pj4

June 9, 2022

1 Project4: Graph Algorithms

- 505851728 Yang-Shan Chen
- 005627440 Chih-En Lin
- 505297814 Rikako Hatoya

[ ]:
```
install.packages("igraph")
library('igraph')
```

```
Installing package into '/usr/local/lib/R/site-library'
(as 'lib' is unspecified)
```

2 Q1

Since $\rho_{ij} = \frac{<r_i(t)r_j(t)> - <r_i(t)><r_j(t)>}{\sqrt{(<r_i(t)^2> - <r_i(t)>^2)(<r_j(t)^2> - <r_j(t)>^2)}}$

The numerator is the covariance of $r_i(t)$ and $r_j(t)$ and the denominator is the product of the SD of $r_i(t)$ and $r_j(t)$. Therefore, it's the same as Pearson's correlation coefficient, which means the upper bound is 1 and the lower bound is -1.

The reasons why we use the log-normalized return instead of regular return:

1.We constraint values within finite bounds, so we can reduce the effects of outliers and variance in the data.

2.Reduces any skewness that may be present in the data.

3.Amplifies the scale of smaller stocks and shrinks the scale of larger stocks, thus providing a homogenous scale to visualize the relative changes of all stocks regardless of their prices.

3 Q2

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

length_data<-c()
```

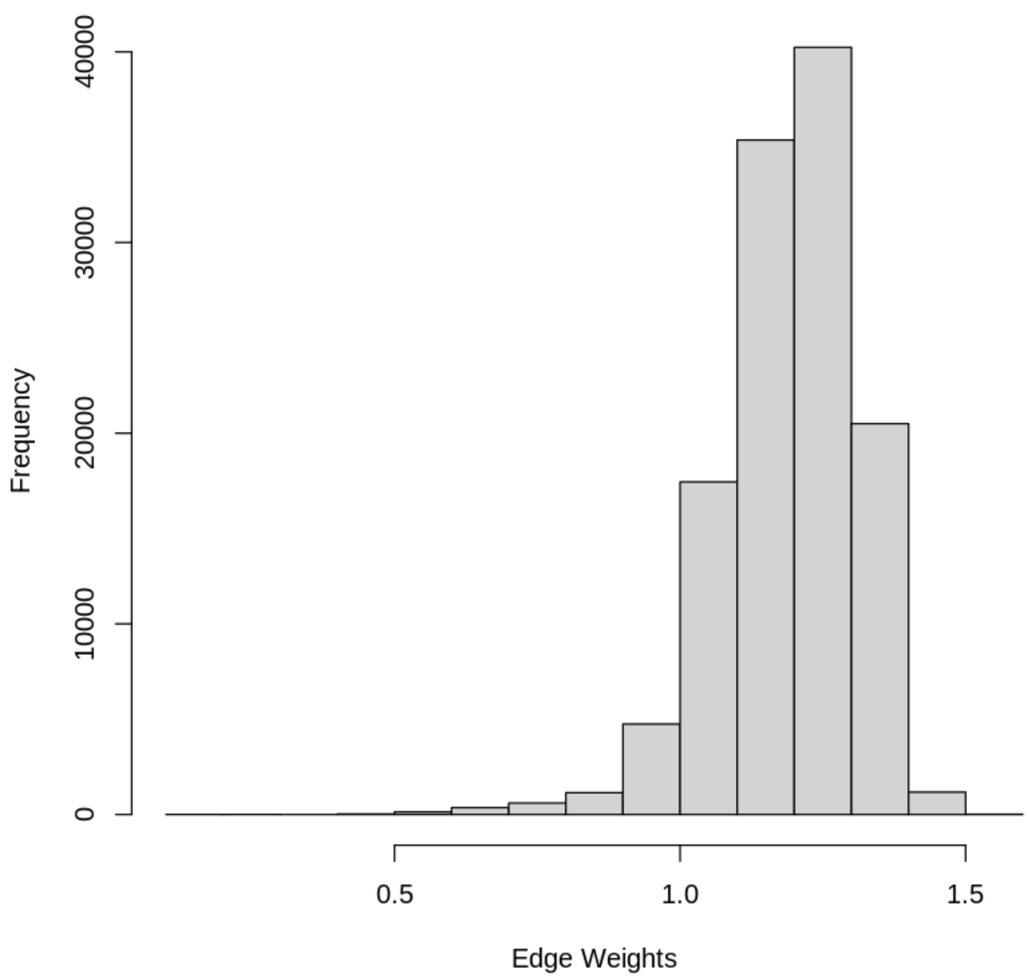
```
i<-1
log_norm_mat = matrix(0,length(filenames)-11,764)
for(j in c(1:length(filenames))){
    df = read.csv(filenames[j],header=TRUE, stringsAsFactors=FALSE)
    length_data[j] = dim(df)[1]
    if(length_data[j]==765){
        p = df[,5]
        q = c()
        r = c()
        for(k in c(2:length(p))){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

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

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

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

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

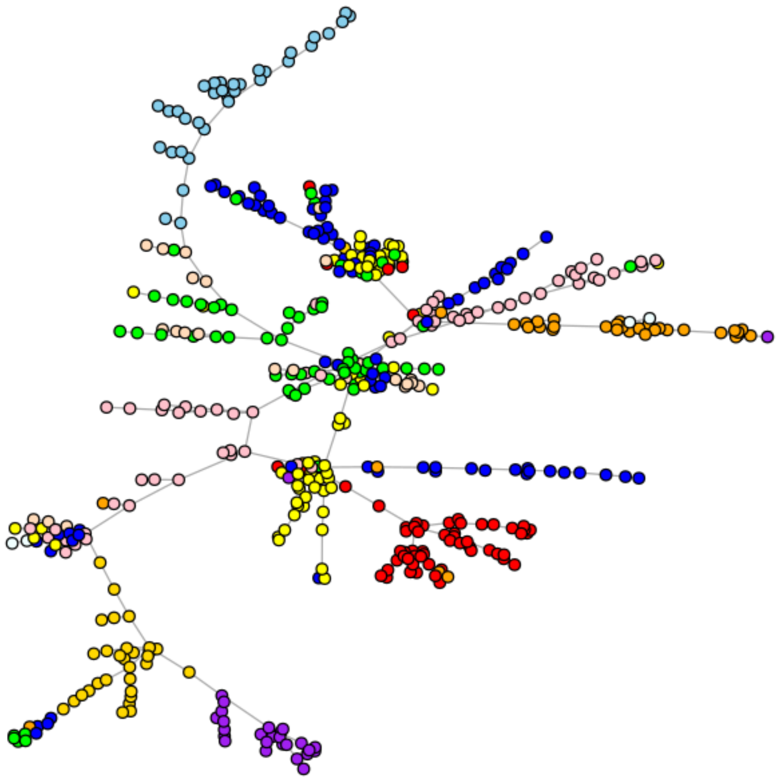
Histogram showing the un-normalized distribution of edge weights.



4 Q3

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

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

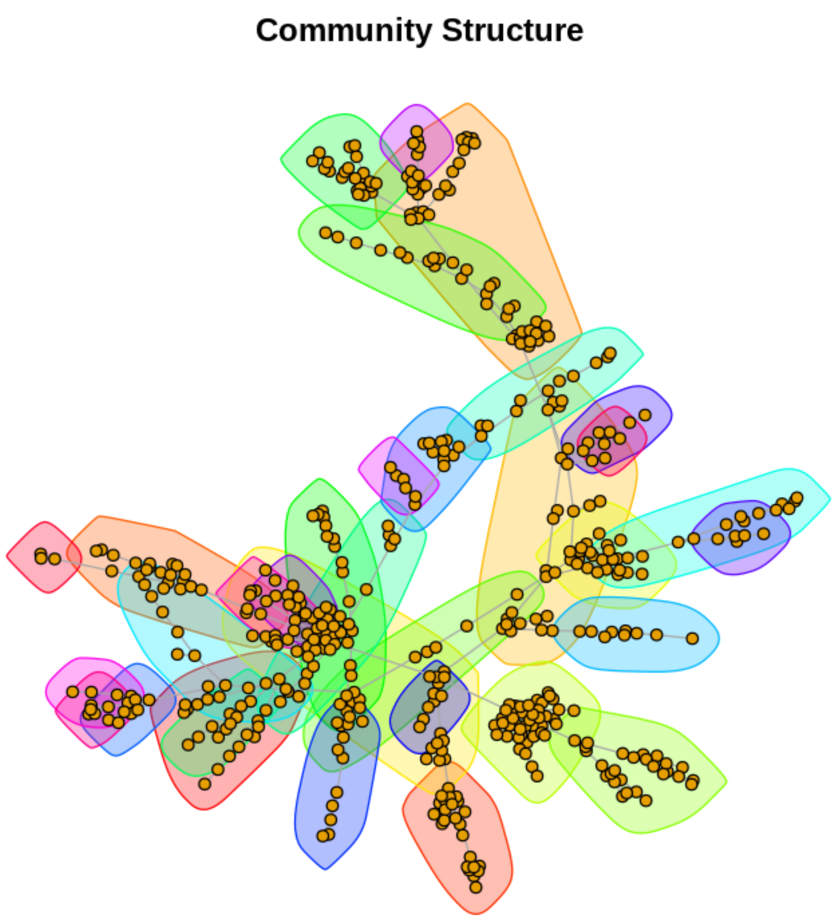
```r
plot(mst,vertex.size=3, vertex.label=NA, vertex.color=colors)
```

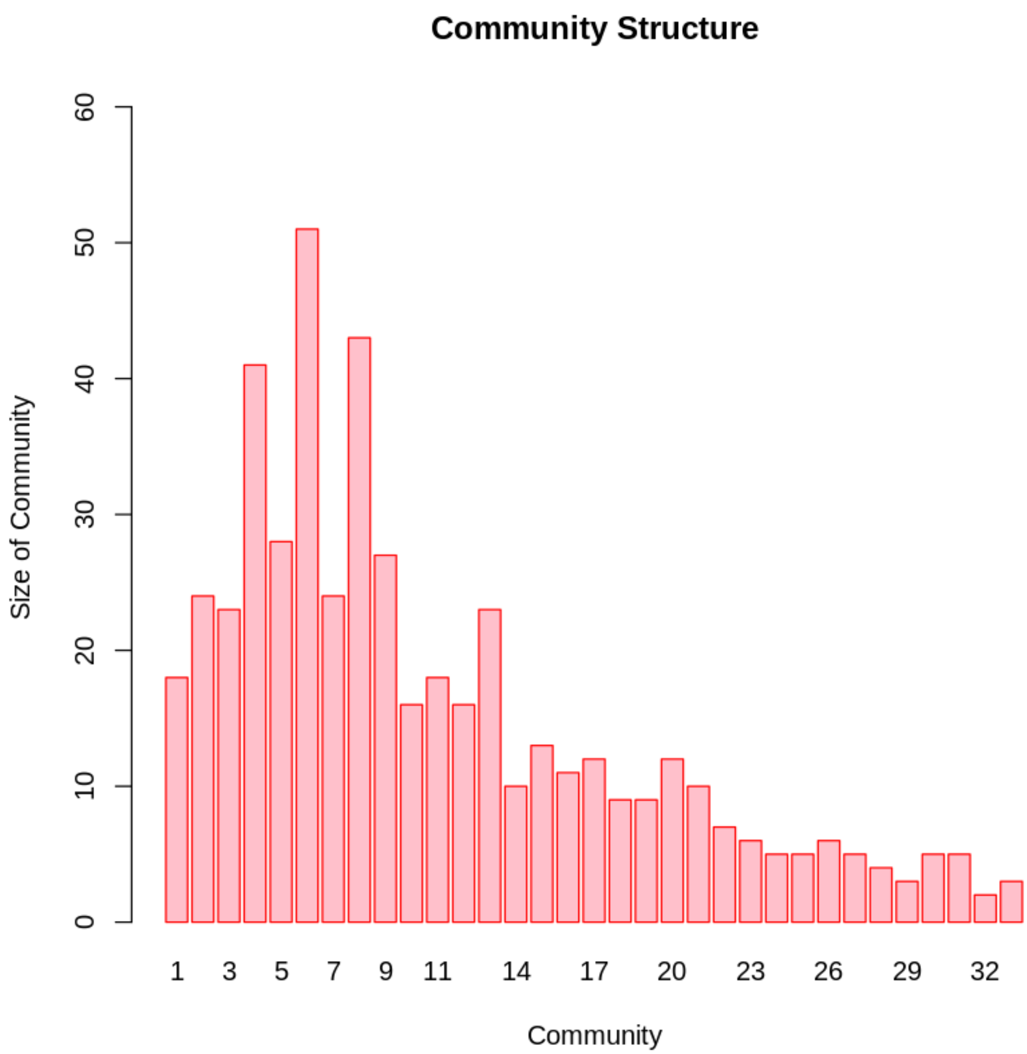


According to the figure above, most of the stocks that have a common color tend to flock together in the MST. That is to say, nodes of different colors (belonging to different sectors) are not connected to each other. In other words, stocks that are highly correlated tend to be connected on the graph with the least possible edge weights (the higher the correlation between stocks). This structure is referred to as vine clusters, as they look like grapes hanging off a main branch. The clusters represent distinct sectors. Stocks belonging to the same cluster tend to have changes in the same direction and thus require similar investment strategies.

5 Q4

[ ]:
```
comm <- walktrap.community(mst)
plot(mst, mark.groups = groups(comm), vertex.size=3, vertex.
 ↪label="",main="Community Structure")
barplot(as.vector(sizes(comm)),names.arg =␣
 ↪seq(1,length(comm),1),main="Community Structure",
  xlab="Community",ylab="Size of Community",ylim=c(0,max(as.
 ↪vector(sizes(comm)))+10),border="red",col="pink")
```





[ ]:
```
N = 494
Si <- c()
for(i in c(1:length(sectors))){
  Si[i] <- length(which(tickers_sectors[,2]==sectors[i]))
}
C = Si
K <- c()
for(i in c(1:length(comm))){
  K[i] = length(V(mst)$name[which(comm$membership == i)])
}
hc = 0
for(i in c(1:length(C))){
  hc = hc - ((C[i]/N) * log10(C[i]/N))
```

```
}
hk = 0
for(i in c(1:length(K))){
  hk = hk - ((K[i]/N) * log10(K[i]/N))
}
A = matrix(0,length(K),length(C))
for(i in c(1:length(K))){
  t <- V(mst)$name[which(comm$membership == i)]
  for(k in c(1:length(t))){
    v = substr(t[k],2,nchar(t[k])-1)
    p <- which(tickers_sectors[,1]==v)
    j <- which(sectors==tickers_sectors[p,2])
    A[i,j] = A[i,j] + 1
  }
}
hck = 0
for(j in c(1:length(K))){
  for(i in c(1:length(C))) {
    if(A[j,i]!=0){
      hck = hck - ((A[j,i]/N) * log10(A[j,i]/K[j]))
    }
  }
}
hkc = 0
for(i in c(1:length(C))){
  for(j in c(1:length(K))) {
    if(A[j,i]!=0){
      hkc = hkc - ((A[j,i]/N) * log10(A[j,i]/C[i]))
    }
  }
}
h = 1 - hck/hc
c = 1 - hkc/hk
print(sprintf("homogeneity: %f, completeness: %f",h,c))
```

```
[1] "homogeneity: 0.682645, completeness: 0.479284"
```

6 Q5

[ ]:
```
p1 <- c()
p2 <- c()
for(v in c(1:vcount(mst))){
  neighbors <- neighbors(mst,v)
  Ni <- length(neighbors)
  Qi<-0
  for(i in neighbors){
```

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

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

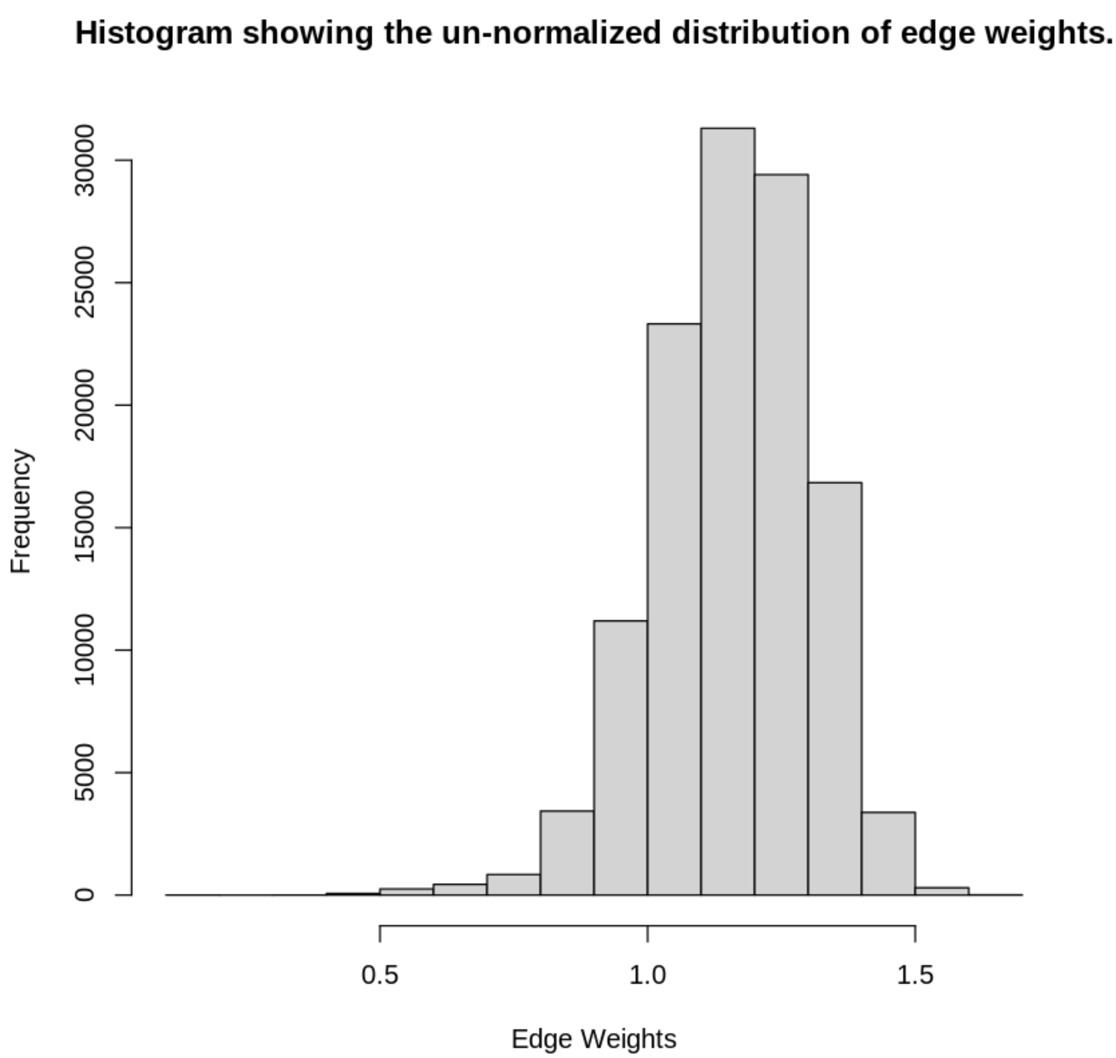
We can see that the first case has a higher value of than the second case. This is reasonable because the first case exploits the MST vine, considering which nodes are highly correlated and flock together instead of all the nodes. That is to say, the first case exploits local connectivity among neighboring nodes instead of the global correlation graph to make decisions. On the other hand, the second case considers all the nodes that belong to a sector and fails to extract local spatial connections or cluster formations by taking into account the entire graph. The second case thus provides only a general probability estimate.

7 Q6

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

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

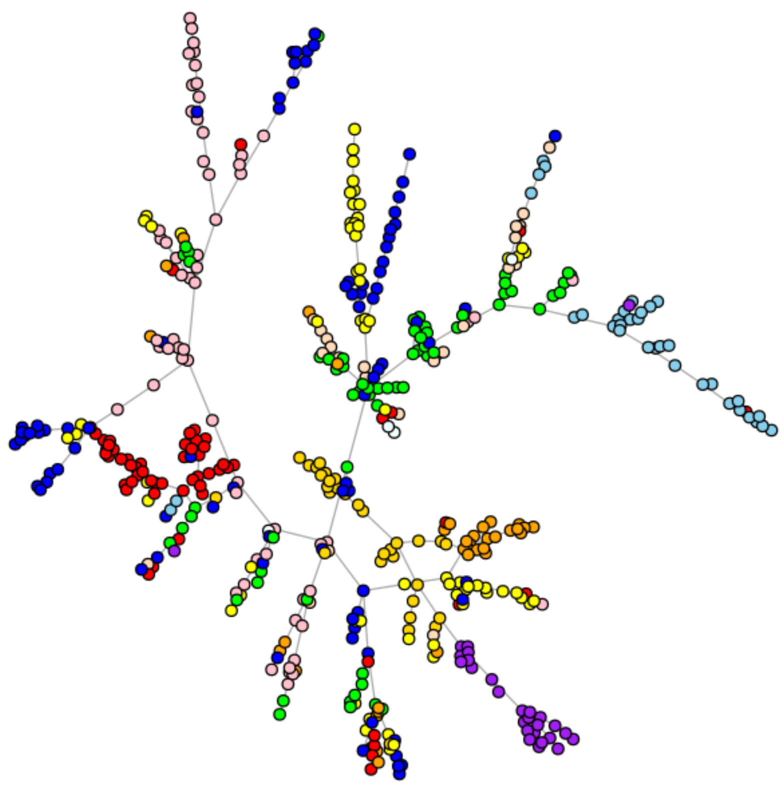
```
[ ]: tickers_sectors_week=tickers_sectors_week[-which(length_data_week!=143),]
     edge_weight_file_week <- file("finance_data/edge_weights_week.txt", "w")
     get_edges(edge_weight_file_week,log_norm_mat_week,tickers_sectors_week)
     close(edge_weight_file_week)

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

```
[ ]: hist(edge_list_week[,"Weight"],main="Histogram showing the un-normalized␣
      ↪distribution of edge weights.",xlab="Edge Weights",ylab="Frequency")
```



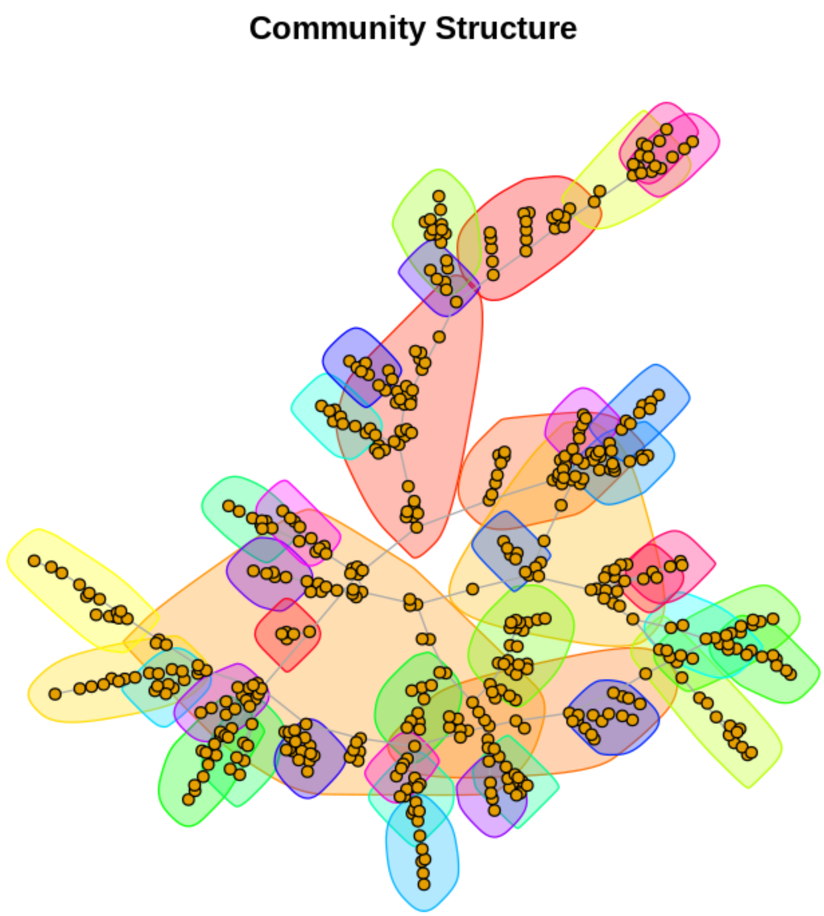
[ ]:

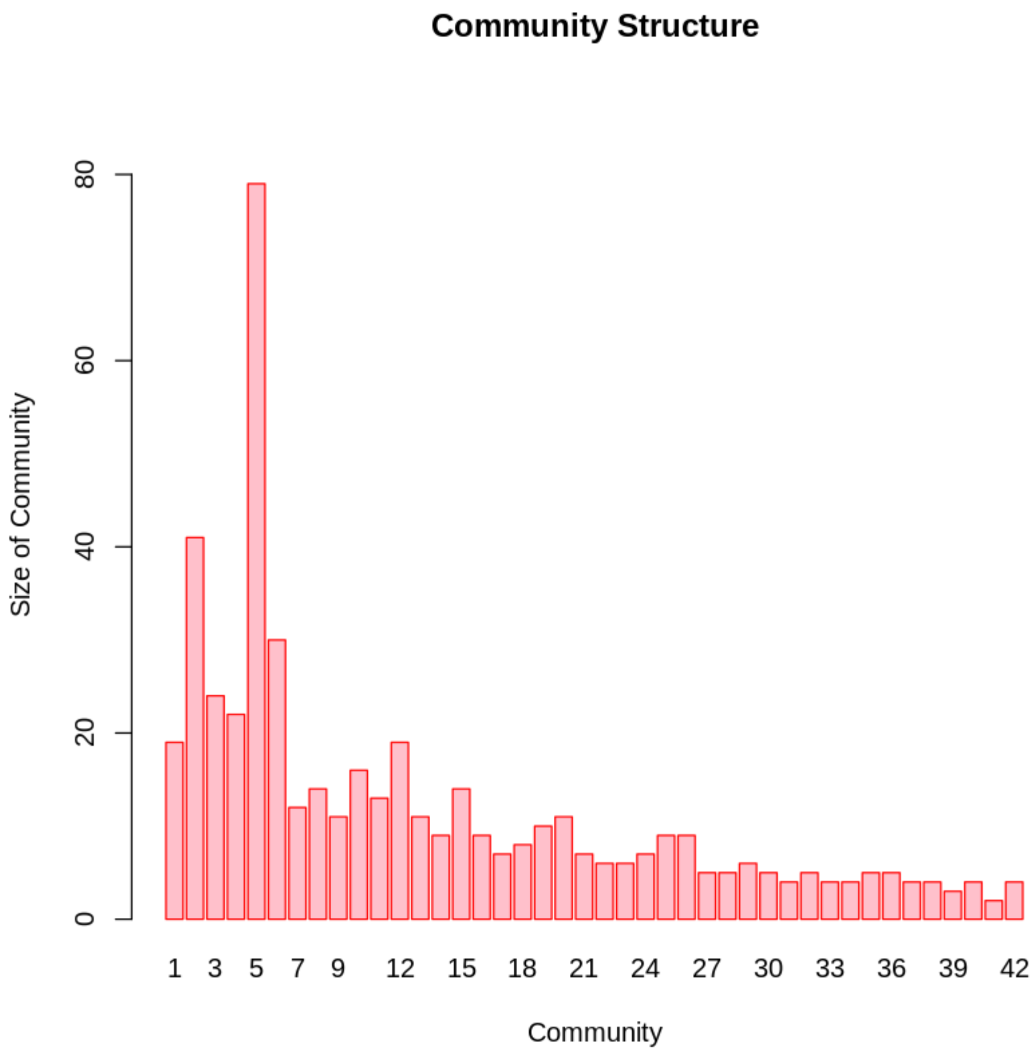
```
mst_week <- mst(correlation_graph_week,algorithm="prim")
sectors_week = unique(tickers_sectors_week[,2])

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

plot(mst_week,vertex.size=3, vertex.label=NA, vertex.color=colors_week)
```



According to the figure above, some of the stocks are still forming vine clusters. However, some stocks belonging to the same sector are not forming clusters, with the nodes not forming clearly separable regions in the MST graph. This indicates that clustering is better with daily data than weekly data. In other words, stocks from the same sector lose their correlation when the time-scale increases, leading to the edge weights in the correlation graph to increase among them even though they belong to the same node.

[ ]:
```
comm_week <- walktrap.community(mst_week)
plot(mst_week, mark.groups = groups(comm_week), vertex.size=3, vertex.
 ↪label="",main="Community Structure")
barplot(as.vector(sizes(comm_week)),names.arg =␣
 ↪seq(1,length(comm_week),1),main="Community Structure",
  xlab="Community",ylab="Size of Community",ylim=c(0,max(as.
 ↪vector(sizes(comm_week))+10),border="red",col="pink")
```






```
[ ]: N = 492
     Si <- c()
     for(i in c(1:length(sectors_week))){
       Si[i] <- length(which(tickers_sectors_week[,2]==sectors_week[i]))
     }
     C = Si
     K <- c()
     for(i in c(1:length(comm_week))){
       K[i] = length(V(mst_week)$name[which(comm_week$membership == i)])
     }
     hc = 0
```

```
for(i in c(1:length(C))){
  hc = hc - ((C[i]/N) * log10(C[i]/N))
}
hk = 0
for(i in c(1:length(K))){
  hk = hk - ((K[i]/N) * log10(K[i]/N))
}
A = matrix(0,length(K),length(C))
for(i in c(1:length(K))){
  t <- V(mst_week)$name[which(comm_week$membership == i)]
  for(k in c(1:length(t))){
    v = substr(t[k],2,nchar(t[k])-1)
    p <- which(tickers_sectors_week[,1]==v)
    j <- which(sectors==tickers_sectors_week[p,2])
    A[i,j] = A[i,j] + 1
  }
}
hck = 0
for(j in c(1:length(K))){
  for(i in c(1:length(C))) {
    if(A[j,i]!=0){
      hck = hck - ((A[j,i]/N) * log10(A[j,i]/K[j]))
    }
  }
}
hkc = 0
for(i in c(1:length(C))){
  for(j in c(1:length(K))) {
    if(A[j,i]!=0){
      hkc = hkc - ((A[j,i]/N) * log10(A[j,i]/C[i]))
    }
  }
}
h = 1 - hck/hc
c = 1 - hkc/hk
print(sprintf("homogeneity: %f, completeness: %f",h,c))
```

```
[1] "homogeneity: 0.581124, completeness: 0.390044"
```

[ ]:

```
p1 <- c()
p2 <- c()
for(v in c(1:vcount(mst_week))){
  neighbors <- neighbors(mst_week,v)
  Ni <- length(neighbors)
  Qi<-0
  for(i in neighbors){
    if(tickers_sectors_week[i,2]==tickers_sectors_week[v,2])
```

```
      Qi<-Qi+1
  }
  p1[v] <- Qi/Ni
  p2[v] <- Si[which(sectors_week==tickers_sectors_week[v,2])]/vcount(mst_week)
}
alpha1 <- sum(p1)/vcount(mst_week)
alpha2 <- sum(p2)/vcount(mst_week)
print(sprintf("Values of alpha for the two cases: %f and %f",alpha1,alpha2))
```

```
[1] "Values of alpha for the two cases: 0.743957 and 0.114309"
```

We can see that the alpha value decreases for the first case. This is expected since the clustering is not as strong as the daily data. As for the second case, it just provides a general probability estimate. Hence, the value is similar to the daily data.

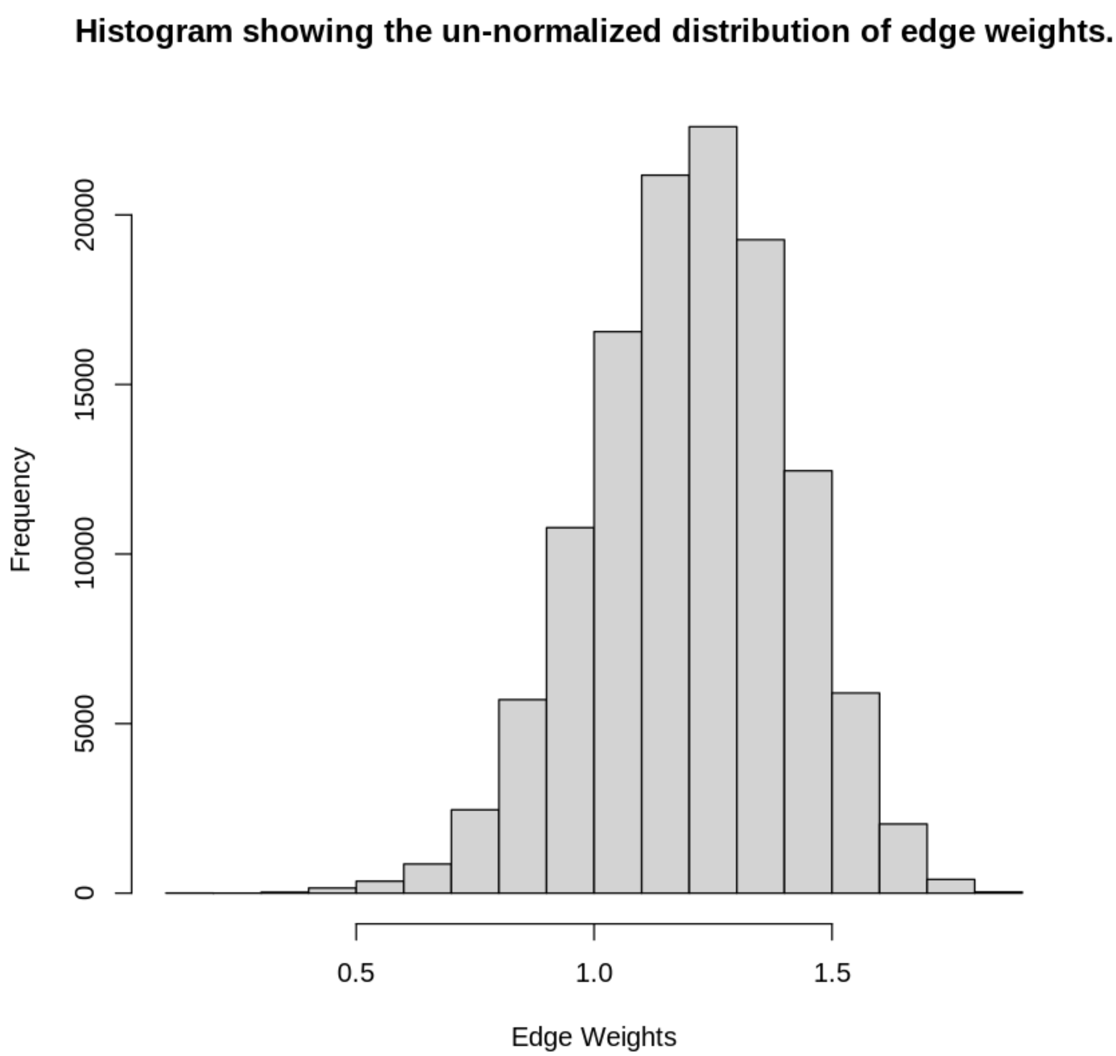
8 Q7

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

length_data_month<-c()
i<-1
log_norm_mat_month = matrix(0,length(filenames_week)-13,24) #omit files with␣
 ↪NaN data
for(j in c(1:length(filenames_month))){
    df = read.csv(filenames_month[j],header=TRUE, stringsAsFactors=FALSE)
    df["Day"] = substring(df[,1],9,10)
    df = subset(df, Day=='15')
    length_data_month[j] = dim(df)[1]
    if(length_data_month[j]==25){
        p = df[,5]
        q = c()
        r = c()
        for(k in c(2:length(p))){
            q[k-1] = (p[k]-p[k-1])/p[k-1]
        }
        r = log(1+q)
        log_norm_mat_month[i,] = r
        i = i+1
    }
}
```

[ ]:
```
tickers_sectors_month=tickers_sectors_month[-which(length_data_month!=25),]
edge_weight_file_month <- file("finance_data/edge_weights_month.txt", "w")
get_edges(edge_weight_file_month,log_norm_mat_month,tickers_sectors_month)
```

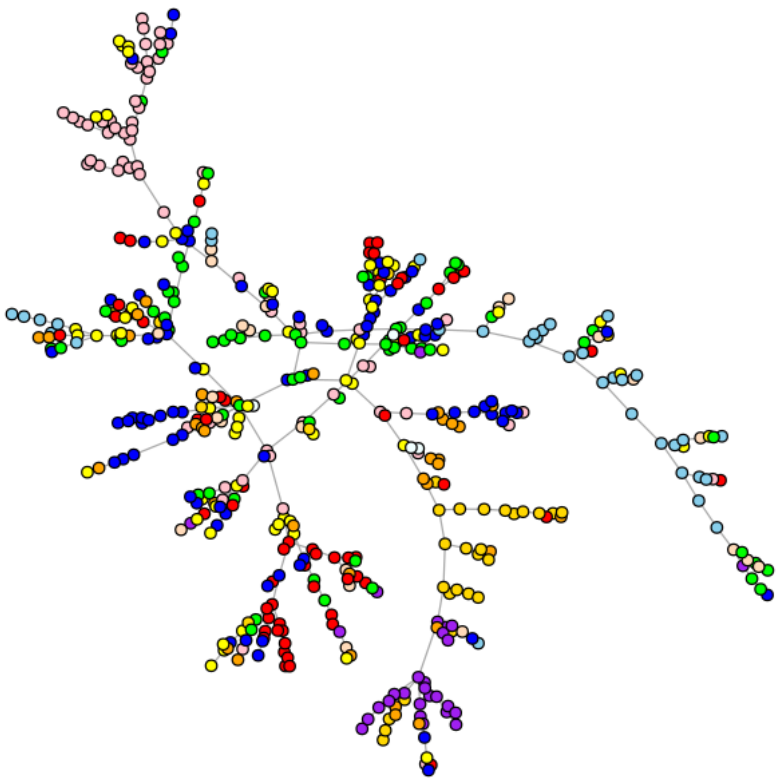
```
close(edge_weight_file_month)

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

[ ]:
```
hist(edge_list_month[,"Weight"],main="Histogram showing the un-normalized␣
↪distribution of edge weights.",xlab="Edge Weights",ylab="Frequency")
```



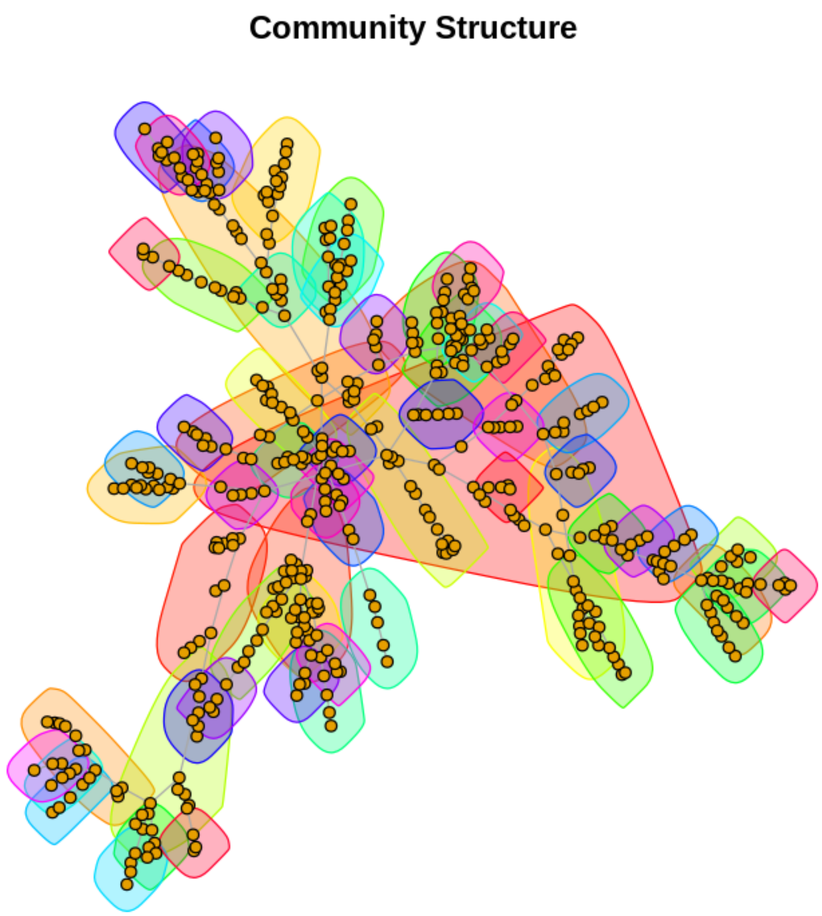
[ ]:
```
mst_month <- mst(correlation_graph_month,algorithm="prim")
sectors_month = unique(tickers_sectors_month[,2])

colors_month <- c()
```

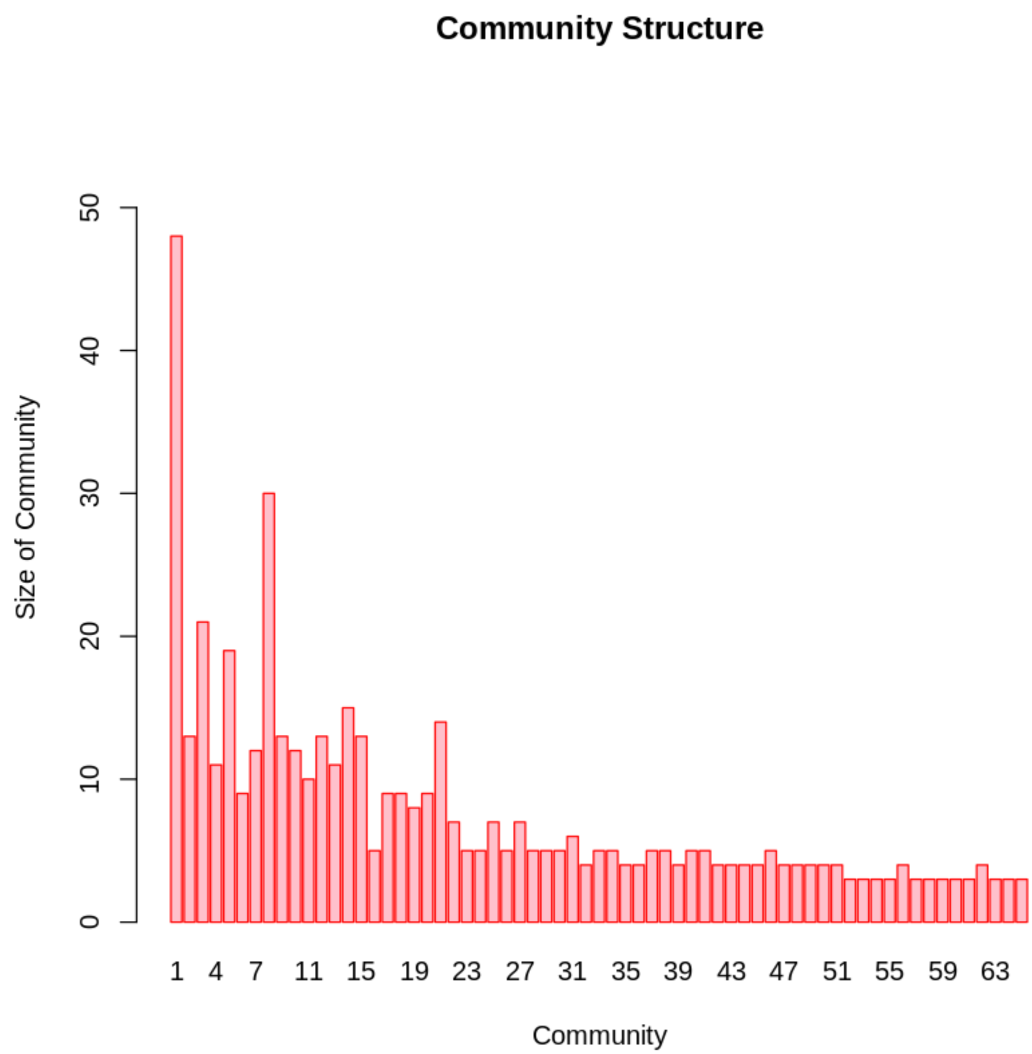
```
  for(v in c(1:vcount(correlation_graph_month))){
    cur_sector <- tickers_sectors_month[v,2]
    i <- which(sectors_month==cur_sector)
    colors_month[v] <-␣
  ↪switch(i,"red","green","blue","yellow","orange","purple","pink","gold","peachpuff","skyblue
}

plot(mst_month,vertex.size=3, vertex.label=NA, vertex.color=colors_month)
```



According to the figure above, some of the stocks are still forming vine clusters. However, some stocks belonging to the same sector are not forming clusters, with the nodes not forming clearly separable regions in the MST graph. In addition, it means that it will be more difficult to assign a sector to an unknown stock if the data is sampled monthly. We also observe a circular structure in the figure. The presence of loops in the MST indicate that most of the edge weights are similar regardless of the sector to which the stocks belong to.

[ ]:
```
comm_month <- walktrap.community(mst_month)
plot(mst_month, mark.groups = groups(comm_month), vertex.size=3, vertex.
 ↪label="",main="Community Structure")
barplot(as.vector(sizes(comm_month)),names.arg =␣
 ↪seq(1,length(comm_month),1),main="Community Structure",
   xlab="Community",ylab="Size of Community",ylim=c(0,max(as.
 ↪vector(sizes(comm_month)))+10),border="red",col="pink")
```





[ ]:
```
N = 492
Si <- c()
for(i in c(1:length(sectors_month))){
  Si[i] <- length(which(tickers_sectors_month[,2]==sectors_month[i]))
}
C = Si
K <- c()
for(i in c(1:length(comm_month))){
  K[i] = length(V(mst_month)$name[which(comm_month$membership == i)])
}
hc = 0
for(i in c(1:length(C))){
  hc = hc - ((C[i]/N) * log10(C[i]/N))
```

```
}
hk = 0
for(i in c(1:length(K))){
  hk = hk - ((K[i]/N) * log10(K[i]/N))
}
A = matrix(0,length(K),length(C))
for(i in c(1:length(K))){
  t <- V(mst_month)$name[which(comm_month$membership == i)]
  for(k in c(1:length(t))){
    v = substr(t[k],2,nchar(t[k])-1)
    p <- which(tickers_sectors_month[,1]==v)
    j <- which(sectors==tickers_sectors_month[p,2])
    A[i,j] = A[i,j] + 1
  }
}
hck = 0
for(j in c(1:length(K))){
  for(i in c(1:length(C))) {
    if(A[j,i]!=0){
      hck = hck - ((A[j,i]/N) * log10(A[j,i]/K[j]))
    }
  }
}
hkc = 0
for(i in c(1:length(C))){
  for(j in c(1:length(K))) {
    if(A[j,i]!=0){
      hkc = hkc - ((A[j,i]/N) * log10(A[j,i]/C[i]))
    }
  }
}
h = 1 - hck/hc
c = 1 - hkc/hk
print(sprintf("homogeneity: %f, completeness: %f",h,c))
```

```
[1] "homogeneity: 0.479447, completeness: 0.277551"
```

[ ]:
```
p1 <- c()
p2 <- c()
for(v in c(1:vcount(mst_month))){
  neighbors <- neighbors(mst_month,v)
  Ni <- length(neighbors)
  Qi<-0
  for(i in neighbors){
    if(tickers_sectors_month[i,2]==tickers_sectors_month[v,2])
      Qi<-Qi+1
  }
```

```r
  p1[v] <- Qi/Ni
  p2[v] <- Si[which(sectors_month==tickers_sectors_month[v,2])]/
 ↪vcount(mst_month)
}
alpha1 <- sum(p1)/vcount(mst_month)
alpha2 <- sum(p2)/vcount(mst_month)
print(sprintf("Values of alpha for the two cases: %f and %f",alpha1,alpha2))
```

```
[1] "Values of alpha for the two cases: 0.484446 and 0.114309"
```

We can see that the alpha value decreases for the first case again compared to weekly data. This is expected since the clustering is not as strong as the weekly data. As for the second case, it just provides a general probability estimate. Hence, the value is similar to the weekly data.

9 Q8

Similarity:

For the alpha value calculated by the second case, three kinds of data are roughly the same, which makes sense since just provides a general probability estimate.

Difference:

1. MST of the correlation graph are different for each data. For daily data, most of the stocks that have a common color (belong to the same sector) tend to flock together in the MST. For weekly data, although some of the stocks are still forming vine clusters, a significant number of stocks belonging to the same sector are not forming clusters, with the nodes not forming clearly separable regions in the MST graph. This phenomenon is more obvious in the monthly data. This is because stocks from the same sector lose their correlation when the time-scale increases, causing the edge weights in the correlation graph to increase among them even though they belong to the same node.

2. We can see that the both homogeneity and completeness of MST is daily>weekly>monthly. The reason is that clustering is better with daily data.

3. For the alpha value calculated by the first case is daily>weekly>monthly. The reason is that clustering is better with daily data.

In a nutshell, daily data gives the best results when predicting the sector of an unknown stock according to the above results. The MST of the correlation graph has the highest alpha value calculated by the first case, homogeneity and completeness.

Project4_part2

June 10, 2022

1 Initialization

[ ]:
```
from google.colab import drive
drive.mount('/content/drive')
import sys
import os
path_to_module = '/content/drive/MyDrive/Project4'
sys.path.append(path_to_module)
os.chdir(path_to_module)
```

```
Mounted at /content/drive
```

[ ]:
```
!pip install pandas
!pip install igraph
!pip install cairocffi

!apt-get install libcairo2-dev libjpeg-dev libgif-dev
!pip install pycairo
```

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

2 Q9

[ ]:
```
with open('/content/drive/MyDrive/Project4/los_angeles_censustracts.json') as f:
    census_tracts = json.loads(f.readline())
```

[ ]:
```
display_names = dict()
coordinates = dict()
for area in census_tracts['features']:
    id = int(area['properties']['MOVEMENT_ID'])
    display_name = area['properties']['DISPLAY_NAME']
    display_names[id] = display_name
    a = area['geometry']['coordinates'][0]
    coordinates[id] = np.array(a if type(a[0][0]) == float else a[0]).mean(axis␣
↪= 0)
```

[ ]:
```
g = Graph(directed = False)
g.add_vertices(len(display_names))
g.vs['display_name'] = list(display_names.values()) # index = id - 1
g.vs['coordinates'] = list(coordinates.values())
```

[ ]:
```
month_filter = {12}

edges = []
weights = []
travel_time_dict = {}

with open('/content/drive/MyDrive/Project4/
↪los_angeles-censustracts-2019-4-All-MonthlyAggregate.csv') as f:
    f.readline()
    while True:
        line = f.readline()
        if line == '':
            break
        vals = line.strip().split(',')
        src, dest, month, dist = int(vals[0]), int(vals[1]), int(vals[2]),␣
↪float(vals[3])
        if month not in month_filter:
            continue
        edges.append((src - 1, dest - 1))
        weights.append(dist)
        travel_time_dict[(src - 1, dest - 1)]=dist
```

[ ]:
```
g.add_edges(edges)
g.es['weight'] = weights
del edges, weights
```

[ ]:
```python
components = g.components()
gcc = max(components, key = len)
vs_to_delete = [i for i in range(len(g.vs)) if i not in gcc]
g.delete_vertices(vs_to_delete)

g = g.simplify(combine_edges = dict(weight = 'mean'))
```

[ ]:
```python
print("Number of nodes and edges: {}, {}".format(len(g.vs), len(g.es)))
```

```
Number of nodes and edges: 2649, 1003858
```

3 Q10

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

Q10_ig_plot.png

[ ]:
```python
edf = mst.get_edge_dataframe()
edf.head()
```

[ ]:

edge ID	source	target	weight
0	0	2	129.765
1	0	13	118.335
2	1	2	90.235
3	1	3	126.475
4	1	9	125.675

[ ]:
```python
for i, e in enumerate(mst.es):
    print('Distance in miles: {:.3f}, Time taken: {:.
↪1f}\n-------------------------------------------'.format(linalg.norm(mst.
↪vs[e.source]['coordinates']-mst.vs[e.target]['coordinates'])*69,␣
↪e['weight']))
    if i > 10:
        break
```

```
Distance in miles: 0.885, Time taken: 129.8
-------------------------------------------
Distance in miles: 0.570, Time taken: 118.3
-------------------------------------------
Distance in miles: 0.447, Time taken: 90.2
```

```
------------------------------------------
Distance in miles: 0.621, Time taken: 126.5
------------------------------------------
Distance in miles: 0.812, Time taken: 125.7
------------------------------------------
Distance in miles: 0.618, Time taken: 119.9
------------------------------------------
Distance in miles: 0.936, Time taken: 125.2
------------------------------------------
Distance in miles: 0.412, Time taken: 91.8
------------------------------------------
Distance in miles: 0.256, Time taken: 60.9
------------------------------------------
Distance in miles: 0.204, Time taken: 87.1
------------------------------------------
Distance in miles: 0.620, Time taken: 110.9
------------------------------------------
Distance in miles: 0.493, Time taken: 162.3
------------------------------------------
```

To report the street address, since we got the location of source, location of target, centroid location=((source location)+(target location))/2, and hence the coordinates, we can get the following website to get the street address for the coordinates: https://www.gps-coordinates.net/

Above edge length were converted from coordinates to miles approximated by converting each degree of latitude to 69 miles. California states 25mph as the speed limit on residential speed and 65 mph on freeways. Taking the first weight as an example, 0.885 miles is travelled in 129.8 seconds, which equates to 24.5mph. This coincides with the residential speed limit and we have verified that these numbers are indeed intuitive, especially with the added effect of notorious LA traffic.

4 Q11

[ ]:
```
triangles = []

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

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

```
        counter += 1
```

[ ]:
```
print(counter / len(triangles))
```

0.928

5 Q12

[ ]:
```
df = pd.read_csv('./los_angeles-censustracts-2019-4-All-MonthlyAggregate.csv',␣
 ↪usecols=['sourceid', 'dstid', 'mean_travel_time', 'month'])
df = df[df['month']==12][['sourceid', 'dstid', 'mean_travel_time']]
gd = df.values
arr = gd

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

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

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

costs, cur_paths = [], []
for vertex in vertices:
    tour = [u for u, v in nx.eulerian_circuit(mg, source=vertex)]
    cur_path, visited_nodes = [], set()
    for i in tour:
```

```
        if i not in visited_nodes:
            cur_path.append(i)
            visited_nodes.add(i)
    cur_path.append(cur_path[0])
    cur_paths.append(cur_path)

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

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

[ ]:
```
print('MST cost: {}'.format(mst_cost))
print('Approximate cost: {}'.format(min_approx_cost))
print('Upper bound: {}'.format(min_approx_cost/mst_cost))
```

```
MST cost: 269084.54500000016
Approximate cost: 421489.3149999998
Upper bound: 1.5663824728395292
```

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

Normalizing all the costs by dividing everything with MST cost, we get the following inequality: $1 \leq \frac{Optimal\ TSP\ cost}{MST\ cost} \leq \rho \leq 2$

And, the value of ρ we obtained was 1.5663824728395292

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

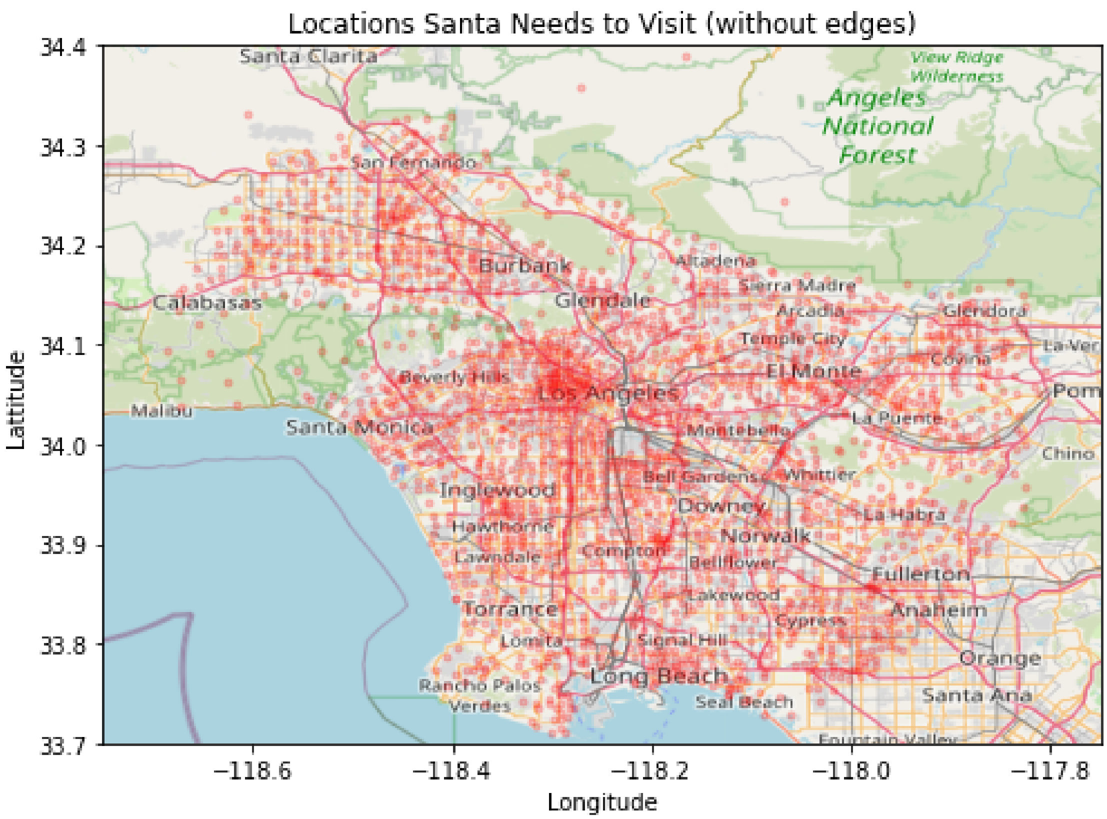
6 Q13

[ ]:
```
data = json.load(open('./los_angeles_censustracts.json'))
location_data = []

for i in trajectory:
    for j in range(len(data['features'])):
        if data['features'][j]['properties']['MOVEMENT_ID'] == str(int(i)):
            cur_loc = data['features'][j]['geometry']['coordinates'][0]
```

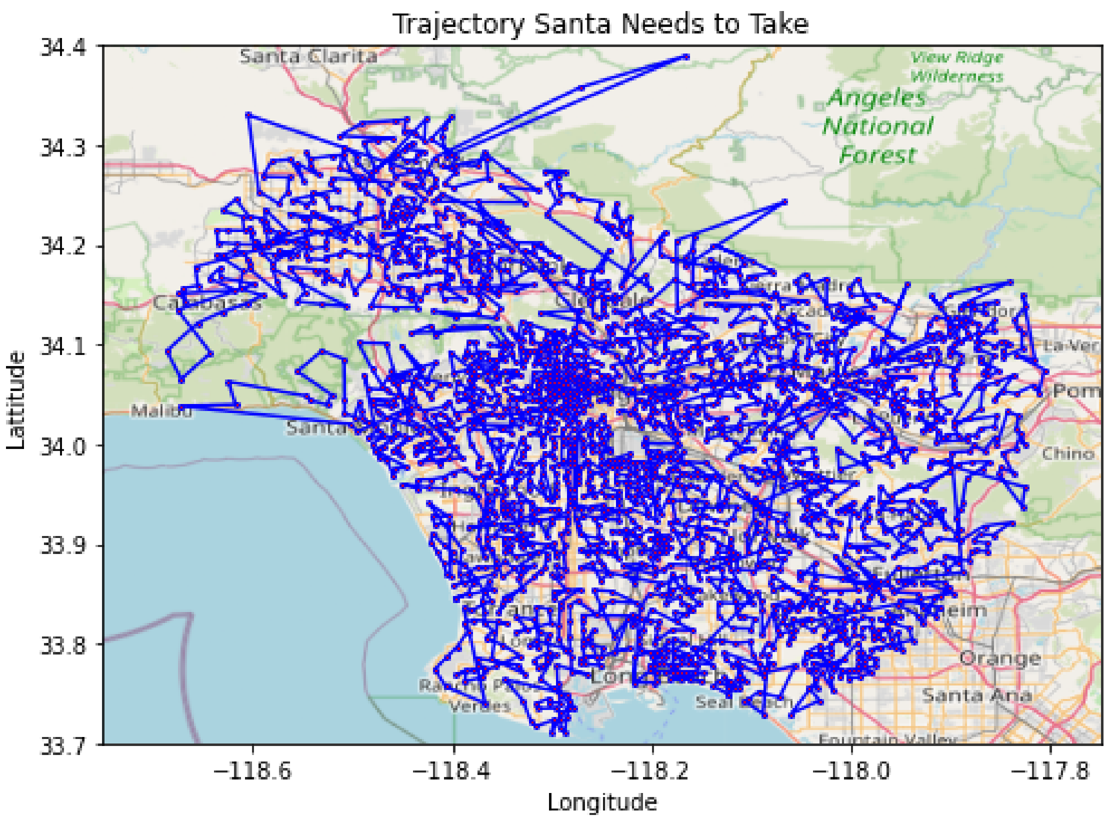
```
            if len(cur_loc) == 1:
                t = np.asarray(cur_loc[0]).mean(axis = 0)
                location_data.append(t)
            elif len(cur_loc) == 2:
                t = np.asarray(cur_loc[0]+cur_loc[1]).mean(axis = 0)
                location_data.append(t)
            elif i ==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

x, y = [i[0] for i in location_data], [i[1] for i in location_data]
```

[ ]:
```
BBox = ((-118.75, -117.75, 33.7, 34.4))
```

[ ]:
```
ruh_m = plt.imread('./map_LA.png')

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



[ ]:
```
ruh_m = plt.imread('./map_LA.png')

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



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

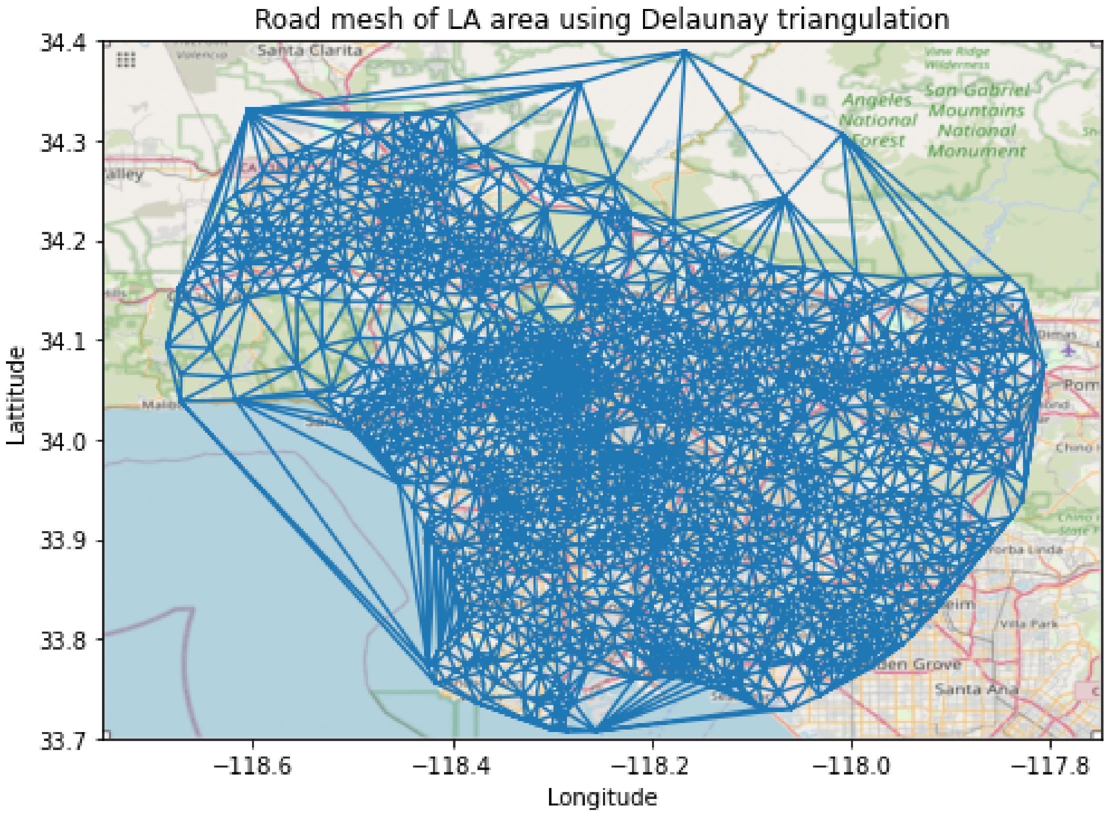
```
( -118.12911933333332 , 34.08759475 )
( -118.13138209090911 , 34.09626386363636 )
( -118.13785063157897 , 34.09645121052631 )
( -118.13224544444446 , 34.10349303174603 )
( -118.14492316666666 , 34.098681500000005 )
( -118.15023891071432 , 34.09595766071429 )
( -118.15266638571427 , 34.09029572857144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

7 Q14

[ ]:
```
lat_long = {}

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

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

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

f.close()

lat = []
lon = []

for i in range(1, len(lat_long)+1):
    lat.append(lat_long[str(i)][1])
    lon.append(lat_long[str(i)][2])

lat_lon = tuple(zip(lat, lon))
delaunay_out = Delaunay(lat_lon)
```

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

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



We can observe that the triangulation algorithm has extracted almost the road structures of Los Angeles, especially in the downtown.
However, we also see roads going over non-existent roads. It is because the nature of the DT algorithm, which tries to avoid generating silver triangles or very narrow triangles by ensuring no point in the set of points lies within the circumcircle of any triangle. That is, DT algorithm tries to fit every triangle inside a circumcircle. It is clear from the plot below, where the nodes represent the locations and the edges are produced by triangulation. Most of the polygons don't have extremely small acute angles.

[ ]:
```
g_del = ig.Graph()
g_del.add_vertices(len(delaunay_out.points))
remove_duplicates = set()
weights_del = []

for i in range(len(delaunay_out.simplices)):
    a = ((delaunay_out.simplices[i][0], delaunay_out.simplices[i][1]))
    b = ((delaunay_out.simplices[i][0], delaunay_out.simplices[i][2]))
    c = ((delaunay_out.simplices[i][1], delaunay_out.simplices[i][2]))

    list(a).sort()
    list(b).sort()
```

```
    list(c).sort()

    if not a in remove_duplicates:
        remove_duplicates.add(a)
        g_del.add_edges([a])
        weight = 69*np.
↪sqrt((lat_lon[a[0]][0]-lat_lon[a[1]][0])**2+((lat_lon[a[0]][1]-lat_lon[a[1]][1])**2))
        weights_del.append(weight)
    if not b in remove_duplicates:
        remove_duplicates.add(b)
        g_del.add_edges([b])
        weight = 69*np.
↪sqrt((lat_lon[b[0]][0]-lat_lon[b[1]][0])**2+((lat_lon[b[0]][1]-lat_lon[b[1]][1])**2))
        weights_del.append(weight)
    if not c in remove_duplicates:
        remove_duplicates.add(c)
        g_del.add_edges([c])
        weight = 69*np.
↪sqrt((lat_lon[c[0]][0]-lat_lon[c[1]][0])**2+((lat_lon[c[0]][1]-lat_lon[c[1]][1])**2))
        weights_del.append(weight)
```

[ ]:
```
#weights are distance in miles
g_del.es['weight'] = weights_del
```

[ ]:
```
print("Number of vertices: {}".format(len(g_del.vs())))
print("Number of edges: {}".format(len(g_del.es())))
```

```
Number of vertices: 2716
Number of edges: 10823
```

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

Q14_ig_plot.png

8 Q15

- $Total\ Distance = \frac{Velocity\ of\ Car \times Mean\ Travel\ Time}{60 \times 60}$ (divided by 3600 means to convert from seconds to hours)
- $Gap = 0.003 + \frac{2 \times Velocity\ of\ Car}{60 \times 60}$ (to accommodate for length of each car and the distance between the cars)
- $Total\ Numbers\ of\ Cars\ on\ the\ Road = \frac{2 \times Total\ Distance}{Gap}$ (multiplied by 2 to accommodate flow in both directions)

- $Traffic\ Flow\ (cars/hour) = \frac{60 \times 60}{Mean\ Travel\ Time} \times Total\ Number\ of\ Cars\ on\ the\ Road$

Simplifying it, we can get:
$Traffic\ Flow\ (cars/hour) = \frac{3600 \times Velocity\ of\ Car}{5.4 + Velocity\ of\ Car}$

The unit of velocity is in miles per hour

From Pythagoras Theorem, the velocity of car between two coordinates is given as:
$Velocity\,of\,Car = \frac{69}{Mean\ Travel\ Time} \times \sqrt{(Longitude_{point\ 1} - Longitude_{point\ 2})^2 + (Latitude_{point\ 1} - Latitude_{point}$

9 Q16

```
[ ]: malibu = [34.026, -118.78]
     long_beach = [33.77, -118.18]

     vcar = (69*np.sqrt((malibu[0]-long_beach[0])**2 +␣
      ↪(malibu[1]-long_beach[1])**2)) / 1.05
     max_car_num = (3600*vcar) / (5.4+vcar)

     min_long_beach = np.inf
     min_malibu = np.inf
     long_beach_node = 0
     malibu_node = 0

     for i in range(1, len(lat_lon)):
         long_beach_closest = np.sqrt(((lat_lon[i][0])-long_beach[0])**2 +␣
      ↪((lat_lon[i][1])-long_beach[1])**2)
         malibu_closest = np.sqrt((malibu[0]-lat_lon[i][0])**2 +␣
      ↪(malibu[1]-lat_lon[i][1])**2)

         if long_beach_closest < min_long_beach:
             min_long_beach = long_beach_closest
             long_beach_node = i
         if malibu_closest < min_malibu:
             min_malibu = malibu_closest
             malibu_node = i
```

```
[ ]: #Distance between node closest to Malibu and node closest to LB
     print(69*np.sqrt((lat_lon[malibu_node][0]-lat_lon[long_beach_node][0])**2 +␣
      ↪(lat_lon[malibu_node][1]-lat_lon[long_beach_node][1])**2))
```

```
38.699315429177275
```

```
[ ]: flow = g_del.maxflow(malibu_node, long_beach_node)
     print("Max flow: {}".format(flow.value))
```

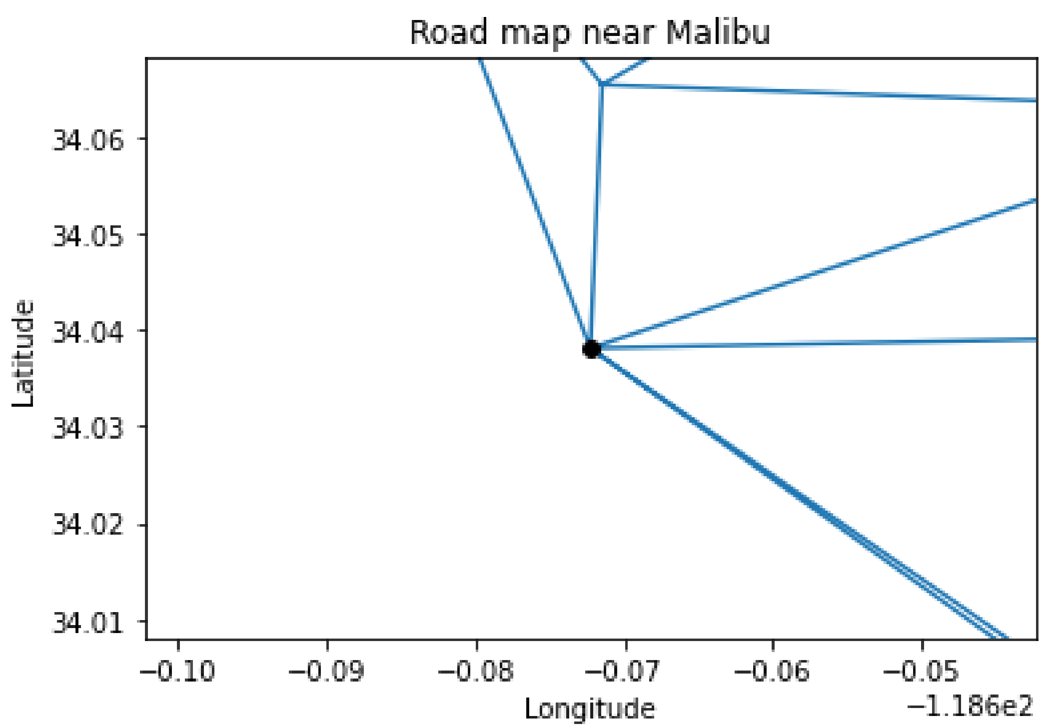
```
Max flow: 7.0
```

Assuming that all roads carry the same weight, max-flow is 7.0 and approximate distance between the source and sink nodes each closest to Malibu and Long Beach is 38.7 miles. If each road consists of two lanes, max-flow = 2x7 = 14.0. Assume average carspeed = 65miles/hour and each car has a gap between one another making the distance between each car: 0.003+2x65/3600 = 0.0391. The total number of cars on the road is 14x38.7/0.0391 = 13857. Then traffic flow is, 13857/(38.7/65) = 23274 cars/hour.

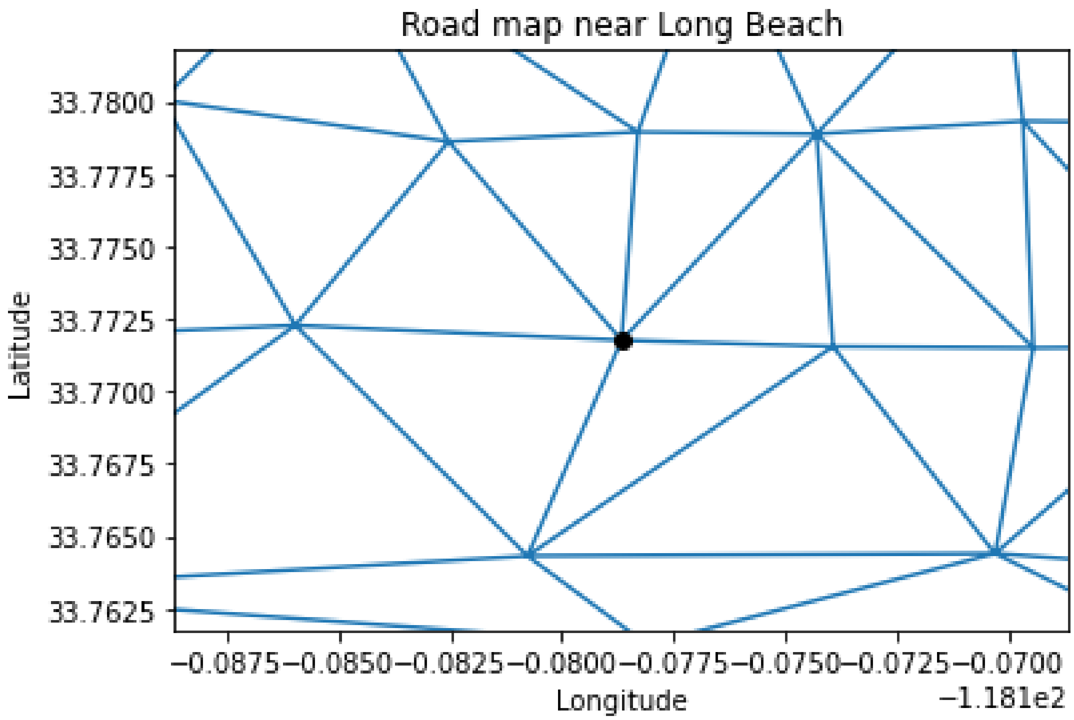
[ ]:
```
print("Number of edge disjoint paths: {}".format(g_del.
  ↪edge_disjoint_paths(malibu_node, long_beach_node) - 1))
print('Degree Distribution of nodes (Malibu, Long Beach): {} {}'.format(g_del.
  ↪degree(malibu_node,mode='out') - 1, g_del.degree(long_beach_node,mode='in')␣
  ↪- 1))
```

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

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



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



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

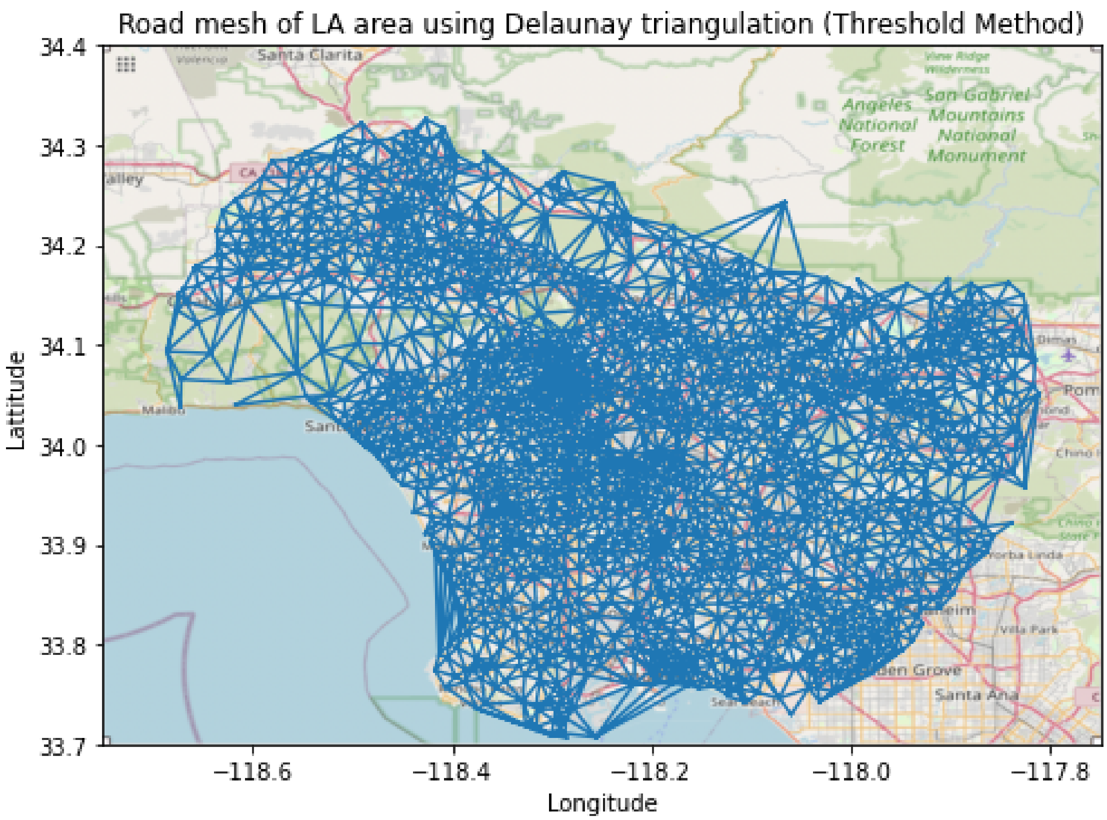
10 Q17

[ ]:
```
#Get rid of longer paths
threshold = 8

g_del_cleaned = ig.Graph()
g_del_cleaned.add_vertices(len(delaunay_out.points))
remove_duplicates = set()
```

```
edge_cut = set()
weights_del_cleaned = []

for i in range(len(delaunay_out.simplices)):
    a = ((delaunay_out.simplices[i][0], delaunay_out.simplices[i][1]))
    b = ((delaunay_out.simplices[i][0], delaunay_out.simplices[i][2]))
    c = ((delaunay_out.simplices[i][1], delaunay_out.simplices[i][2]))

    if not a in remove_duplicates:
        distance = 69*np.sqrt((lat_lon[a[0]][0]-lat_lon[a[1]][0])**2 + ((lat_lon[a[0]][1]-lat_lon[a[1]][1])**2))
        if distance < threshold:
            remove_duplicates.add(a)
            g_del_cleaned.add_edges([a])
            weights_del_cleaned.append(distance)
        else:
            edge_cut.add(a)
    if not b in remove_duplicates:
        distance = 69*np.sqrt((lat_lon[b[0]][0]-lat_lon[b[1]][0])**2 + ((lat_lon[b[0]][1]-lat_lon[b[1]][1])**2))
        if distance < threshold:
            remove_duplicates.add(b)
            g_del_cleaned.add_edges([b])
            weights_del_cleaned.append(distance)
        else:
            edge_cut.add(b)
    if not c in remove_duplicates:
        distance = 69*np.sqrt((lat_lon[c[0]][0]-lat_lon[c[1]][0])**2 + ((lat_lon[c[0]][1]-lat_lon[c[1]][1])**2))
        if distance < threshold:
            remove_duplicates.add(c)
            g_del_cleaned.add_edges([c])
            weights_del_cleaned.append(distance)
        else:
            edge_cut.add(c)

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

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

```
[ ]: #weights are distance in miles
     g_del_cleaned.es['weight'] = weights_del_cleaned
```

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

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



From the plot above, we can observe that the long non-existent routes over the Topanga mountains and the routes over the ocean have disappeared. It indicates that the thresholding method did work.

Furthermore, we can see fewer long edges among neighboring nodes in the plot below, showing that the threshold on speed has indeed removed only those edges which are long but connect neighboring nodes.

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

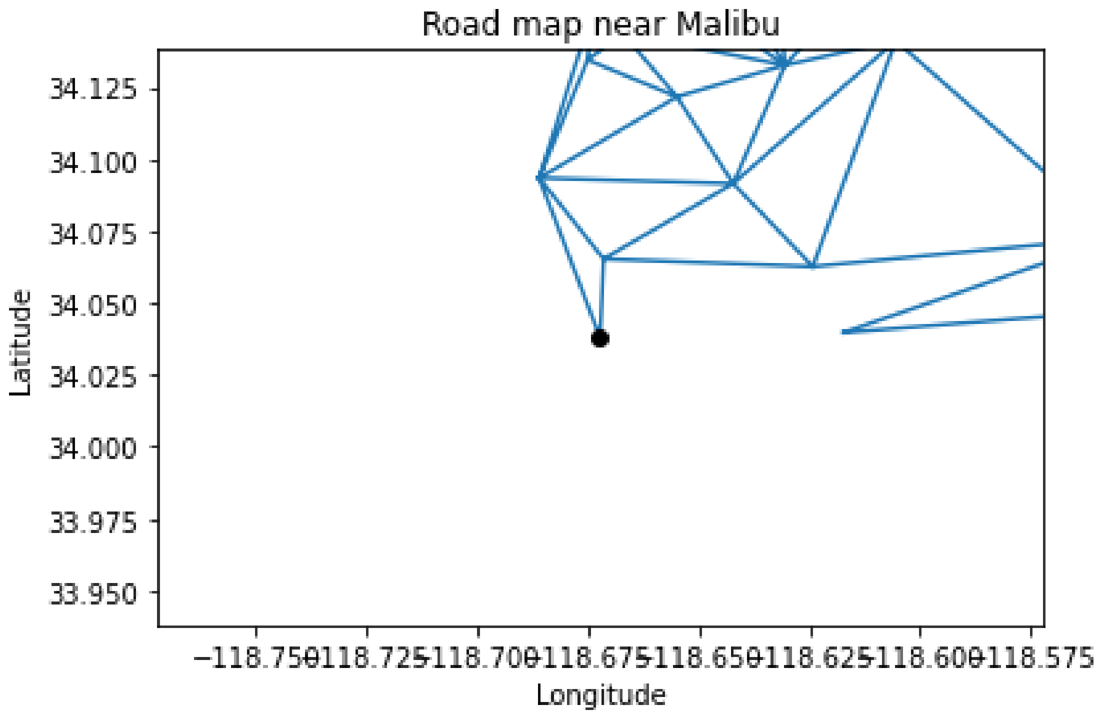
Q17_ig_plot.png

11 Q18

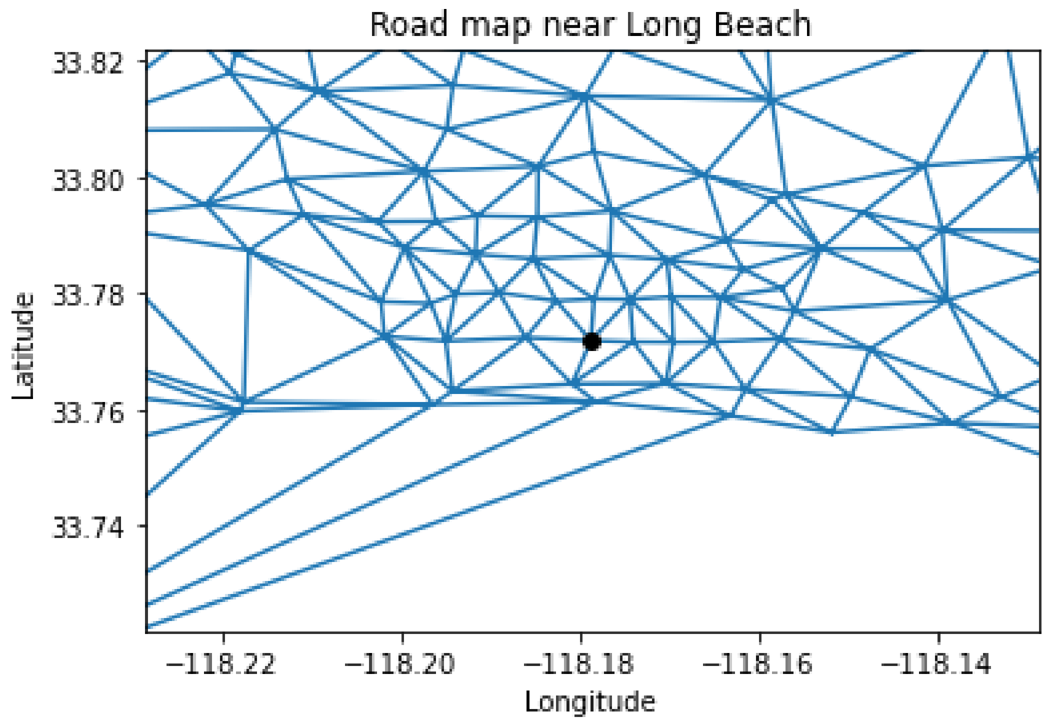
[ ]:
```
print('Number of edge-disjoint paths: {}'.format(g_del_cleaned.
↪adhesion(long_beach_node,malibu_node) - 1))
print('Degree Distribution of nodes (Malibu, Long Beach): {} {}'.
↪format(g_del_cleaned.degree(malibu_node,mode='out') - 1, g_del_cleaned.
↪degree(long_beach_node,mode='in') - 1))
```

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

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




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



After removing the non-existent paths, the number of disjoint paths decreased from 6 to 3.
From the two plots above, we can observe that there has not been any significant change to the road map near long long beach. However, several non-existent paths to Malibu have been pruned out. For two nodes on a graph, the minimum of the number of edges outgoing and incoming to each node provides the number of edge-disjoint paths, which in this case is 3. Since some of the paths to Malibu have been cut, we see a reduction in the number of possible edge-disjoint paths between Malibu and Long Beach.

The maximum flow is still 23274 cars/hour. It is because there still exist many paths that cars can take to reach Long Beach from Malibu. In other words, the capacity of all the roads leading from Malibu to Long Beach is still larger than 23274 cars. It also has to do with how the max-flow algorithm works. The maximum flow is achieved by summing the flow across the minimum cuts (edges with least weights). It is unlikely that the non-existent roads will have the lowest weights, and hence not contribute to the maximum flow at all. Thus, there will be no significant difference in the maximum flow.

12 Q19 - Strategy 1

We solve this problem using Dijkstra's algorithm implemented in NetworkX.

[ ]:
```
edge_ends = np.array([[e.source, e.target] for e in g_del_cleaned.es])
sources, targets = edge_ends[:, 0], edge_ends[:, 1]
sources = sources.tolist()
targets = targets.tolist()
```

```
#distances between nodes
distances = []
count = 0
while count < len(sources):
    distance = 69*np.
 ↪sqrt((lat_lon[sources[count]][0]-lat_lon[targets[count]][0])**2+((lat_lon[sources[count]][1
    distances.append(distance)
    count += 1
```

[ ]:
```
#Create graph
G_1 = nx.Graph()

nodes_list = list(range(len(delaunay_out.points)))
G_1.add_nodes_from(nodes_list)

weighted_edges_list = []
count = 0
while count < len(distances):
    weighted_edges_list.append((sources[count], targets[count],␣
 ↪distances[count]))
    count += 1

G_1.add_weighted_edges_from(weighted_edges_list)
```

[ ]:
```
#Calculate all pairs of shortest paths
length=dict(nx.all_pairs_dijkstra_path_length(G_1))
```

[ ]:
```
#extra_distance_dict_1 returns: source node, sink node, distance in miles
extra_distance_dict = {}
for source in range(int(g_del_cleaned.vcount())):
    print("Iteration: ", source)
    for sink in range(int(g_del_cleaned.vcount())):
        if (source != sink) and ((sink, source) not in extra_distance_dict):
            shortest_distance = length[source][sink]
            geographic_distance = 69*np.
 ↪sqrt((lat_lon[source][0]-lat_lon[sink][0])**2+((lat_lon[source][1]-lat_lon[sink][1])**2))
            extra_distance = shortest_distance - geographic_distance
            extra_distance_dict[(source, sink)] = extra_distance
```

[ ]:
```
highest_extra_distances = {}
extra_distance_dict_removed = extra_distance_dict.copy()
for i in range(20):
    highest_extra_distance_key = max(extra_distance_dict_removed,␣
 ↪key=extra_distance_dict_removed.get)
    highest_extra_distances[highest_extra_distance_key] =␣
 ↪extra_distance_dict_removed[highest_extra_distance_key]
```

```
    del extra_distance_dict_removed[highest_extra_distance_key]

print("Top 20 pairs with highest extra distance ((source, destination): extra␣
 ↪distance):")
highest_extra_distances
```

```
Top 20 pairs with highest extra distance ((source, destination): extra
distance):
```

[ ]:
```
{(43, 2470): 15.308536390955997,
 (56, 2470): 15.688525117834438,
 (383, 2470): 16.181871385580855,
 (384, 2470): 16.429392595905497,
 (385, 2470): 15.883981801049899,
 (386, 2470): 15.883779048335324,
 (387, 2470): 15.32512591590434,
 (388, 2470): 15.37859831013142,
 (389, 2470): 15.883982471259795,
 (390, 2470): 15.719433531847287,
 (391, 2470): 15.400409168337024,
 (392, 2470): 15.391293116863245,
 (393, 2470): 15.708145705549871,
 (394, 2470): 15.63437711058955,
 (1914, 2470): 15.634531058782297,
 (2164, 2470): 16.42154894698211,
 (2166, 2470): 15.368892098333237,
 (2167, 2470): 15.575137925605674,
 (2470, 2472): 15.423680842500227,
 (2470, 2474): 21.326933021964663}
```

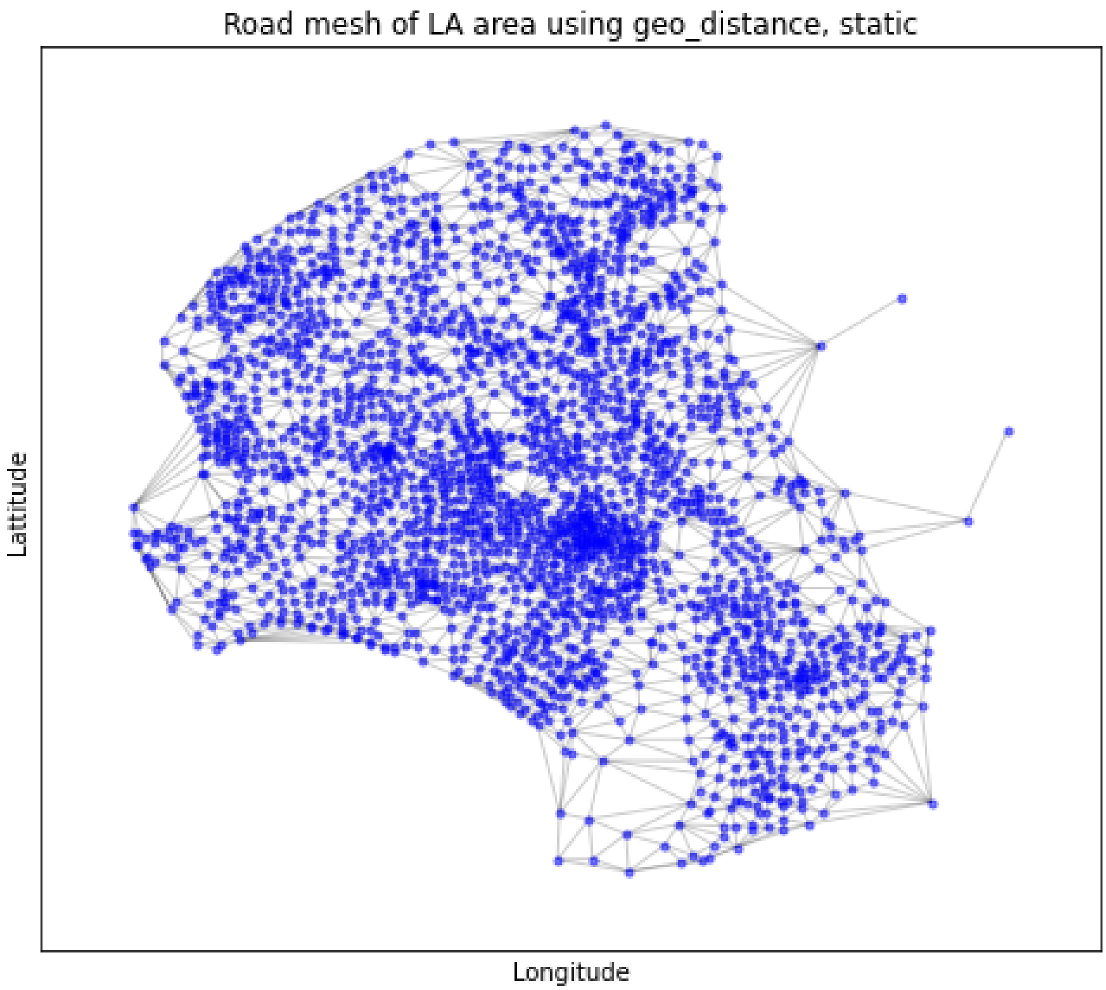
[ ]:
```
print("Node 2470: ", lat_lon[2470])
print("Node 2472: ", lat_lon[2472])
print("Node 2474: ", lat_lon[2474])
```

```
Node 2470:  (34.38948489307656, -118.16620295644888)
Node 2472:  (34.242974950043056, -118.06647650215321)
Node 2474:  (34.30687687852889, -118.00882743874683)
```

This seems to be intuitive as these nodes are located near Angeles National Forest, an isolated region.

[ ]:
```
pos = {}
for node in G_1.nodes:
    pos[node] = (lat_lon[node])
fig, ax = plt.subplots(figsize = (8,7))
nx.draw_networkx_nodes(G_1,pos=pos,node_size=10,node_color='blue',alpha=.5)
nx.draw_networkx_edges(G_1,pos=pos,edge_color='black', alpha=.2)
ax.set_title('Road mesh of LA area using geo_distance, static')
```

```
plt.xlabel('Longitude')
plt.ylabel('Lattitude')
plt.show()
```



Time complexity: O(E+VlogV) where E is the number of edges and V is the number of vertices. The most time consuming part of this strategy is calculating shortest paths for all pairs of nodes. (Run time: 93 seconds)

13 Q20 - Strategy 2

[ ]:
```
G_2 = nx.Graph()
G_2.add_nodes_from(nodes_list)
G_2.add_weighted_edges_from(weighted_edges_list)
```

[ ]:
```
length=dict(nx.all_pairs_dijkstra_path_length(G_2))
```

[ ]:
```
#extra_distance_dict returns: source node, sink node, distance in miles
extra_distance_dict = {}
for source in range(int(g_del_cleaned.vcount())):
    print("Iteration: ", source)
    for sink in range(int(g_del_cleaned.vcount())):
        if (source != sink) and ((sink, source) not in extra_distance_dict):
            shortest_distance = length[source][sink]
            geographic_distance = 69*np.
 ↪sqrt((lat_lon[source][0]-lat_lon[sink][0])**2+((lat_lon[source][1]-lat_lon[sink][1])**2))
            frequency = np.random.randint(1, 1001)
            extra_distance = frequency*(shortest_distance - geographic_distance)
            extra_distance_dict[(source, sink)] = extra_distance
```

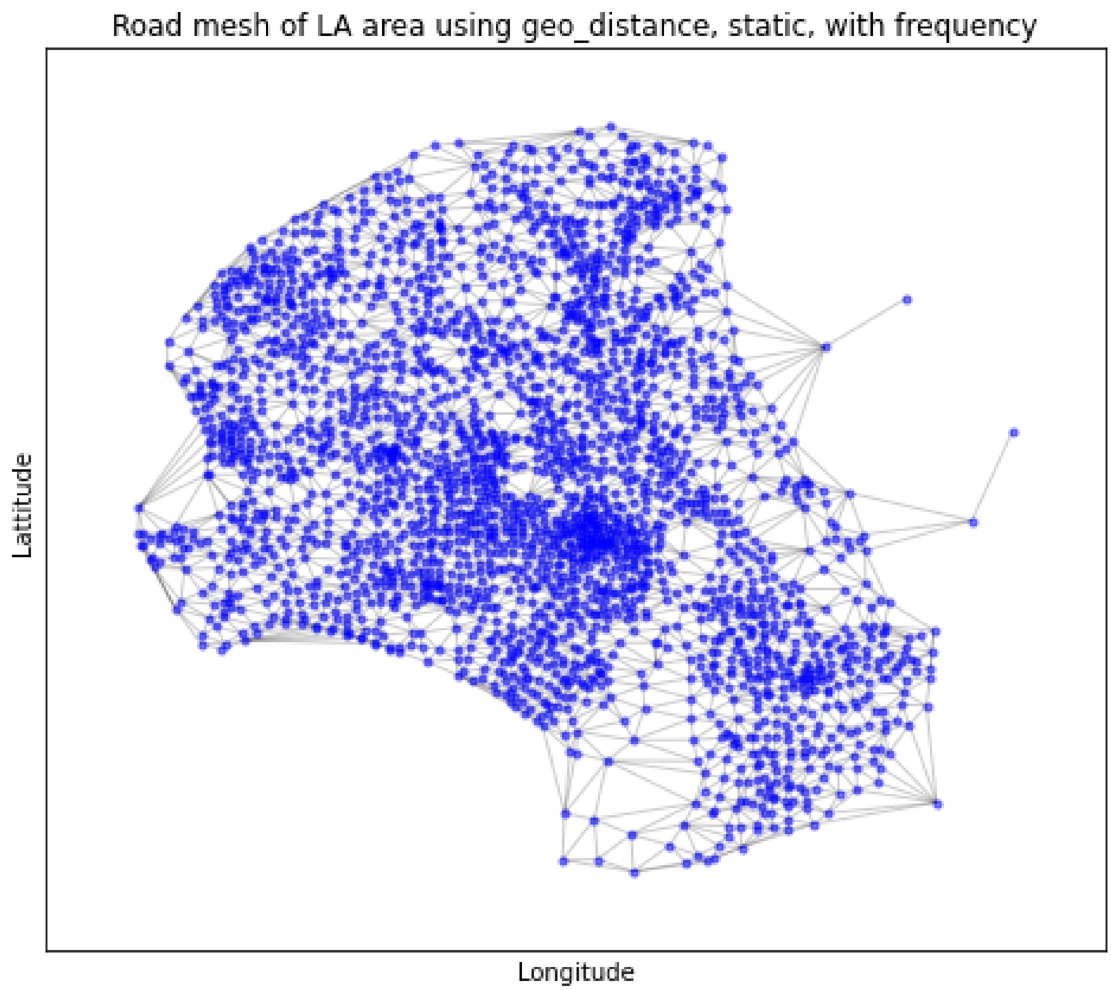
[ ]:
```
highest_extra_distances = {}
extra_distance_dict_removed = extra_distance_dict.copy()
for i in range(20):
    highest_extra_distance_key = max(extra_distance_dict_removed,␣
 ↪key=extra_distance_dict_removed.get)
    highest_extra_distances[highest_extra_distance_key] =␣
 ↪extra_distance_dict_removed[highest_extra_distance_key]
    del extra_distance_dict_removed[highest_extra_distance_key]

print("Top 20 pairs with highest extra distance ((source, destination):␣
 ↪weighted extra distance):")
highest_extra_distances
```

```
Top 20 pairs with highest extra distance ((source, destination): weighted extra
distance):
```

[ ]:
```
{(47, 2470): 12873.299318325342,
 (66, 2470): 12616.416573427505,
 (223, 2470): 13989.554785578124,
 (229, 2470): 13550.207422821632,
 (258, 2470): 12609.972649912488,
 (385, 2470): 14803.871038578505,
 (386, 2470): 14898.984747338534,
 (394, 2470): 14727.583238175357,
 (2063, 2470): 12850.132282530873,
 (2068, 2470): 13202.08098789028,
 (2097, 2470): 13103.122251605686,
 (2101, 2470): 12680.349754699018,
 (2165, 2470): 14129.553253695742,
 (2179, 2470): 13878.976340879562,
 (2198, 2470): 13009.503088410162,
 (2201, 2470): 13553.432445777333,
 (2202, 2470): 12826.876682822098,
 (2203, 2470): 12747.681124659524,
```

```
 (2214, 2470): 12747.718213542652,
 (2470, 2530): 13511.0373567156}
```

[ ]:
```python
pos = {}
for node in G_2.nodes:
    pos[node] = (lat_lon[node])
fig, ax = plt.subplots(figsize = (8,7))
nx.draw_networkx_nodes(G_2,pos=pos,node_size=10,node_color='blue',alpha=.5)
nx.draw_networkx_edges(G_2,pos=pos,edge_color='black', alpha=.2)
ax.set_title('Road mesh of LA area using geo_distance, static, with frequency')
plt.xlabel('Longitude')
plt.ylabel('Lattitude')
plt.show()
```



Time complexity: O(E+VlogV) where E is the number of edges and V is the number of vertices. Same as strategy 1 as the most time consuming part of this strategy is calculating shortest paths for all pairs of nodes. (Run time: 90 seconds)

14 Q21 - Strategy 3

[ ]:
```
G_3 = nx.Graph()
G_3.add_nodes_from(nodes_list)
G_3.add_weighted_edges_from(weighted_edges_list)
```

[ ]:
```
highest_extra_distances = {}

for i in range(20):
    print("Iteration: ", i)
    length=dict(nx.all_pairs_dijkstra_path_length(G_3))
    extra_distance_dict = {}

    #extra_distance_dict_3 returns: source node, sink node, distance in miles
    for source in range(int(g_del_cleaned.vcount())):
        for sink in range(int(g_del_cleaned.vcount())):
            if (source != sink) and ((sink, source) not in extra_distance_dict):
                shortest_distance = length[source][sink]
                geographic_distance = 69*np.
 ↪sqrt((lat_lon[source][0]-lat_lon[sink][0])**2+((lat_lon[source][1]-lat_lon[sink][1])**2))
                extra_distance = (shortest_distance - geographic_distance)
                extra_distance_dict[(source, sink)] = extra_distance

    highest_extra_distance_key = max(extra_distance_dict,␣
 ↪key=extra_distance_dict.get)
    highest_extra_distances[highest_extra_distance_key] =␣
 ↪extra_distance_dict[highest_extra_distance_key]
    add_source, add_sink = highest_extra_distance_key[0],␣
 ↪highest_extra_distance_key[1]
    add_distance = 69*np.
 ↪sqrt((lat_lon[add_source][0]-lat_lon[add_sink][0])**2+((lat_lon[add_source][1]-lat_lon[add_

    #add new edge to graph
    G_3.add_weighted_edges_from([(add_source, add_sink, add_distance)])
```

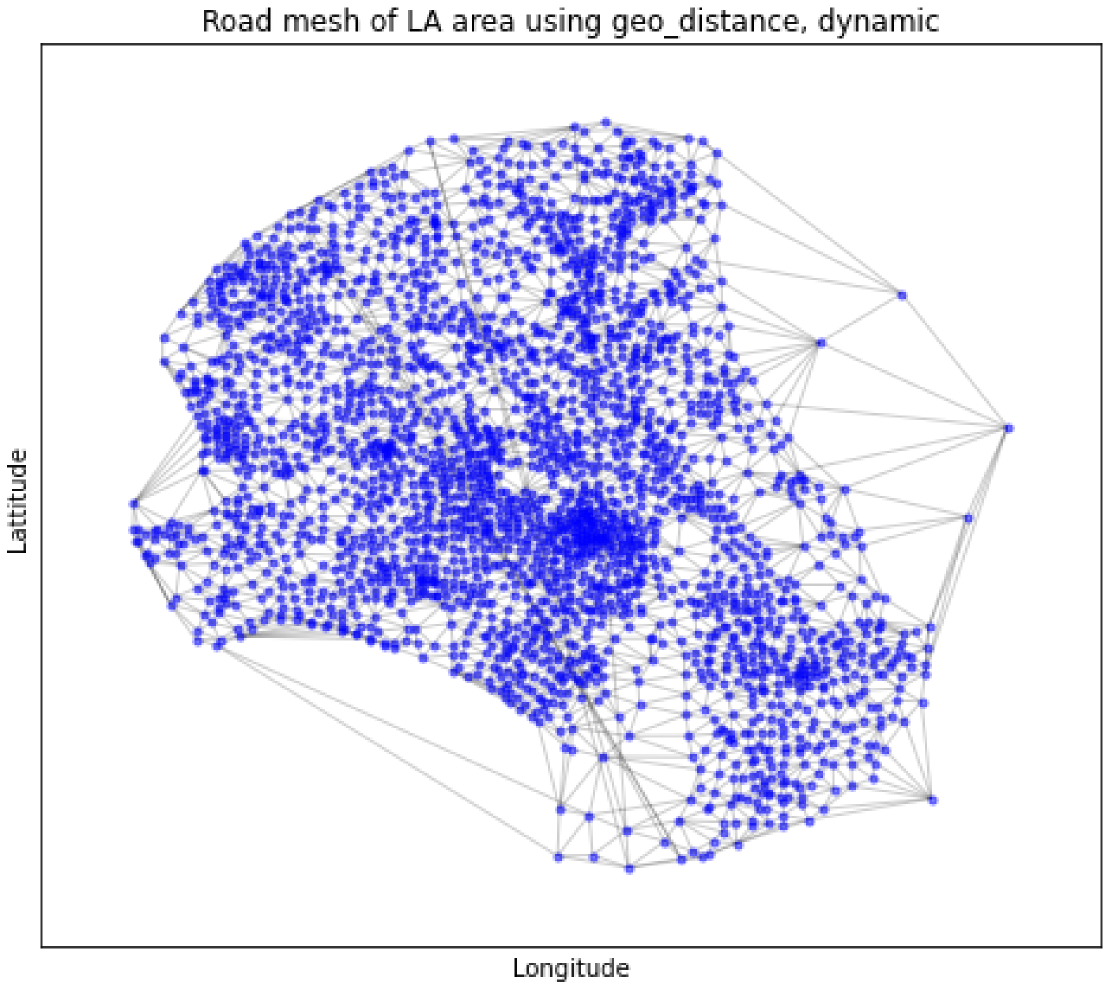
[ ]:
```
print("All new added edges ((source, destination): weighted extra distance):")
highest_extra_distances
```

```
All new added edges ((source, destination): weighted extra distance):
```

[ ]:
```
{(56, 2474): 7.419290133673753,
 (146, 553): 4.078323952505087,
 (147, 2502): 4.21010097080309,
 (147, 2548): 4.192299978899186,
```

```
 (382, 2470): 4.471669235598153,
 (384, 2474): 8.564068545943378,
 (513, 2470): 5.482913396059887,
 (740, 2470): 4.916290845604443,
 (740, 2473): 4.884165313613959,
 (1710, 2448): 5.689950504005186,
 (1866, 2464): 4.039596846378217,
 (2146, 2470): 4.853891510308447,
 (2146, 2473): 4.84496812905946,
 (2252, 2474): 4.06262002150501,
 (2267, 2470): 9.57397724592894,
 (2448, 2469): 4.300824433704594,
 (2464, 2704): 4.168702625660529,
 (2465, 2704): 4.198136861430456,
 (2470, 2472): 5.9734666798800315,
 (2470, 2474): 21.326933021964663}
```

[ ]:
```
pos = {}
for node in G_3.nodes:
    pos[node] = (lat_lon[node])
fig, ax = plt.subplots(figsize = (8,7))
nx.draw_networkx_nodes(G_3,pos=pos,node_size=10,node_color='blue',alpha=.5)
nx.draw_networkx_edges(G_3,pos=pos,edge_color='black', alpha=.2)
ax.set_title('Road mesh of LA area using geo_distance, dynamic')
plt.xlabel('Longitude')
plt.ylabel('Lattitude')
plt.show()
```



Time complexity: O(n(E+VlogV)) where n is the number of for loops (number of roads to add), where E is the number of edges and V is the number of vertices. This is approximately 20 times that of strategy 1 and 2 as the most time consuming part of this strategy is calculating shortest paths for all pairs of nodes but this time it is nested in a for-loop. (Run time: 22 minutes)

15 Q21 - Strategy 4

[ ]:
```
G_4 = nx.MultiDiGraph()
```

[ ]:
```
#Add edges but directed since travel time is not equal in both ways but␣
↪bidirectional
threshold=8
remove_duplicates=set()
source_list_G4 = set()
sink_list_G4 = set()
```

```
for i in range(len(delaunay_out.simplices)):
    a=((delaunay_out.simplices[i][0], delaunay_out.simplices[i][1]))
    b=((delaunay_out.simplices[i][0], delaunay_out.simplices[i][2]))
    c=((delaunay_out.simplices[i][1], delaunay_out.simplices[i][2]))

    if not a in remove_duplicates:
        distance = 69*np.
 ↪sqrt((lat_lon[a[0]][0]-lat_lon[a[1]][0])**2+((lat_lon[a[0]][1]-lat_lon[a[1]][1])**2))
        if distance < threshold:
          remove_duplicates.add(a)
          if (a[0], a[1]) in travel_time_dict:
              G_4.add_edge(a[0], a[1], distance=distance,␣
 ↪time=travel_time_dict[(a[0], a[1])])
              source_list_G4.add(a[0])
              sink_list_G4.add(a[1])
          if (a[1], a[0]) in travel_time_dict:
              G_4.add_edge(a[1], a[0], distance=distance,␣
 ↪time=travel_time_dict[(a[1], a[0])])
              source_list_G4.add(a[1])
              sink_list_G4.add(a[0])
        else:
          pass
    if not b in remove_duplicates:
        distance = 69*np.
 ↪sqrt((lat_lon[b[0]][0]-lat_lon[b[1]][0])**2+((lat_lon[b[0]][1]-lat_lon[b[1]][1])**2))
        if distance < threshold:
          remove_duplicates.add(a)
          if (b[0], b[1]) in travel_time_dict:
              G_4.add_edge(b[0], b[1], distance=distance,␣
 ↪time=travel_time_dict[(b[0], b[1])])
              source_list_G4.add(b[0])
              sink_list_G4.add(b[1])
          if (b[1], b[0]) in travel_time_dict:
              G_4.add_edge(b[1], b[0], distance=distance,␣
 ↪time=travel_time_dict[(b[1], b[0])])
              source_list_G4.add(b[1])
              sink_list_G4.add(b[0])
        else:
          pass
    if not c in remove_duplicates:
        distance = 69*np.
 ↪sqrt((lat_lon[c[0]][0]-lat_lon[c[1]][0])**2+((lat_lon[c[0]][1]-lat_lon[c[1]][1])**2))
        if distance < threshold:
          remove_duplicates.add(a)
          if (c[0], c[1]) in travel_time_dict:
```

```
            G_4.add_edge(c[0], c[1], distance=distance,␣
↪time=travel_time_dict[(c[0], c[1])])
            source_list_G4.add(c[0])
            sink_list_G4.add(c[1])
        if (c[1], c[0]) in travel_time_dict:
            G_4.add_edge(c[1], c[0], distance=distance,␣
↪time=travel_time_dict[(c[1], c[0])])
            source_list_G4.add(c[1])
            sink_list_G4.add(c[0])
      else:
        pass
```

[ ]:
```
length_dist=dict(nx.all_pairs_dijkstra_path_length(G_4, weight='distance'))
length_time=dict(nx.all_pairs_dijkstra_path_length(G_4, weight='time'))
```

[ ]:
```
#extra_time_dict returns: source node, sink node, distance in miles
extra_time_dict = {}
for source in source_list_G4:
    print("Iteration: ", source)
    for sink in sink_list_G4:
        if (source != sink):
            try:
              distance_of_shortest_path = length_dist[source][sink]
              travel_time_of_shortest_path = length_time[source][sink]
              euclidean_distance = 69*np.
↪sqrt((lat_lon[source][0]-lat_lon[sink][0])**2+((lat_lon[source][1]-lat_lon[sink][1])**2))
              travel_speed = distance_of_shortest_path /␣
↪travel_time_of_shortest_path
              extra_time = travel_time_of_shortest_path - (euclidean_distance/
↪travel_speed)
              extra_time_dict[(source, sink)] = extra_time
            except:
              #some subgraphs are cyclic
              pass
```

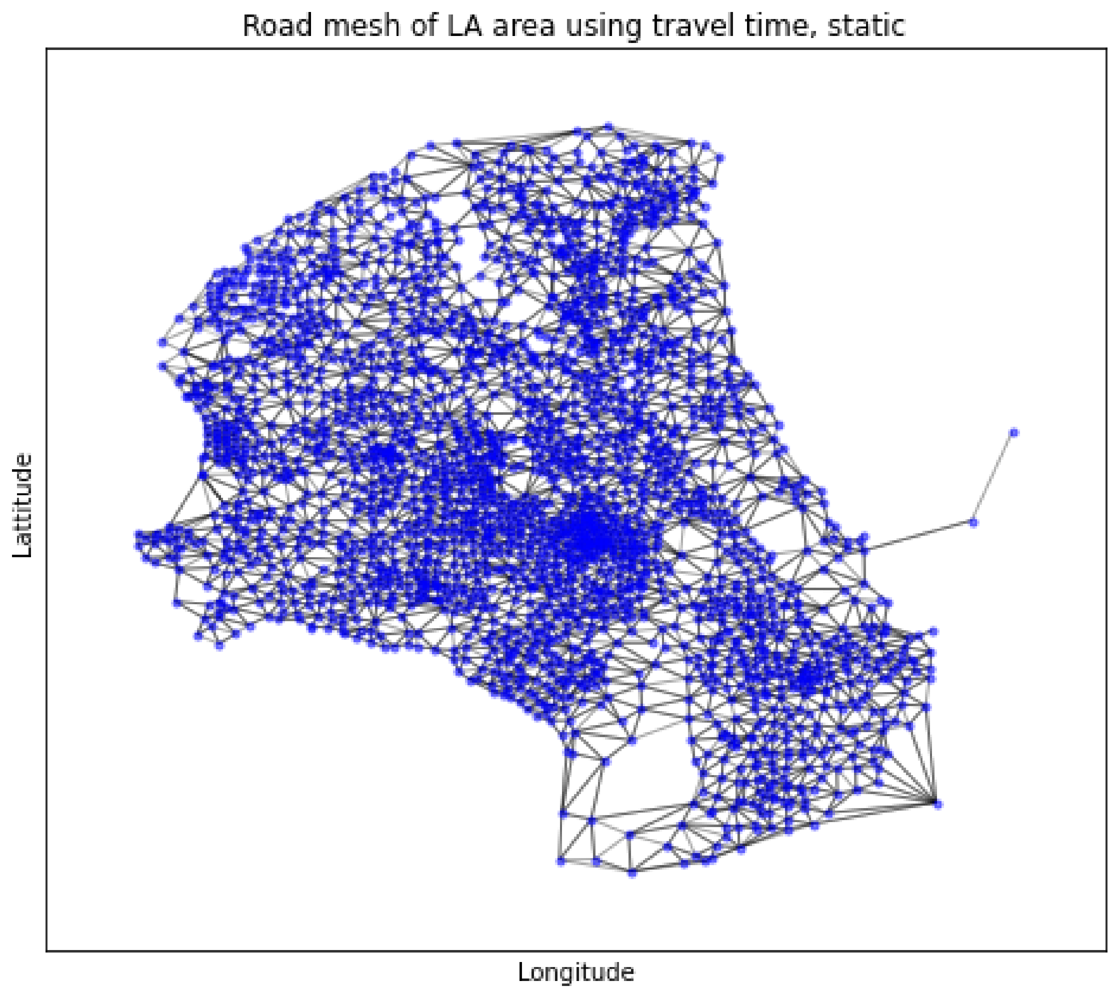
[ ]:
```
highest_extra_times = {}
extra_time_dict_removed = extra_time_dict.copy()
for i in range(20):
    print("Iteration: ", i)
    highest_extra_time_key = max(extra_time_dict_removed,␣
↪key=extra_time_dict_removed.get)
    highest_extra_times[highest_extra_time_key] =␣
↪extra_time_dict_removed[highest_extra_time_key]
    del extra_time_dict_removed[highest_extra_time_key]
```

```
[ ]: print("Top 20 pairs with highest extra times ((source, destination): extra␣
     ↪time):")
     highest_extra_times
```

```
Top 20 pairs with highest extra times ((source, destination): extra time):
```

```
[ ]: {(226, 2470): 2495.422825607556,
      (383, 2470): 2621.509735544352,
      (384, 2470): 2696.651414883527,
      (385, 2470): 2812.4179287423854,
      (388, 2470): 2549.266231336667,
      (389, 2470): 2632.785852072856,
      (390, 2470): 2632.9417594600086,
      (391, 2470): 2495.140188434246,
      (392, 2470): 2573.6975664358865,
      (393, 2470): 2516.430042767532,
      (394, 2470): 2525.7745068944905,
      (1914, 2470): 2544.7168930562243,
      (1917, 2470): 2513.0884306244116,
      (1919, 2470): 2491.831159690905,
      (2067, 2470): 2507.2682000331824,
      (2164, 2470): 2820.0685047211823,
      (2166, 2470): 2516.722066120142,
      (2167, 2470): 2559.8447241004396,
      (2169, 2470): 2627.598881801304,
      (2176, 2470): 2519.162999258432}
```

```
[ ]: pos = {}
     for node in G_4.nodes:
         pos[node] = (lat_lon[node])
     fig, ax = plt.subplots(figsize = (8,7))
     nx.draw_networkx_nodes(G_4,pos=pos,node_size=10,node_color='blue',alpha=.5)
     nx.draw_networkx_edges(G_4,pos=pos,edge_color='black', alpha=.2, arrows=False)
     ax.set_title('Road mesh of LA area using travel time, static')
     plt.xlabel('Longitude')
     plt.ylabel('Lattitude')
     plt.show()
```



Time complexity: O(E+VlogV) where E is the number of edges and V is the number of vertices. Same as strategy 1 and 2 as the most time consuming part of this strategy is calculating shortest paths for all pairs of nodes. However, the run time is twice as the types of weights to add is increased to 2 for path time and path distance. If we had n types of weights, the time complexity would be O(n(E+VlogV)). (Run time: 4 minutes)

16 Q22 - Strategy 5

[ ]:
```
G_5 = G_4.copy()

highest_extra_times = {}

for i in range(20):
    print("Iteration: ", i)
    length_dist=dict(nx.all_pairs_dijkstra_path_length(G_5, weight='distance'))
    length_time=dict(nx.all_pairs_dijkstra_path_length(G_5, weight='time'))
```

```
    extra_time_dict = {}
    travel_time_of_new_edge_dict = {}

    #extra_time_dict returns: source node, sink node, time
    for source in source_list_G4:
        for sink in sink_list_G4:
            if (source != sink):
                try:
                  distance_of_shortest_path = length_dist[source][sink]
                  travel_time_of_shortest_path = length_time[source][sink]
                  euclidean_distance = 69*np.
 ↪sqrt((lat_lon[source][0]-lat_lon[sink][0])**2+((lat_lon[source][1]-lat_lon[sink][1])**2))
                  travel_speed = distance_of_shortest_path /␣
 ↪travel_time_of_shortest_path
                  travel_time_of_new_edge = euclidean_distance/travel_speed
                  extra_time = travel_time_of_shortest_path -␣
 ↪travel_time_of_new_edge
                  extra_time_dict[(source, sink)] = extra_time
                  travel_time_of_new_edge_dict[(source, sink)] =␣
 ↪travel_time_of_new_edge
                except:
                  #some subgraphs are cyclic
                  pass

    highest_extra_time_key = max(extra_time_dict, key=extra_time_dict.get)
    highest_extra_times[highest_extra_time_key] =␣
 ↪extra_time_dict[highest_extra_time_key]
    add_source, add_sink = highest_extra_time_key[0], highest_extra_time_key[1]
    add_time = travel_time_of_new_edge_dict[(add_source, add_sink)]

    #add new edge to graph
    G_5.add_weighted_edges_from([(add_source, add_sink, add_time)])
```

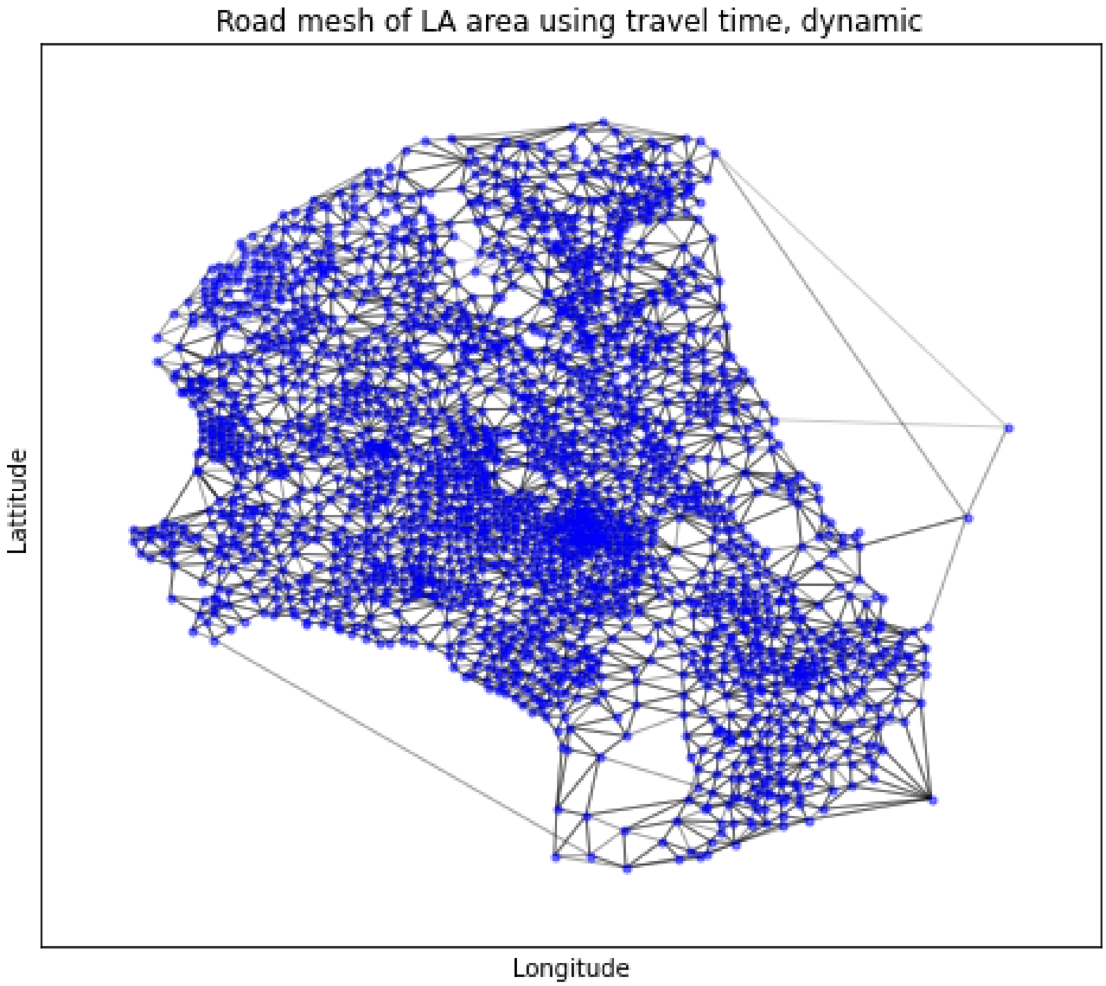
[ ]:
```
print("Top 20 pairs with highest extra times ((source, destination): extra␣
 ↪time):")
highest_extra_times
```

```
Top 20 pairs with highest extra times ((source, destination): extra time):
```

[ ]:
```
{(1000, 1521): 1028.770883896505,
 (1521, 1000): 959.2156746981334,
 (1794, 2468): 1244.693411907312,
 (2146, 2473): 738.5330768171541,
 (2164, 2470): 2820.0685047211823,
 (2164, 2473): 1207.6087613306481,
 (2238, 2142): 846.0612436132911,
 (2241, 2292): 1023.5567496825952,
```

```
 (2268, 2470): 1545.2919980122285,
 (2468, 1794): 1294.6235642245156,
 (2473, 2146): 787.4301318567634,
 (2473, 2164): 1199.6968173499772,
 (2604, 2113): 787.4084086082498,
 (2610, 2614): 1447.0355256037797,
 (2613, 2609): 1088.97454766907,
 (2637, 2242): 893.3467753223767,
 (2678, 2108): 902.9706207848049,
 (2678, 2685): 765.3422483507171,
 (2683, 2113): 918.5613740995213,
 (2710, 2678): 717.8503731712833}
```

[ ]:
```
pos = {}
for node in G_5.nodes:
    pos[node] = (lat_lon[node])
fig, ax = plt.subplots(figsize = (8,7))
nx.draw_networkx_nodes(G_5,pos=pos,node_size=10,node_color='blue',alpha=.5)
nx.draw_networkx_edges(G_5,pos=pos,edge_color='black', alpha=.2, arrows=False)
ax.set_title('Road mesh of LA area using travel time, dynamic')
plt.xlabel('Longitude')
plt.ylabel('Lattitude')
plt.show()
```



Time complexity: O(n(E+VlogV)) where n is the number of roads/for-loop iterations, E is the number of edges and V is the number of vertices. Same as strategy 3 as the most time consuming part of this strategy is calculating shortest paths for all pairs of nodes and that is nested in a for-loop. If we had n types of weights, the time complexity would be O(n^2(E+VlogV)). (Run time: 1 hour)

17 Q23

a) Which one is better? Strategy 1 or 2 Both strategies have the same time complexity except strategy 2 factors popularities of each road into account. Since the weighted distance better reflects its travelling times due to road congestion, strategy 2 is better. We also observe that by multiplying frequency, our maximum travelling distance is increased on a linear scale (i.e n*extra_distance where n=range[1,1000]). Therefore, the weights for those with higher extra travelling distance and higher frequency will be increased by N^2 and if we had real data on road frequencies, results for highest travelling distance will be further more accurate.

b) Which one is better? Strategy 1 or 3 Strategy 3 is better because it updates distances between each road pair and identifies the following maximum road distances in each iteration. Since strategy 1 does not update accordingly, we observe that the pairs with highest maximum distances include node 2470. However after updating, we observe node 2470 in fewer occasions because it may have needed only a few major roads that connect to node 2470. Had we chosen to adopt strategy 1, we would've created multiple roads connecting to 2470, which wouldn't have been as efficient as just creating the minimum number of roads needed to minimize distances of all paths that lead to 2470 as the sink and creating other edges that have higher priorities of construction. The only issue of strategy 3 is the increased time complexity. However, as it only took 20 minutes to compute with 12GB RAM, it would be more worthwhile to adopt strategy 3 when investing in new roads.

c) Which one is better? Strategy 1 or 4 Strategy 4 is better because while strategy 1 is an undirected graph (distances are same regardless of direction,) strategy 2 is a bidirectional graph which takes into account travelling times, which are not the same for both paths between a pair of nodes. Furthermore, travelling speeds were assumed the same for all edges in strategy 1 as we only incorporated distances as weights, but in strategy 4, we account for travelling speeds between each path, which allows us to better depict efficiencies in building roads. The time complexity of strategy 4 would be the time complexity of strategy 3 multiplied by the number of weight types but in this case we have only considered distance and time, which only leads to twice the computation time.

d) Which one is better? Static or dynamic Dynamic updates the graph with new roads while static does not. Therefore, dynamic is better because it allows us to exclude duplicated roads and only build roads where they have the highest priority. The time complexity is increased by a multiple of n where n is the number of roads or the number of for-loop iterations. Again, while it does take more intensive computation, we observe huge differences in the pairs of newly added edges between strategy 1 & 3, and strategy 4 & 5, making the dynamic method worthwhile when investing for a costly process such as building roads.

e) Come up with new strategy I think that one of the easiest strategies to implement would be to get more specific data: consider the road frequencies/travelling speeds during peak times in order to resolve traffic congestions (i.e instead of getting monthly data averaged over the entire duration, get weighted distances/times averaged monthly during the peak times only.) Furthermore, get the number of lanes for each edges in order to calculate more accurate weights. Another would be to get more detailed data and increase the number of bins in our travelling times data so that we can create more nodes and have further precise travelling speeds, as well as more accurate geodistances as increasing more nodes would better road approximation since we consider all paths to be in a straight line (which is physically not the case.)

Asides from getting more specific data, we could try to modify strategy 4. Instead of updating our graph after each iteration and re-calculating maximum distances, we may: 1. for each iteration calculate several pairs with the highest travelling times(we only calculate 1 pair in strategy 5 but change that to say 20 or 30), 2. store the highest pair in that list but remove all the other pairs that have either the same node as source or sink, 3. repeat this iteration 20 times. This way, we could avoid the huge time complexity, which comes with strategy 5 because we no longer have to recalculate for all the distant paths which takes the same node in common.

Own_Task

June 10, 2022

1 Our own task

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

We can formulate the problem as a linear programming problem. Consider a graph $G(V, E)$, where V represents the location of customers and E represents the routes. The edges of the graph are weighted, with c_{ij} representing the travel distance between node i and j. Node 0 refers to the warehouse where the packages are stored, and the other nodes represent the customer locations. K represents the vehicles. d_i is the number of packages required by customer i and $Q \geq 0$ is the max capacity. Define x_{ij}^k that indicates whether a vehicle $k \in K$ will take route e_{ij}, $x_{ij}^k = 1$ if route taken, otherwise equals to 0.

The following are objective functions and constraints for this programming problem:

objective function: $\min \sum_{k\in K} \sum_{(i,j)\in E} c_{ij}x_{ij}^k$

constraints:

1.only one vehicle can visit each customer: $\sum_{k\in K} \sum_{i\in V, i\notin j} x_{ij}^k = 1, \forall j \in V\backslash 0$

2.each vehicle start from the warehouse: $\sum_{j\in V\backslash 0} x_{0j}^k = 1$

3.number of vehicles entering and exiting a node is equal: $\sum_{i\in V, i\notin j} x_{ij}^k - \sum_{i\in V} x_{ji}^k = 0$

4.each vehicle carry at most Q: $\sum_{i\in V} \sum_{j\in V\backslash 0, i\neq j} q_j x_{ij}^k \leq Q, \forall k \in K$

5.vehicle starts and returns to warehouse: $\sum_{k\in K} \sum_{(i,j)\in S, i\neq j} x_{ij}^k \leq |S| - 1, S \subseteq V\backslash 0$

6.$x_{ij}^k \in \{0, 1\}$

[ ]:
```
!pip install pulp
!pip install gmaps
!pip install googlemaps
```

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

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

```
[ ]: vehicle_count = 4
     vehicle_capacity = 50
     customer_count = 10
     np.random.seed(seed=150)
     depot_latitude = 40.748817
     depot_longitude = -73.985428

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

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

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

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

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

```
9  40.755834 -73.983573            11
```

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

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

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

    return _fig

def _distance_calculator(_df):

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

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

    return _distance_result
```

[ ]:
```
#Locations of customers relative to the warehouse on Google Maps
distance = _distance_calculator(df)
plot_result = _plot_on_gmaps(df)
plot_result
```

figure click here

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

    problem = pulp.LpProblem("CVRP", pulp.LpMinimize)
```

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

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

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

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

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

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

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

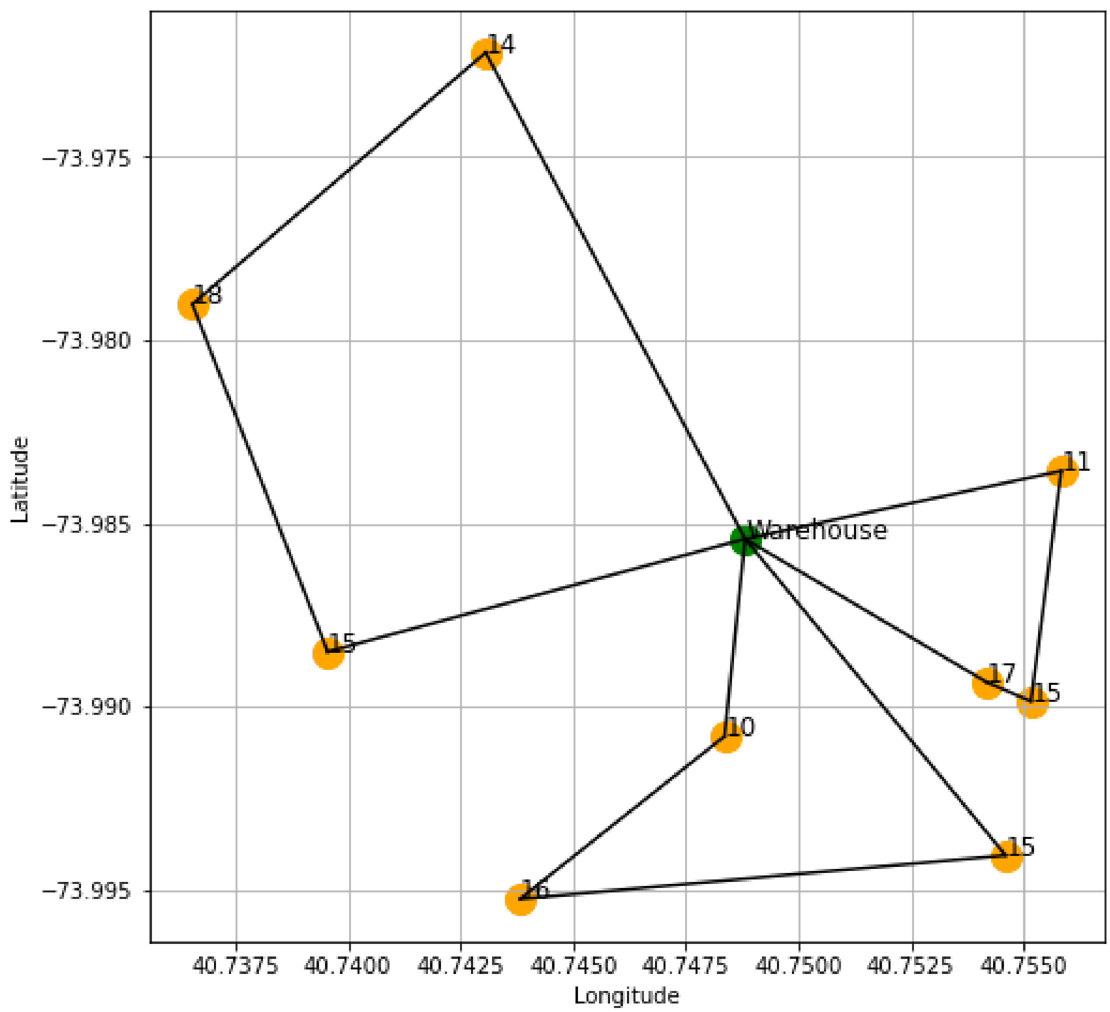
[ ]:
```
#Most optimal route for the CVRP, using 3 out of 4 vehicles
fig = gmaps.figure()
layer = []
color_list = ["red","blue","green"]

for k in range(vehicle_count):
```

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

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

fig
```

figure click here

[ ]:

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

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

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



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