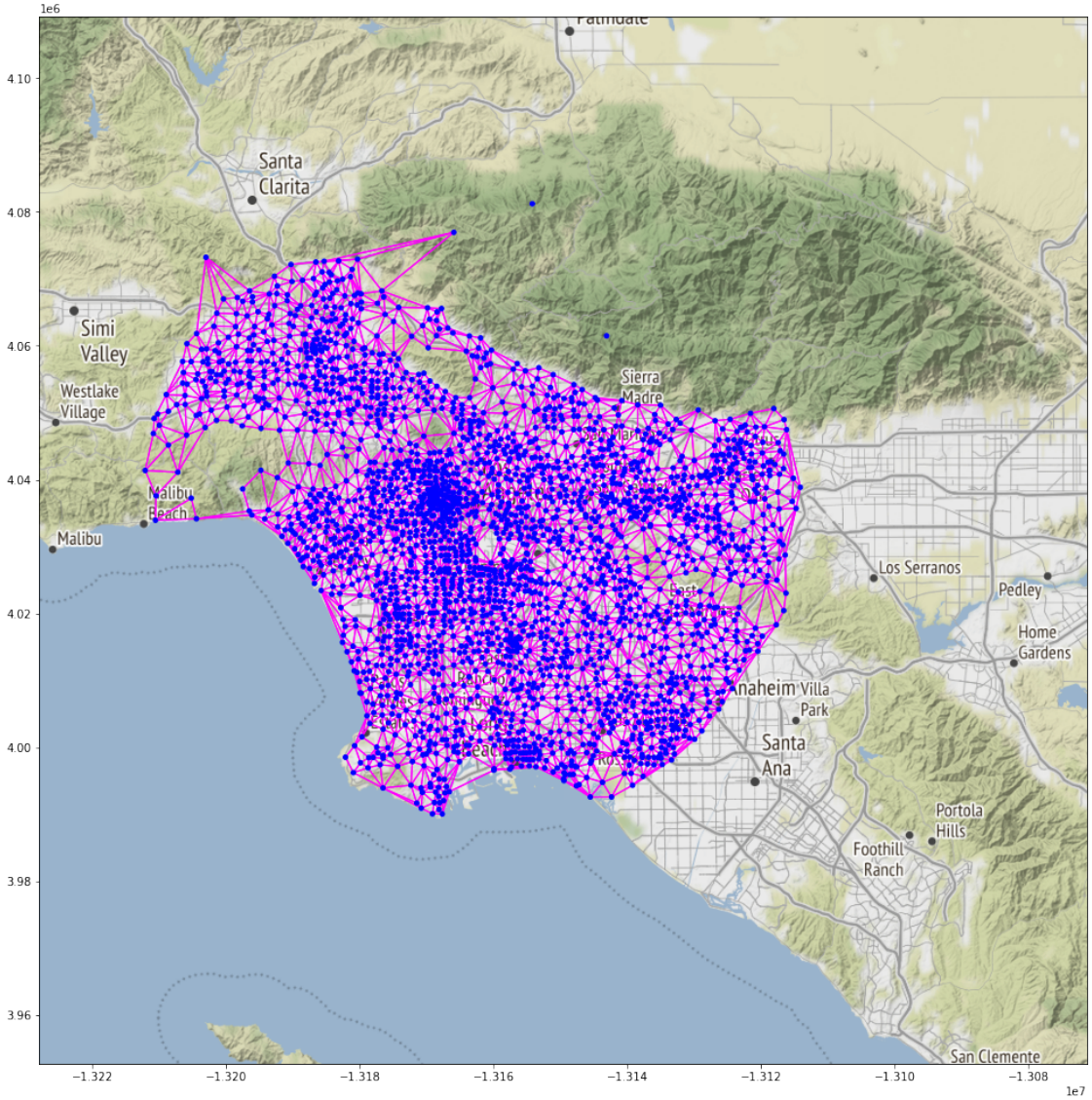
Plot $\tilde{G_\Delta}$ on actual coordinates. Do you think the thresholding method worked?

[146]:
```
uber_triangle_directed_prune = uber_triangle_directed.copy()
thresh = 600
edges_to_delete = []
for i in range(len(uber_triangle_directed_prune.es)):
```

```
    travel_time = uber_triangle_directed_prune.es[i]['mean_travel_time']
    if travel_time > thresh:
        edges_to_delete.append(i)

uber_triangle_directed_prune.delete_edges(edges_to_delete)
```

[111]:
```
start = time.time()
plot_graph(uber_triangle_directed_prune, "results/q14.png")
end = time.time()
print(end - start)
```

2.7871062755584717



Thresholding worked

13 Question 15

Now, repeat question 13 for $\tilde{G_\Delta}$ and report the results. Do you see any changes? Why?

[346]:
```
# find the id of a vertex with a coordinate closest to malibu
malibu_deg = [34.04, -118.56]
malibu_id = find_closest_vertex(malibu_deg, centroids)
print(malibu_id)
# find the string identificator of the vertex closest to malibu
malibu_vertex_string = str(malibu_id) + ".0a"
# find the vertex id
malibu_vertex = uber_triangle_directed_prune.vs.
 ↪find(vetrex_name_string=malibu_vertex_string)
print(malibu_vertex)
malibu_vertex_id = malibu_vertex.index
print(malibu_vertex_id)
# find the id of a vertex with a coordinate closest to Long Beach
long_beach_deg = [33.77, -118.18]
long_beach_id = find_closest_vertex(long_beach_deg, centroids)
print(long_beach_id)
# find the string identificator of the vertex closest to Long Beach
long_beach_vertex_string = str(long_beach_id) + ".0a"
# find the vertex id
long_beach_vertex = uber_triangle_directed_prune.vs.
 ↪find(vetrex_name_string=long_beach_vertex_string)
print(long_beach_vertex)
long_beach_vertex_id = long_beach_vertex.index
print(long_beach_vertex_id)
```

```
1523
igraph.Vertex(<igraph.Graph object at 0x7fcdf98f78b0>, 1511, {'name': 1523.0,
'centroid': [-118.54625915050165, 34.048403046822735], 'vetrex_name_string':
'1523.0a', 'x': -13196509.204081291, 'y': 4035303.2259358317})
1511
672
igraph.Vertex(<igraph.Graph object at 0x7fcdf98f78b0>, 660, {'name': 672.0,
'centroid': [-118.17865950000001, 33.771767700000005], 'vetrex_name_string':
'672.0a', 'x': -13155588.198171662, 'y': 3998197.020574805})
660
```

[441]:
```
# find the vertex with a coordinate closest to malibu
M_to_LB_flow = uber_triangle_directed_prune.maxflow(malibu_vertex_id,
                                                    long_beach_vertex_id,
                                                    capacity='max_flow')
print(M_to_LB_flow)
```

```
Graph flow (4 edges, 3 vs 2646 vertices, value=11074.1447)
```

We can see that the value of max flow is the same, from which we can conclude that these were roads leading to Malibu Beach that were the bottleneck, not the other parts of the graph

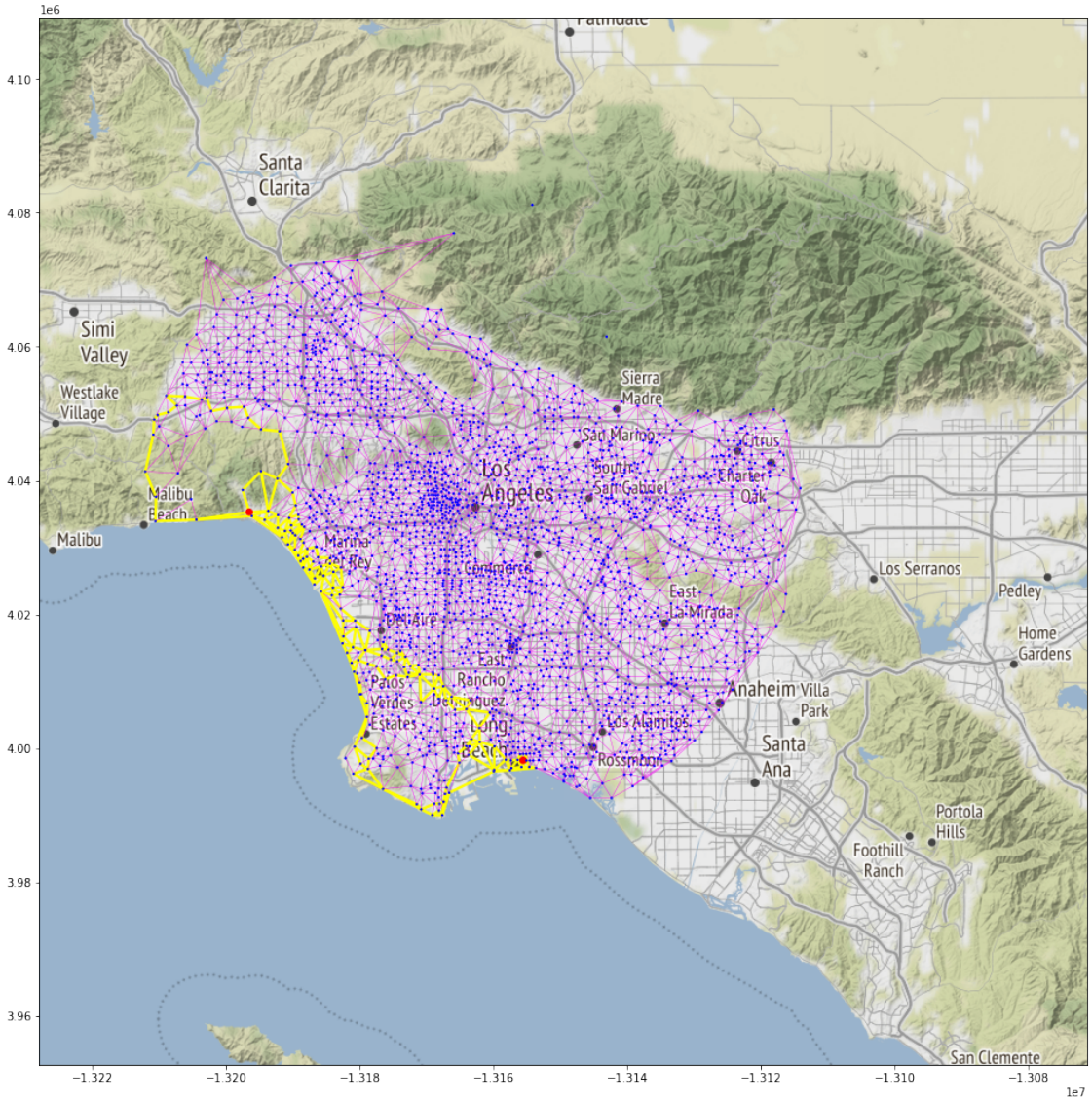
[499]:
```
# let's copy the original triangle graph
uber_triangle_plotting_copy = uber_triangle.copy()

thresh = 600
edges_to_delete = []
for i in range(len(uber_triangle_plotting_copy.es)):
    travel_time = uber_triangle_plotting_copy.es[i]['mean_travel_time']
    if travel_time > thresh:
        edges_to_delete.append(i)

uber_triangle_plotting_copy.delete_edges(edges_to_delete)

# let's reset default vertex parameters for the triangle graph
for i in range(uber_triangle_plotting_copy.vcount()):
    uber_triangle_plotting_copy.vs[i]["color"] = (0, 0, 1)
    uber_triangle_plotting_copy.vs[i]["size"] = 0.1

# we need to specify source and destination points with big red dots
uber_triangle_plotting_copy.vs.
 ↪find(vetrex_name_string=malibu_vertex_string)["color"] = (1, 0, 0)
uber_triangle_plotting_copy.vs.
 ↪find(vetrex_name_string=malibu_vertex_string)["size"] = 2
uber_triangle_plotting_copy.vs.
 ↪find(vetrex_name_string=long_beach_vertex_string)["color"] = (1, 0, 0)
uber_triangle_plotting_copy.vs.
 ↪find(vetrex_name_string=long_beach_vertex_string)["size"] = 2

# increase size of the edges in the graph
for i in range(uber_triangle_plotting_copy.ecount()):
    uber_triangle_plotting_copy.es[i]["color"] = (1, 0, 1)
    uber_triangle_plotting_copy.es[i]["size"] = 5
```

[500]:
```
# for each edge
for i in range(uber_triangle_directed_prune.ecount()):
    # which has a positive flow
    if M_to_LB_flow.flow[i] > 0:
        # find the id-s of the vertices it connects
        source_id, target_id = uber_triangle_directed_prune.es[i].source,␣
 ↪uber_triangle_directed_prune.es[i].target
        # find their string names
        source_name = uber_triangle_directed_prune.
 ↪vs[source_id]["vetrex_name_string"]
```

```
        target_name = uber_triangle_directed_prune.
↪vs[target_id]["vetrex_name_string"]
        # find their id-s in plotting copy
        source_id = uber_triangle_plotting_copy.vs.
↪find(vetrex_name_string=source_name).index
        target_id = uber_triangle_plotting_copy.vs.
↪find(vetrex_name_string=target_name).index
        # find the edge they are connected by
        edge_id = uber_triangle_plotting_copy.get_eid(source_id, target_id,␣
↪directed=False, error=False)
        # if coordinate is valid
        if source_id not in exclude and target_id not in exclude:
            # if the id is valid
            if edge_id > 0:
                # color it and increase size
                uber_triangle_plotting_copy.es[edge_id]["color"] = (1, 1, 0)
                uber_triangle_plotting_copy.es[edge_id]["size"] = 30
            # else pop up an error
            else:
                print("no edge is found!")
```

[501]:
```
start = time.time()
plot_graph(uber_triangle_plotting_copy, "results/q15-maxflow.png")
end = time.time()
print(end - start)
```

```
2.5086188316345215
```



[140]:
```
M_to_LB_paths = uber_triangle_directed_prune.edge_disjoint_paths(source =␣
↪malibu_vertex_id,
                                                                 target =␣
↪long_beach_vertex_id)
```

[141]:
```
print(M_to_LB_paths)
```

```
4
```

[139]:
```
# let's init paths list
paths = []
# get a copy of traingle graph
uber_triangle_directed_copy = uber_triangle_directed_prune.copy()
```

```
# for each path in edge disjoint paths:
for i in range(M_to_LB_paths):
    # append a vertex path to the list
    paths.append(uber_triangle_directed_copy.
 ↪get_shortest_paths(malibu_vertex_id,
                                                                 ␣
 ↪to=long_beach_vertex_id,
                                                                  mode='out',
                                                                 ␣
 ↪output='vpath')[0])
    # get an edge path
    edges = uber_triangle_directed_copy.get_shortest_paths(malibu_vertex_id,
                                                              ␣
 ↪to=long_beach_vertex_id,
                                                               mode='out',
                                                               output='epath')[0]
    # delete those edges
    uber_triangle_directed_copy.delete_edges(np.array(edges))

print(paths)
```

```
[[1511, 1512, 1941, 1950, 1584, 1590, 2406, 1612, 1701, 1704, 426, 2402, 1875,
1782, 1783, 1860, 1861, 1658, 1660, 1661, 1686, 2423, 2373, 663, 660], [1511,
1509, 1513, 1940, 1939, 1950, 1583, 1584, 1588, 1589, 2406, 1604, 1612, 324,
320, 1705, 424, 561, 560, 2399, 2016, 2017, 2134, 2365, 163, 165, 170, 610, 647,
651, 652, 654, 660], [1511, 1510, 1678, 1508, 1505, 1014, 1015, 1502, 112, 1545,
1546, 1569, 1571, 249, 1592, 1599, 1607, 1611, 1610, 323, 1705, 1706, 1866,
1868, 2020, 2021, 2031, 1620, 2367, 158, 160, 166, 170, 1676, 684, 2373, 664,
661, 660], [1511, 1700, 1512, 1509, 1507, 1506, 1504, 1503, 1520, 1533, 1545,
1547, 1554, 1560, 244, 246, 251, 1600, 1609, 1610, 2393, 407, 406, 418, 2395,
326, 330, 329, 1616, 149, 154, 152, 167, 168, 604, 605, 603, 616, 617, 642, 657,
660]]
```

[143]:
```
# let's copy the original triangle graph
uber_triangle_plotting_copy = uber_triangle_directed_prune.copy()

thresh = 600
edges_to_delete = []
for i in range(len(uber_triangle_plotting_copy.es)):
    travel_time = uber_triangle_plotting_copy.es[i]['mean_travel_time']
    if travel_time > thresh:
        edges_to_delete.append(i)

uber_triangle_plotting_copy.delete_edges(edges_to_delete)

# let's reset default vertex parameters for the triangle graph
for i in range(uber_triangle_plotting_copy.vcount()):
```

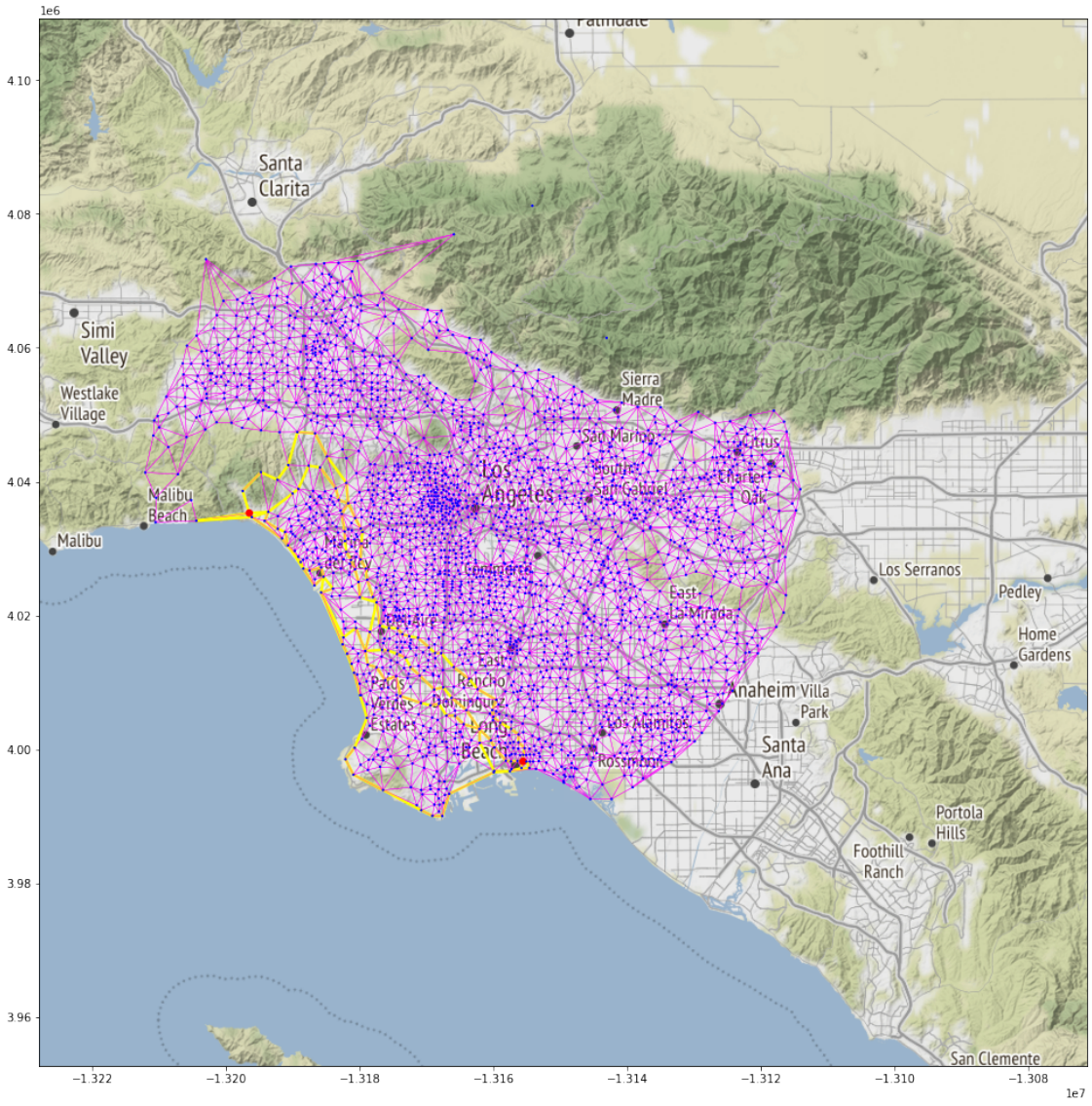
```
    uber_triangle_plotting_copy.vs[i]["color"] = (0, 0, 1)
    uber_triangle_plotting_copy.vs[i]["size"] = 0.1

# we need to specify source and destination points with big red dots
uber_triangle_plotting_copy.vs.
 ↪find(vetrex_name_string=malibu_vertex_string)["color"] = (1, 0, 0)
uber_triangle_plotting_copy.vs.
 ↪find(vetrex_name_string=malibu_vertex_string)["size"] = 2
uber_triangle_plotting_copy.vs.
 ↪find(vetrex_name_string=long_beach_vertex_string)["color"] = (1, 0, 0)
uber_triangle_plotting_copy.vs.
 ↪find(vetrex_name_string=long_beach_vertex_string)["size"] = 2

# increase size of the edges in the graph
for i in range(uber_triangle_directed_prune.ecount()):
    uber_triangle_plotting_copy.es[i]["color"] = (1, 0, 1)
    uber_triangle_plotting_copy.es[i]["size"] = 5
```

[144]:
```
# for each path from the paths
for i in range(len(paths)):
    for j in range(len(paths[i]) - 1):
        # find the string names of the vertices
        source_name = uber_triangle_directed_prune.
 ↪vs[paths[i][j]]["vetrex_name_string"]
        target_name = uber_triangle_directed_prune.
 ↪vs[paths[i][j+1]]["vetrex_name_string"]
        # find their id-s in plotting copy
        source_id = uber_triangle_plotting_copy.vs.
 ↪find(vetrex_name_string=source_name).index
        target_id = uber_triangle_plotting_copy.vs.
 ↪find(vetrex_name_string=target_name).index
        # find the edge they are connected by
        edge_id = uber_triangle_plotting_copy.get_eid(source_id, target_id,␣
 ↪directed=False, error=False)
        # if the id is valid
        if edge_id > 0:
            # color it and increase size
            uber_triangle_plotting_copy.es[edge_id]["color"] = (1, 1, 0)
            uber_triangle_plotting_copy.es[edge_id]["size"] = 30
        # else pop up an error
        else:
            print("no edge is found!")
```

[145]:
```
start = time.time()
plot_graph(uber_triangle_plotting_copy, "results/q15-edge-disjoint.png")
end = time.time()
```

```
print(end - start)
```

```
3.289912462234497
```



In [61]:
```
import csv
import numpy as np
from scipy import stats
import igraph
import os
import time
import json
import random
import itertools
import math
import matplotlib.pyplot as plt
```

In [8]:
```
import pickle
# Loads lines of the dataset that were used to make all the edges, plus the two graphs
with open('save_variables.pickle', 'rb') as f:
    pre_pandemic_rides_in_graph,post_pandemic_rides_in_graph,g_pre,g_post = pickle.load
```

In [9]:
```
print(g_pre.degree(0))
```

```
1073
```

In [10]:
```
print(g_post.degree(0))
```

```
1115
```

In [30]:
```
print(len(pre_pandemic_rides_in_graph))
print(len(post_pandemic_rides_in_graph))
```

```
267854
293671
```

In [6]:
```
g_post
```

Out[6]:
```
<igraph.Graph at 0x20098019d68>
```

In [25]:
```
GEO_BOUNDARIES = os.path.join('Uber_Data', "los_angeles_censustracts.json")

# load geographical data of node id-s
with open(GEO_BOUNDARIES) as f:
    geo = json.load(f)


def find_centroid(points):
    # this function finds centroid of a set of points
    # init center coordinates
    center = [0, 0]
    # check if there are nested lists
    if not type(points[0][0]) is float:
        # if there is only one list nested
        if len(points) == 1:
            # just access it
            points = points[0]
        # if not
```

```
        else:
            # flatten the whole structure
            points = list(itertools.chain.from_iterable(points))
    # sweep through all coordinates
    for j in range(len(points)):
        # update coordinates with corresponding values
        center[0] += points[j][0]
        center[1] += points[j][1]
    # normilize coordinates
    center[0] /= len(points)
    center[1] /= len(points)

    return center


# init a dictionary to hold coordinates
centroids = {}

# sweep through the file
for i in range(len(geo["features"])):
    try:
        centroids[int(geo['features'][i]['properties']['MOVEMENT_ID'])] = \
            find_centroid(geo['features'][i]['geometry']['coordinates'][0])
    except TypeError:
        print("error in", i)
        pass
```

In [32]:
```
centroids[2054][0] - centroids[2053][0]
```

Out[32]: -0.004793566801637894

In [54]:
```
predist = np.zeros(len(pre_pandemic_rides_in_graph))
postdist = np.zeros(len(post_pandemic_rides_in_graph))
templat = 0
templong = 0
startid = 0
endid = 0
for i in range(len(pre_pandemic_rides_in_graph)):
    startid = int(pre_pandemic_rides_in_graph[i][0])
    endid = int(pre_pandemic_rides_in_graph[i][1])
    templat = 69 * ((centroids[startid][0]) - (centroids[endid][0]))
    templong = 54.6 * (centroids[startid][1] - centroids[endid][1])
    tempdist = (templat*templat) + (templong*templong)
    predist[i] = math.sqrt(tempdist)
for i in range(len(post_pandemic_rides_in_graph)):
    startid = int(post_pandemic_rides_in_graph[i][0])
    endid = int(post_pandemic_rides_in_graph[i][1])
    templat = 69 * ((centroids[startid][0]) - (centroids[endid][0]))
    templong = 54.6 * (centroids[startid][1] - centroids[endid][1])
    tempdist = (templat*templat) + (templong*templong)
    postdist[i] = math.sqrt(tempdist)
```

In [58]:
```
print(np.sum(predist)/len(pre_pandemic_rides_in_graph))
print(np.sum(postdist)/len(post_pandemic_rides_in_graph))
```

```
7.609031985965012
7.983110076394108
```

In [69]:

```
print(np.max(predist))
print(np.max(postdist))
```

```
30.532671690922626
32.600449661383884
```

In [ ]:

```
!pip install pyproj
!pip install python-igraph
!pip install contextily
```

```
Requirement already satisfied: pyproj in /usr/local/lib/python3.7/dist-packages (3.1.0)
Requirement already satisfied: certifi in /usr/local/lib/python3.7/dist-packages (from
Requirement already satisfied: python-igraph in /usr/local/lib/python3.7/dist-packages
Requirement already satisfied: texttable>=1.6.2 in /usr/local/lib/python3.7/dist-packag
Requirement already satisfied: contextily in /usr/local/lib/python3.7/dist-packages (1.
Requirement already satisfied: rasterio in /usr/local/lib/python3.7/dist-packages (from
Requirement already satisfied: requests in /usr/local/lib/python3.7/dist-packages (from
Requirement already satisfied: mercantile in /usr/local/lib/python3.7/dist-packages (fr
Requirement already satisfied: matplotlib in /usr/local/lib/python3.7/dist-packages (fr
Requirement already satisfied: joblib in /usr/local/lib/python3.7/dist-packages (from c
Requirement already satisfied: pillow in /usr/local/lib/python3.7/dist-packages (from c
Requirement already satisfied: geopy in /usr/local/lib/python3.7/dist-packages (from co
Requirement already satisfied: cligj>=0.5 in /usr/local/lib/python3.7/dist-packages (fr
Requirement already satisfied: setuptools in /usr/local/lib/python3.7/dist-packages (fr
Requirement already satisfied: affine in /usr/local/lib/python3.7/dist-packages (from r
Requirement already satisfied: certifi in /usr/local/lib/python3.7/dist-packages (from
Requirement already satisfied: click>=4.0 in /usr/local/lib/python3.7/dist-packages (fr
Requirement already satisfied: click-plugins in /usr/local/lib/python3.7/dist-packages
Requirement already satisfied: snuggs>=1.4.1 in /usr/local/lib/python3.7/dist-packages
Requirement already satisfied: numpy in /usr/local/lib/python3.7/dist-packages (from ra
Requirement already satisfied: attrs in /usr/local/lib/python3.7/dist-packages (from ra
Requirement already satisfied: urllib3!=1.25.0,!=1.25.1,<1.26,>=1.21.1 in /usr/local/li
Requirement already satisfied: chardet<4,>=3.0.2 in /usr/local/lib/python3.7/dist-packa
Requirement already satisfied: idna<3,>=2.5 in /usr/local/lib/python3.7/dist-packages (
Requirement already satisfied: python-dateutil>=2.1 in /usr/local/lib/python3.7/dist-pa
Requirement already satisfied: kiwisolver>=1.0.1 in /usr/local/lib/python3.7/dist-packa
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.7/dist-packages (
Requirement already satisfied: pyparsing!=2.0.4,!=2.1.2,!=2.1.6,>=2.0.1 in /usr/local/l
Requirement already satisfied: geographiclib<2,>=1.49 in /usr/local/lib/python3.7/dist-
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.7/dist-packages (from
```

```
import os
import time
import json
import random
import itertools
import csv
import numpy as np
from scipy import stats
import igraph
import numpy as np
import pandas as pd
import igraph as ig
import contextily as cx
import matplotlib.pyplot as plt
from matplotlib.collections import LineCollection
from pyproj import Transformer
from scipy.spatial import Delaunay
```

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mou
```

```
list_jan_data = []
list_feb_data = []
list_mar_data = []
list_all_data = []

with open('/content/drive/My Drive/uberdata/los_angeles-censustracts-2020-1-All-MonthlyAggreg
    for line in f1:
        line =line.strip('\n')
        arr = line.split(',')
        #print(arr[2])
        list_all_data.append(arr)
        if arr[2] == '01':
            list_jan_data.append(arr)
        if arr[2] == '02':
            list_feb_data.append(arr)
        if arr[2] == '03':
            list_mar_data.append(arr)



# Make everything into a numpy array, rows are still for each entry and cols are attributes
list_jan_data = np.array(list_jan_data)
list_feb_data = np.array(list_feb_data)
list_mar_data = np.array(list_mar_data)
list_all_data = np.array(list_all_data)


print(list_mar_data[:,0])
```

```
['1575' '939' '988' ... '167' '585' '679']
```

```
# "Source" is the "main node" the placeholder for DTLA
source = '1575'
# Iterate through the list
# I tried this with January + February but it made less sense
#pre_pandemic = np.vstack((list_jan_data,list_feb_data))
pre_pandemic = list_jan_data

post_pandemic = list_mar_data

# Find the pre-pandemic entries from location '1575' that take less than 30 mins
# Initialize np array with 0 rows and correct # of cols
pre_pandemic_less_30 = np.zeros((0,pre_pandemic.shape[1]))
```

```
# Iterate row by row
for line_num in range(pre_pandemic.shape[0]):
    # Check source location
    if pre_pandemic[line_num,0] == source:
        # Check that mean travel time is less than 30 mins (or 1800 seconds)
        if float(pre_pandemic[line_num,3]) < 300.0:
            pre_pandemic_less_30 = np.vstack((pre_pandemic_less_30,pre_pandemic[line_num,:]))


# Find the post-pandemic entries from location '1575' that take less than 30 mins
# Initialize np array with 0 rows and correct # of cols
post_pandemic_less_30 = np.zeros((0,post_pandemic.shape[1]))

# Iterate row by row
for line_num in range(post_pandemic.shape[0]):
    # Check source location
    if post_pandemic[line_num,0] == source:
        # Check that mean travel time is less than 30 mins (or 1800 seconds)
        if float(post_pandemic[line_num,3]) < 300.0:
            post_pandemic_less_30 = np.vstack((post_pandemic_less_30,post_pandemic[line_num,:


print(pre_pandemic.shape)
print(pre_pandemic_less_30.shape)
print(post_pandemic.shape)
print(post_pandemic_less_30.shape)
```

```
(1564592, 7)
(28, 7)
(1247442, 7)
(31, 7)
```

```
#GEO_BOUNDARIES = os.path.join('uber data', "los_angeles_censustracts.json")

# load geographical data of node id-s
with open('/content/drive/My Drive/uberdata/los_angeles_censustracts.json', 'r') as f:
    geo = json.load(f)



def find_centroid(points):
    # this function finds centroid of a set of points
    # init center coordinates
    center = [0, 0]
    # check if there are nested lists
    if not type(points[0][0]) is float:
        # if there is only one list nested
        if len(points) == 1:
            # just access it
            points = points[0]
```

```
            # if not
            else:
                # flatten the whole structure
                points = list(itertools.chain.from_iterable(points))
        # sweep through all coordinates
        for j in range(len(points)):
            # update coordinates with corresponding values
            center[0] += points[j][0]
            center[1] += points[j][1]
        # normilize coordinates
        center[0] /= len(points)
        center[1] /= len(points)

        return center


    # init a dictionary to hold coordinates
    centroids = {}

    # sweep through the file
    for i in range(len(geo["features"])):
        try:
            centroids[int(geo['features'][i]['properties']['MOVEMENT_ID'])] = \
                find_centroid(geo['features'][i]['geometry']['coordinates'][0])
        except TypeError:
            print("error in", i)
            pass


    # Make two graphs: One for pre-pandemic and one post-pandemic
    # The nodes: '1575' and every number that exists in the second column above (destinations)
    # The edges: An edge exists between two nodes if their travel time is less than 30 mins
    # So an edge can exist between two nodes even if neither is '1575'

    # Number of vertices pre-pandemic
    num_vertices_pre = 1+pre_pandemic_less_30.shape[0]
    # Create a directed graph
    g_pre = igraph.Graph(directed=True)
    # Add num_vertices_pre vertices
    g_pre.add_vertices(num_vertices_pre)

    # Add ids and labels to vertices
    # First label is '1575' and then iterate through the rest
    g_pre.vs[0]["id"] = 0
    g_pre.vs[0]["label"] = '1575'
    g_pre.vs[0]["centroid"] = centroids[1575]
    for i in range(1,len(g_pre.vs)):
        #print(i)
        g_pre.vs[i]["id"]= i
        # Label is the location id
        g_pre.vs[i]["label"]= pre_pandemic_less_30[i-1,1]
        label = int(g_pre.vs[i]["label"])
```

```
        g_pre.vs[i]["centroid"] = centroids[label]


    node_labels = g_pre.vs["label"]
    # For all the node labels
    # Iterate through each row of list_jan_data
    # if source and dest are both in node_labels
    # if the travel time is less than 30 mins
    # If an edge doesn't already exist between these two nodes
    # Add an edge between these two nodes
    # And add this line to a new matrix in case we need it for later

    # ADD EDGES FOR PRE-PANDEMIC
    pre_pandemic_rides_in_graph = np.zeros((0,pre_pandemic.shape[1]))
    for line_num in range(pre_pandemic.shape[0]):
        # Check that source and destination are both in node_labels
        if pre_pandemic[line_num,0] in node_labels:
            if pre_pandemic[line_num,1] in node_labels:
                # Check that travel time is less than 30 mins
                if float(pre_pandemic[line_num,3]) < 300.0:
                    # Get the corresponding node ids
                    v1 = node_labels.index(pre_pandemic[line_num,0])
                    v2 = node_labels.index(pre_pandemic[line_num,1])
                    # Check if these two nodes are already connected
                    if g_pre.get_eid(v1, v2, directed=False, error=False) == (-1):
                        if (v1 != v2):
                        # Add an edge between v1 and v2
                            #print(v1,v2)
                            g_pre.add_edges([(v1, v2)])
                        # Add this line to a new matrix
                            pre_pandemic_rides_in_graph= np.vstack((pre_pandemic_rides_in_graph,p


    # Number of vertices post-pandemic
    num_vertices_post = 1+post_pandemic_less_30.shape[0]
    # Create a directed graph
    g_post = igraph.Graph(directed=True)
    # Add num_vertices_pre vertices
    g_post.add_vertices(num_vertices_post)

    # Add ids and labels to vertices
    # First label is '1575' and then iterate through the rest
    g_post.vs[0]["id"] = 0
    g_post.vs[0]["label"] = '1575'
    g_post.vs[0]["centroid"] = centroids[1575]
    label = 1575
    for i in range(1,len(g_post.vs)):
        #print(i)
        g_post.vs[i]["id"]= i
        # Label is the location id
        g_post.vs[i]["label"]= post_pandemic_less_30[i-1,1]
        label = int(g_post.vs[i]["label"])
```

```
        label = int(g_post.vs[i]['label'])
        g_post.vs[i]["centroid"] = centroids[label]


    node_labels_post = g_post.vs["label"]
    # ADD EDGES FOR POST-PANDEMIC
    post_pandemic_rides_in_graph = np.zeros((0,post_pandemic.shape[1]))
    for line_num in range(post_pandemic.shape[0]):
        # Check that source and destination are both in node_labels
        if post_pandemic[line_num,0] in node_labels_post:
            if post_pandemic[line_num,1] in node_labels_post:
                # Check that travel time is less than 30 mins
                if float(post_pandemic[line_num,3]) < 300.0:
                    # Get the corresponding node ids
                    v1 = node_labels_post.index(post_pandemic[line_num,0])
                    v2 = node_labels_post.index(post_pandemic[line_num,1])
                    # Check if these two nodes are already connected
                    if g_post.get_eid(v1, v2, directed=False, error=False) == (-1):
                        if (v1 != v2):
                        # Add an edge between v1 and v2
    #                         if (v1 == 0) or (v2 == 0):
    #                             print(v1,v2)
                            g_post.add_edges([(v1, v2)])
                        # Add this line to a new matrix
                            post_pandemic_rides_in_graph= np.vstack((post_pandemic_rides_in_graph


    def plot_graph(g, path, margin = 0.1):
        """
        a function that allows to plot specified graph over a map
        Args:
            g (igraph.Grah):        graph to plot. Each vertex needs to have attributes:
                                        centroid    latitude and longditude of the point
                                        size        a multipler of the preset diameter; size = 1
                                                    1 / 10000 of the figure size
                                        color       color of the vertex
                                    Each edge needs to have attributes:
                                        width       a multipler of the preset width; width = 1 cc
                                                    1 / 2000000 of the figure size
                                        color       color of the edge
            path (str):             path to save graph to
            margin (float):         how much of additional map space is added to the bounding box
                                    coordinates
        """
        # init output figure
        fig, ax = plt.subplots(figsize=(17, 17))
        # init coordinates of the snapshot
        west, south, east, north = centroids[1][0], centroids[1][1], centroids[1][0], centroids[1
        # sweep through all centroids to crorect coordinates of the bounding box
        for key in centroids.keys():
            if centroids[key][0] < west:
                west = centroids[key][0]
            if centroids[key][0] > east:
```

```
        if centroids[key][0] > east:
            east = centroids[key][0]
        if centroids[key][1] > north:
            north = centroids[key][1]
        if centroids[key][1] < south:
            south = centroids[key][1]
    # get the center of snapshot
    x_center_degrees = (west - east) / 2 + east
    y_center_degrees = (north - south) / 2 + south
    # choose the higher dimensions, and make the dimension of the other coordinate match with
    # to both dimesions
    margin = -0.6
    # choose an arbitrary dimension
    dim = abs(west - east)
    # check if it is the largest, and if it is not, reset it
    if abs(north - south) > dim: dim = abs(north - south)
    # reset the coordinates
    west = x_center_degrees - dim * (1 + (1/2) * margin) / 2
    east = x_center_degrees + dim * (1 + 2 * margin) / 2
    north = y_center_degrees + dim * (1 + 1.5 * margin) / 2
    south = y_center_degrees - dim * (1 + 1.4 * margin) / 2
    # download a snapshot of the map
    ghent_img, ghent_ext = cx.bounds2img(west,
                                         south,
                                         east,
                                         north,
                                         ll=True,
                                         source=cx.providers.Stamen.Terrain)
    # add it to the figure
    ax.imshow(ghent_img, extent=ghent_ext)
    # at this point a picture is guaranteed to have "approximately" square form; if precisior
    # this can be relaced by separate x and y scales
    scale_contextily = abs(ghent_ext[0] - ghent_ext[1])
    # calculate unit diameter of points and width of lines
    unit_vertex = scale_contextily / 1000
    unit_edge = scale_contextily / 20000
    # initialize transformer to convert coordinates to web merkator
    transformer = Transformer.from_crs("epsg:4326", "epsg:3857")
    # init arrays for vertices' coordinates, colors and sizes
    vertex_xs = np.zeros((g.vcount()))
    vertex_ys = np.zeros((g.vcount()))
    vertex_colors = np.zeros((g.vcount(), 3))
    vertex_sizes = np.zeros((g.vcount()))
    # init array for edges coordinates: (num_lines, num_point_in_line, dots_dim), colors and
    edges_coords = np.zeros((g.ecount(), 2, 2))
    edges_colors = np.zeros((g.ecount(), 3))
    edges_sizes = np.zeros((g.ecount()))
    # for each vertex
    for i in range(g.vcount()):
        # convert coordinates to web merkator
        g.vs[i]['x'], g.vs[i]['y'] = transformer.transform(g.vs[i]["centroid"][1], g.vs[i]["c
        # store coordinates in the array for plotting
```

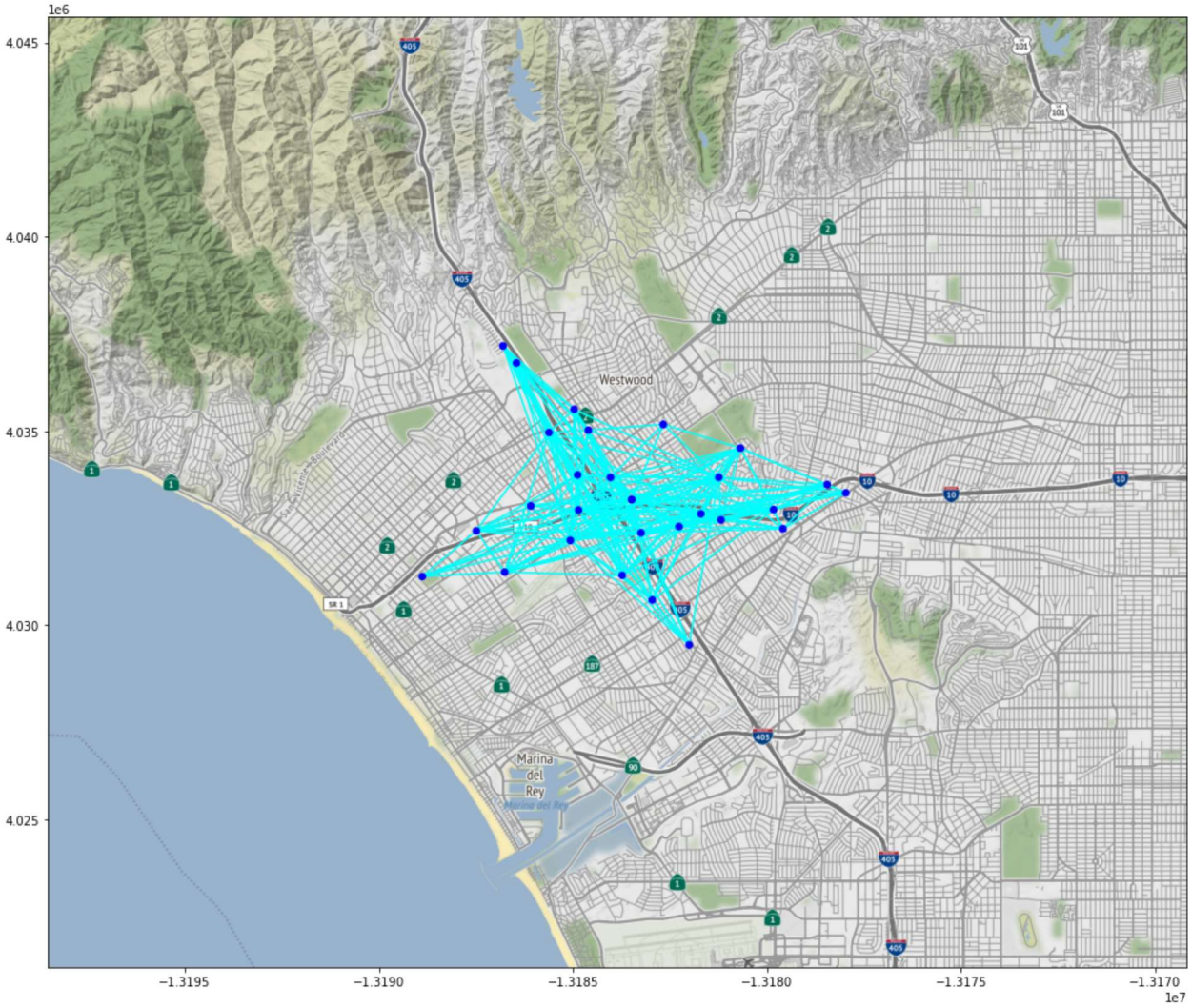
```
            vertex_xs[i] = g.vs[i]['x']
            vertex_ys[i] = g.vs[i]['y']
            # store rgb colors for plotting, if color is present in attributes
            if "color" in g.vs[i].attributes().keys():
                vertex_colors[i] = g.vs[i]["color"]
            else:
                # else set default blue
                vertex_colors[i] = (0, 0, 1)
            # store relative sizes for plotting, if size is present in attributes
            if "size" in g.vs[i].attributes().keys():
                vertex_sizes[i] = g.vs[i]["size"]
            else:
                # else set default 1
                vertex_sizes[i] = 1
        # for each edge
        for i in range(g.ecount()):
            # get the edge
            ed = g.es[i]
            # get the id-s of a start and end points
            v_source_id, v_target_id = ed.source, ed.target
            # fill the coordinates for the source
            edges_coords[i, 0, 0], edges_coords[i, 0, 1] = g.vs[v_source_id]["x"], g.vs[v_source_
            # fill the coordinates for the target
            edges_coords[i, 1, 0], edges_coords[i, 1, 1] = g.vs[v_target_id]["x"], g.vs[v_target_
            # store rgb colors for plotting, if color is present in attributes
            if "color" in g.es[i].attributes().keys():
                edges_colors[i] = g.es[i]["color"]
            else:
                # else set default cyan
                edges_colors[i] = (0, 1, 1)
            # store relative sizes for plotting, if size is present in attributes
            if "size" in g.es[i].attributes().keys():
                edges_sizes[i] = g.es[i]["size"]
            else:
                # else set default 1
                edges_sizes[i] = 1
        # Make a sequence of (x, y) pairs.
        line_segments = LineCollection(edges_coords,
                                       linewidths=edges_sizes * unit_edge,
                                       linestyles='solid',
                                       colors=edges_colors)
        ax.add_collection(line_segments)
        # create a scatter plot for vertices
        ax.scatter(vertex_xs,
                   vertex_ys,
                   s=vertex_sizes * unit_vertex,
                   c=vertex_colors,
                   marker='o',
                   zorder=2)
        # save the figure
        plt.savefig(path, dpi=300)
```

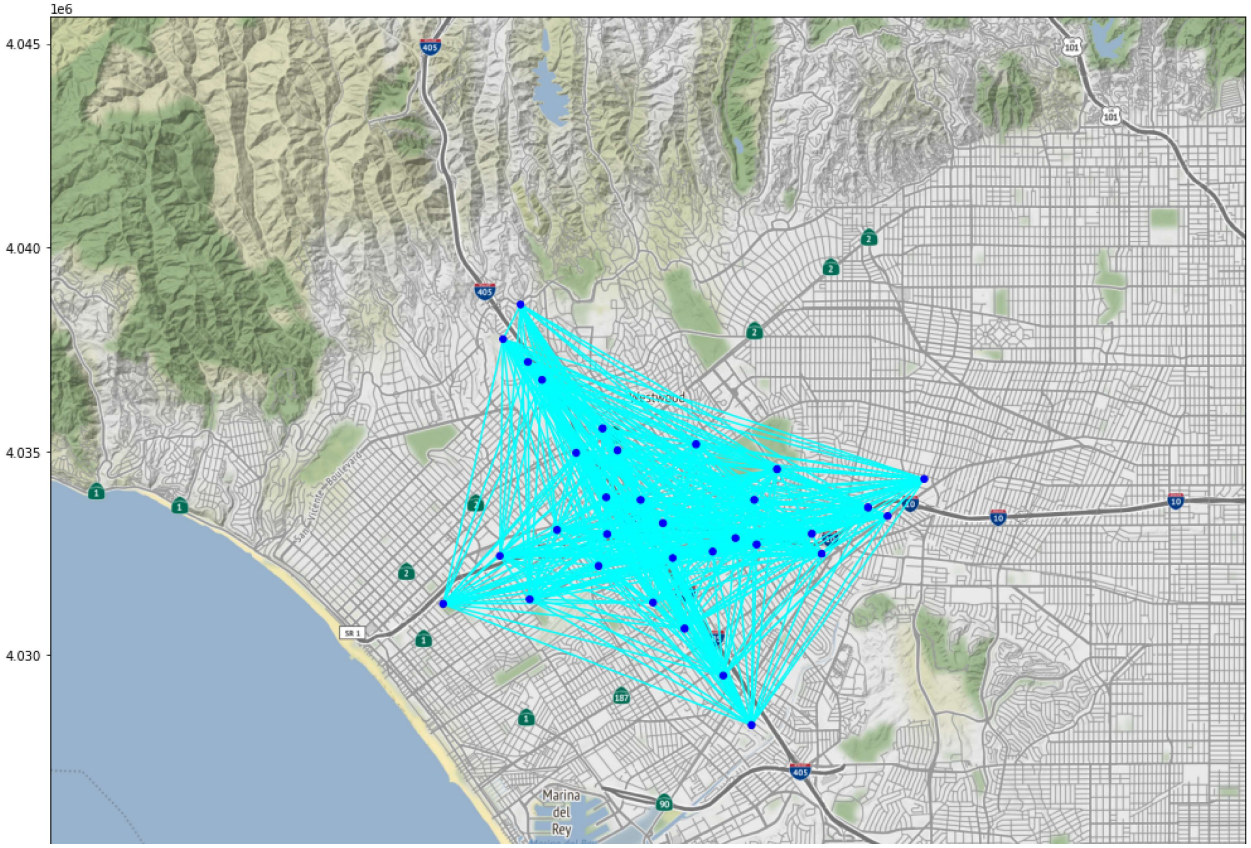
```
print(g_pre.ecount())
```

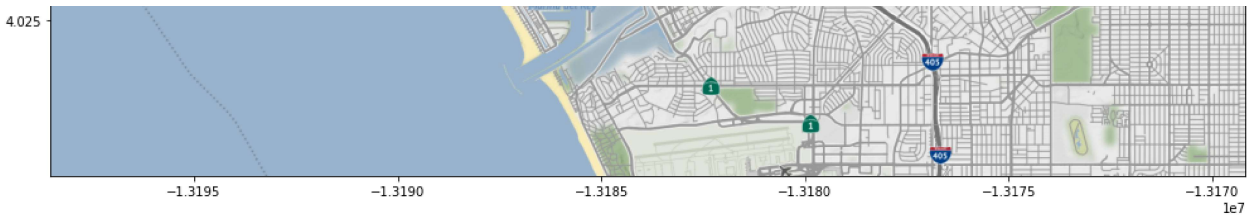
```
206
```

```
start = time.time()
plot_graph(g_pre, "part4-pre.png")
plot_graph(g_post, "part4-post.png")
end = time.time()
print(end - start)
```

```
17.066243886947632
```







✓ 21s completed at 5:18 PM