QUESTION 1:

What are upper and lower bounds on ρij? Provide a justification for using log-normalized return (ri(t)) instead of regular return (qi(t)).

Answer1:

The correlation coefficient ρ_{ij} measures the correlation between the log-normalized returns of stocks i and j. It has upper and lower bounds:

1. **Independent Returns:**
   When $r_i(t)$ and $r_j(t)$ are independent, their product's expected value equals the product of their individual expected values. Thus, $\langle r_i(t) r_j(t) \rangle = \langle r_i(t) \rangle \langle r_j(t) \rangle$, making $\rho_{ij} = 0$.

2. **Perfectly Correlated Returns:**
   If $r_i(t) = k \cdot r_j(t)$, where k is a constant, then $\langle r_i(t) r_j(t) \rangle = k \langle r_i(t)^2 \rangle$. The correlation ρ_{ij} becomes $\frac{k}{|k|}$, which is 1 for $k > 0$ (perfect positive correlation) and -1 for $k < 0$ (perfect negative correlation).

Therefore, the bounds for ρ_{ij} are from -1 to 1.

Why Use Log-Normalized Return $r_i(t)$ Instead of Regular Return $q_i(t)$?

Log-normalized returns are preferred because:

1. **Normal Distribution:** Log returns are closer to being normally distributed, which simplifies statistical analysis.
2. **Additivity:** Log returns over multiple periods can be summed, making it easier to analyze returns over time.

Regular returns lack these properties, making log-normalized returns more suitable for financial analysis.

QUESTION2:

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

In [ ]:
```
install.packages("igraph")
install.packages("clevr")
```

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

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

also installing the dependencies ‘Rcpp’, ‘BH’
```

In [ ]:
```
library(igraph)
library(clevr)
```

In [ ]:
```
install.packages("googledrive")
library(googledrive)

drive_auth()

drive_files <- drive_find()
print(drive_files)
```

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

The googledrive package is requesting access to your Google account.
Enter '1' to start a new auth process or select a pre-authorized account.
1: Send me to the browser for a new auth process.
2: hyj1118@g.ucla.edu
Selection: 2
```

```
Auto-refreshing stale OAuth token.
```

```
# A dribble: 5,895 × 3
   name                  id                                drive_resource
   <chr>                 <drv_id>                          <list>
 1 ece232e-project4-part1 1AkmXRknBj_S454-ei36BmFkDyTnrbYdf <named list [43]>
 2 data                  1SF8IxpeP83mQZJwEk1pUAMiXc4o3zPZK <named list [34]>
 3 UAA.csv               1AsjaB8FSFxVRU44SXhthDAinYtP06xMi <named list [43]>
 4 BLK.csv               1vXgfsxjstdeaL-W8VQJhORTwVpZQJBg3 <named list [43]>
 5 PPL.csv               1XLcG9SxYIDmdrTmpnuPemun3u8r6i_u- <named list [42]>
 6 BWA.csv               1u0Gglo5aEen91gEy-ZicD7zpBzrwRlNO <named list [42]>
 7 NWL.csv               1Fr6L6HaJTMdiajoL7tB2NDGk720G4ZMx <named list [42]>
 8 LRCX.csv              1pWs4_aZIUc1oEVIGnvM_VNBZhgK-1Bdg <named list [42]>
 9 WAT.csv               1O0BVYhKE5XNsspnP-BQ-q-zOq_ElzvFV <named list [42]>
10 FTI.csv               1Mzc9uN1yhAeU1cRuO0wkGCmzLbAJsvDN <named list [42]>
# ℹ 5,885 more rows
```

In [ ]:
```
file_name <- "Name_sector.csv"
```

In [ ]:
```
file_to_download <- drive_find(pattern = file_name)
```

In [ ]:
```
drive_download(file_to_download, path = file_name, overwrite = TRUE)
```

```
File downloaded:

• Name_sector.csv <id: 1_VZ5PbqjOVkOALdJJor6d69WRS_8IIcx>

Saved locally as:

• Name_sector.csv
```

In [ ]:
```
data <- read.csv(file_name)
print(head(data))
```

```
  Symbol                 Sector
1      A            Health Care
2    AAL            Industrials
3    AAP Consumer Discretionary
4   AAPL Information Technology
5   ABBV            Health Care
6    ABC            Health Care
```

In [ ]:
```
all_folders <- drive_ls(type = "folder")
print(all_folders)
```

```
# A dribble: 917 × 3
   name                 id                                drive_resource
   <chr>                <drv_id>                          <list>
 1 data                 1SF8IxpeP83mQZJwEk1pUAMiXc4o3zPZK <named list [34]>
 2 finance_data         1Xmvq6TDZjANWKEQ0rOmKfS8_4jdVurQV <named list [34]>
 3 Colab Notebooks      1GmQI2h_Uaz9QX2-yITk48Phn_V-X7Pi0 <named list [34]>
 4 Capstone             1swKYaawE7qqmq6A73VpMOV8JVFw0_5Y4 <named list [34]>
 5 219-project3         1W7Tg9nDea5DMYrwHtuNJWjsPax0u8-A- <named list [34]>
 6 Synthetic_Movie_Lens 1_JF9plSjE3PAFBuSvUFRkDdftJWo1TFz <named list [31]>
 7 share                1--hMY9NIjoUcjWfIztZPJgNxEcRPnkR4 <named list [33]>
 8 My iMac              1ctd7uKW0-im2fwPt_rMrluM3D_79D-8R <named list [32]>
 9 Pokemon              1MToeLgMs8KUs0KfK-D59ttMjCL1iKVrb <named list [33]>
10 data-9               1hb5nLxoKeDPfN7uPqK9yitDrRATyidlj <named list [31]>
# ℹ 907 more rows
```

In [ ]:
```
directory_name <- "finance_data"
```

In [ ]:
```
directory <- drive_find(pattern = directory_name, type = "folder")

if (nrow(directory) == 0) {
  stop("Directory not found.")
} else {
  directory_id <- directory$id[1]
}
```

In [ ]:
```
files_in_directory <- drive_ls(as_id(directory_id))
print(files_in_directory)
```

```
# A dribble: 2 × 3
  name            id                                drive_resource
  <chr>           <drv_id>                          <list>
1 .DS_Store       1A2z3yxxFKt6fkefxesVoA9JGvjxddlou <named list [42]>
2 Name_sector.csv 1_VZ5PbqjOVkOALdJJor6d69WRS_8IIcx <named list [42]>
```

In [ ]:
```
directory_name1 <- "data"
```

In [ ]:
```
directory <- drive_find(pattern = directory_name1, type = "folder")

if (nrow(directory) == 0) {
  stop("Directory not found.")
} else {
  directory_id <- directory$id[1]
}
```

In [ ]:
```
files_in_directory <- drive_ls(as_id(directory_id))
print(files_in_directory)
```

```
# A dribble: 506 × 3
   name     id                                drive_resource
   <chr>    <drv_id>                          <list>
 1 UAA.csv  1AsjaB8FSFxVRU44SXhthDAinYtP06xMi <named list [43]>
 2 BLK.csv  1vXgfsxjstdeaL-W8VQJhORTwVpZQJBg3 <named list [43]>
 3 NWL.csv  1Fr6L6HaJTMdiajoL7tB2NDGk720G4ZMx <named list [42]>
 4 PPL.csv  1XLcG9SxYIDmdrTmpnuPemun3u8r6i_u- <named list [42]>
 5 BWA.csv  1u0Gglo5aEen91gEy-ZicD7zpBzrwRlNO <named list [42]>
 6 LRCX.csv 1pWs4_aZIUc1oEVIGnvM_VNBZhgK-1Bdg <named list [42]>
 7 WAT.csv  1O0BVYhKE5XNsspnP-BQ-q-zOq_ElzvFV <named list [42]>
 8 FTI.csv  1Mzc9uN1yhAeU1cRuO0wkGCmzLbAJsvDN <named list [42]>
 9 CRM.csv  1Q7LFQvnlCsO5mssNGRyP5Ha2lunkcKKV <named list [42]>
10 TSS.csv  1tmpo_P25tBAU3PuiVAgYp_yFyURIEcMm <named list [42]>
# ℹ 496 more rows
```

In [ ]:
```
# Create a local directory to store the downloaded files
local_dir <- "local_data"
dir.create(local_dir, showWarnings = FALSE)

# Download the CSV files to the local environment
filenames <- vector("character", nrow(files_in_directory))
for (i in seq_len(nrow(files_in_directory))) {
  local_path <- file.path(local_dir, files_in_directory$name[i])
  drive_download(as_id(files_in_directory$id[i]), path = local_path, overwrite = TRUE)
  filenames[i] <- local_path
}
```

In [ ]:
```
if (!requireNamespace("googledrive", quietly = TRUE)) install.packages("googledrive")
if (!requireNamespace("tools", quietly = TRUE)) install.packages("tools")
if (!requireNamespace("readr", quietly = TRUE)) install.packages("readr")

library(googledrive)
library(readr)
drive_auth()
directory_name1 <- "data"

# Find the 'data' directory
directory <- drive_find(pattern = directory_name1, type = "folder")
if (nrow(directory) == 0) {
  stop("Directory not found.")
} else {
  directory_id <- directory$id[1]
}

# List files in the 'data' directory
files_in_directory <- drive_ls(as_id(directory_id))

# Print for debugging
print(sprintf("Contents of the directory '%s':", directory_name1))
print(files_in_directory)

# Remove companies with incomplete data
```

```
comp_del <- 0
for (j in seq_along(filenames)) {
  if (file.exists(filenames[j])) {
    comp_data <- tryCatch(
      {
        readr::read_csv(filenames[j])
      },
      error = function(e) {
        warning(sprintf("Error reading file %s: %s", filenames[j], e))
        NULL
      }
    )
    if (!is.null(comp_data) && nrow(comp_data) != 765) {
      comp_del <- comp_del + 1
    }
  } else {
    warning(sprintf("File not found: %s", filenames[j]))
  }
}
print(sprintf("Number of companies in the ticker list with missing data: %s", comp_del))
```

In [ ]:

```
# Read company ticker list
company_ticker_list <- read.csv(
    "Name_sector.csv", header = TRUE, stringsAsFactors = FALSE
)
head(company_ticker_list, n = 15L)
print(sprintf("Number of companies in the ticker list is %s", nrow(company_ticker_list)))
```

A data.frame: 15 × 2

	Symbol	Sector
	<chr>	<chr>
1	A	Health Care
2	AAL	Industrials
3	AAP	Consumer Discretionary
4	AAPL	Information Technology
5	ABBV	Health Care
6	ABC	Health Care
7	ABT	Health Care
8	ACN	Information Technology
9	ADBE	Information Technology
10	ADI	Information Technology
11	ADM	Consumer Staples
12	ADP	Information Technology
13	ADS	Information Technology
14	ADSK	Information Technology
15	AEE	Utilities

```
[1] "Number of companies in the ticker list is 505"
```

In [ ]:

```r
# Create log-normalized return matrix
log_norm_return <- matrix(0, nrow = length(filenames) - comp_del, ncol = 764)
num_data_points <- c()
i <- 1
for (j in 1:length(filenames)) {
  comp_data <- read.csv(filenames[j], header = TRUE, stringsAsFactors = FALSE)
  num_data_points[j] <- dim(comp_data)[1]
  if (num_data_points[j] == 765) {
    p <- comp_data[, "Close"]
    q <- c()
    q[1] <- 0 # Assumption: gain from 0th day to 1st day is 0.
    for (t in 2:length(p)) {
      q[t - 1] <- (p[t] - p[t - 1]) / p[t - 1]
    }
    r <- log(1 + q)
    log_norm_return[i, ] <- r
    i <- i + 1
  }
}
```

```
Warning message in read.table(file = file, header = header, sep = sep, quote = quote, :
“line 1 appears to contain embedded nulls”
Warning message in read.table(file = file, header = header, sep = sep, quote = quote, :
“line 2 appears to contain embedded nulls”
Warning message in read.table(file = file, header = header, sep = sep, quote = quote, :
“line 3 appears to contain embedded nulls”
Warning message in read.table(file = file, header = header, sep = sep, quote = quote, :
“line 4 appears to contain embedded nulls”
Warning message in read.table(file = file, header = header, sep = sep, quote = quote, :
“line 5 appears to contain embedded nulls”
Warning message in scan(file = file, what = what, sep = sep, quote = quote, dec = dec, :
“embedded nul(s) found in input”
Error in read.table(file = file, header = header, sep = sep, quote = quote, : duplicate 'row.names' are not allowed
Traceback:

1. read.csv(filenames[j], header = TRUE, stringsAsFactors = FALSE)
2. read.table(file = file, header = header, sep = sep, quote = quote,
 .     dec = dec, fill = fill, comment.char = comment.char, ...)
3. stop("duplicate 'row.names' are not allowed")
```

In [ ]:

```r
# Compute edge weights for the correlation graph
generate_correlation_weights <- function(wt_file, log_norm_return, company_ticker_list) {
  cat("from_node", "\t", "to_node", "\t", "edge_wt", file = wt_file)
  num_companies <- nrow(log_norm_return)
  for (i in 1:(num_companies - 1)) {
    for (j in (i + 1):num_companies) {
      r_i <- mean(log_norm_return[i, ])
      r_j <- mean(log_norm_return[j, ])
      r_ij <- mean(log_norm_return[i, ] * log_norm_return[j, ])
      r_i_squared <- log_norm_return[i, ]^2
      r_j_squared <- log_norm_return[j, ]^2
      p_ij <- (r_ij - (r_i * r_j)) / (sqrt((mean(r_i_squared) - (r_i^2)) * (mean(r_j_squared) - (r_j^2))))
      w_ij <- sqrt(2 * (1 - p_ij))
      cat('\n', company_ticker_list[i, 1], '\t', company_ticker_list[j, 1], '\t', w_ij, file = wt_file)
    }
  }
}

# Remove companies with incomplete data from the ticker list
comp_ignore <- which(num_data_points != 765)
company_ticker_list <- company_ticker_list[-comp_ignore, ]

# Write the weights to a file to construct the graph later
wt_file <- file("edge_weights.txt", "w")
generate_correlation_weights(wt_file, log_norm_return, company_ticker_list)
close(wt_file)
```

In [ ]:

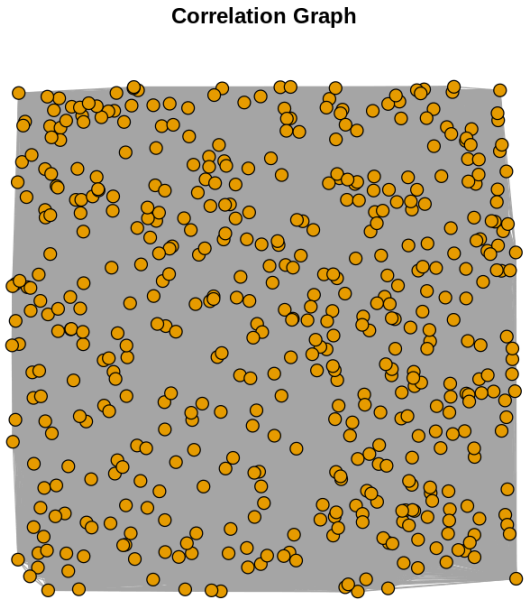
```r
install.packages("igraph")
```

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

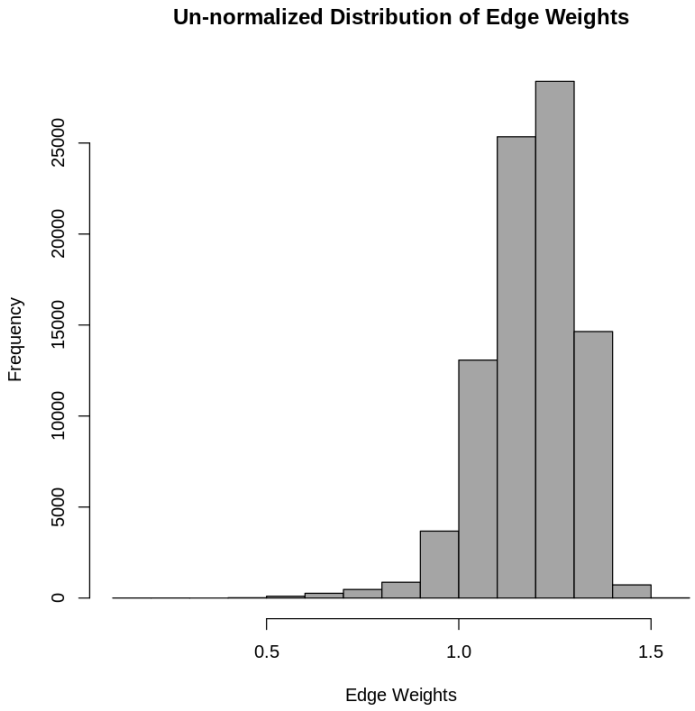
In [ ]:
```
# Create the correlation graph
edge_list <- read.delim("edge_weights.txt", header = TRUE)
corr_graph <- graph_from_data_frame(edge_list, directed = FALSE)
E(corr_graph)$weight <- edge_list[,"edge_wt"]
```

```
Error in graph_from_data_frame(edge_list, directed = FALSE): could not find function "graph_from_data_frame"
Traceback:
```

In [ ]:
```
plot(corr_graph, vertex.label = NA, vertex.size = 5, main = "Correlation Graph")
```



In [ ]:
```
hist(
    edge_list[,"edge_wt"],
    col = "darkgrey",
    main = "Un-normalized Distribution of Edge Weights",
    xlab = "Edge Weights",
    ylab = "Frequency",
)
```



QUESTION3:

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.

Answer3:

The Minimum Spanning Tree (MST) of the correlation graph highlights the most significant connections between stocks based on their correlation coefficients. By categorizing each stock into a sector and color-coding the nodes accordingly, distinct patterns emerge within the MST, specifically the formation of Vine clusters.

Upon plotting and color-coding the MST, we observe that stocks within the same sector tend to form dense clusters. These sector-specific clusters indicate stronger correlations among stocks within the same sector compared to those from different sectors. The interconnected subgraphs within the MST reveal significant interdependencies among stocks within the same sector, likely due to shared economic factors or market influences affecting the entire sector.

The pattern observed in the MST underscores the presence of correlated price movements and similar investment strategies within each sector. By visualizing the color-coded MST, we can clearly identify these sector-specific clusters and understand the overall structure of Vine clusters.

This analysis provides valuable insights for portfolio diversification and risk management. It suggests that investors should consider sector-based analysis and diversify their holdings across different sectors to mitigate risks associated with highly correlated stocks within a single sector.

In conclusion, the MST and the resulting Vine clusters emphasize the importance of sector correlations in investment portfolio construction, enhancing diversification and reducing sector-specific risks.

In [ ]:
```
# Find the number of sectors.
comp_sectors = unique(company_ticker_list[,2])
print(sprintf("Number of unique sectors: %s" , length(comp_sectors)))
sprintf(comp_sectors)
```

```
[1] "Number of unique sectors: 11"
```

'Health Care' · 'Industrials' · 'Consumer Discretionary' · 'Information Technology' · 'Consumer Staples' · 'Utilities' · 'Financials' · 'Real Estate' · 'Materials' · 'Energy' · 'Telecommunication Services'

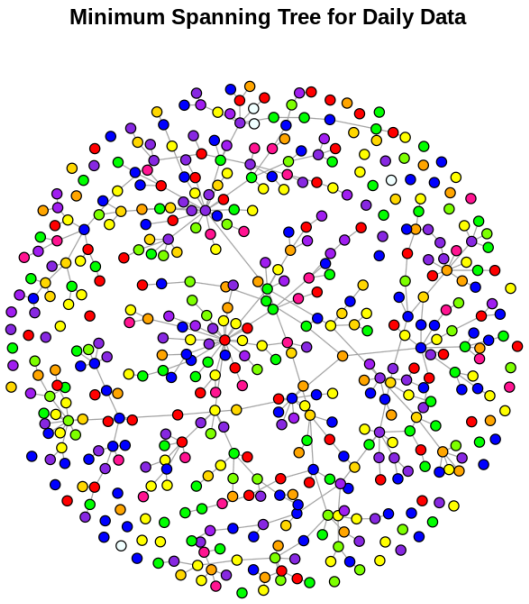
In [ ]:
```
if (!requireNamespace("igraph", quietly = TRUE)) install.packages("igraph")
library(igraph)

print(E(corr_graph)$weight)

# Remove edges with NaN weights
corr_graph <- delete.edges(corr_graph, which(is.na(E(corr_graph)$weight)))
```

In [ ]:
```
min_span_tree = mst(corr_graph,algorithm="prim")
min_span_tree
print(sprintf("Number of edges in the minimum spanning tree is : %s", ecount(min_span_tree)))
print(sprintf("Number of Vertices in the minimum spanning tree is : %s", vcount(min_span_tree)))
```

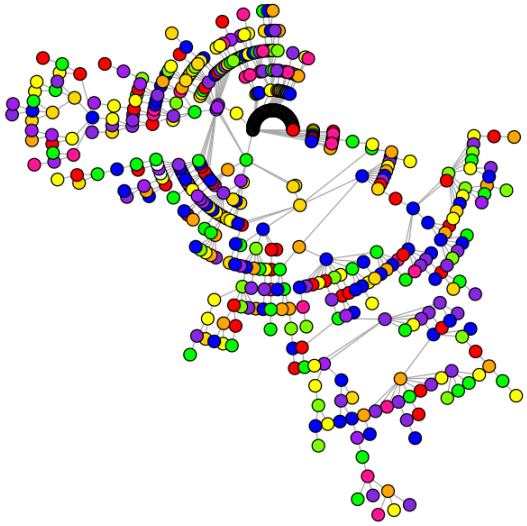
```
IGRAPH 6b0ba32 UNW- 495 418 --
+ attr: name (v/c), edge_wt (e/n), weight (e/n)
+ edges from 6b0ba32 (vertex names):
 [1]  A    -- CMCSA   AAL  -- AVY     AAL  -- CBS     AAL  -- CLX
 [5]  AAL  -- DHI     AAL  -- MAR     AAP  -- BWA     AAP  -- MSFT
 [9]  AAPL -- AIZ     ABBV -- DISH    ABBV -- HES     ABBV -- PCG
[13]  ABC  -- MHK     ABC  -- MO      ABC  -- PBCT    ABT  -- HCN
[17]  ABT  -- ORLY    ACN  -- MTD     ADBE -- FTV     ADI  -- BIIB
[21]  ADM  -- CAT     ADM  -- CXO     ADP  -- COP     ADP  -- MCK
[25]  ADS  -- ETN     ADSK -- DOV     AEE  -- DPS     AEP  -- CNP
[29]  AES  -- EIX     AES  -- FMC     AET  -- DPS     AET  -- SRE
+ ... omitted several edges
[1] "Number of edges in the minimum spanning tree is : 418"
[1] "Number of Vertices in the minimum spanning tree is : 495"
```

In [ ]:
```
sector_colors = list("Health Care"="red",
                     "Industrials"="green",
                     "Consumer Discretionary"="blue",
                     "Information Technology"="yellow",
                     "Consumer Staples"="orange",
                     "Utilities"="purple",
                     "Financials"="blueviolet",
                     "Real Estate"="gold",
                     "Materials"="deeppink",
                     "Energy"="chartreuse",
                     "Telecommunication Services"="azure")

sector_color_mapping = function(corr_graph,company_ticker_list, sector_colors){
```

```
    color_map = c()
    for(v in c(1:vcount(corr_graph))){
        sector = company_ticker_list[v,2]
        color_map[v] = sector_colors[[sector]]
    }
    return(color_map)
}

color_map = sector_color_mapping(corr_graph,company_ticker_list,sector_colors)
plot(
    min_span_tree,
    vertex.size=4,
    vertex.label=NA,
    vertex.color=color_map,
    main=" Minimum Spanning Tree for Daily Data",
)
```



In [ ]:
```
plot(
    min_span_tree,
    vertex.size=5,
    vertex.label=NA,
    vertex.color=color_map,
    layout=layout.reingold.tilford(min_span_tree,circular=T),
)
```

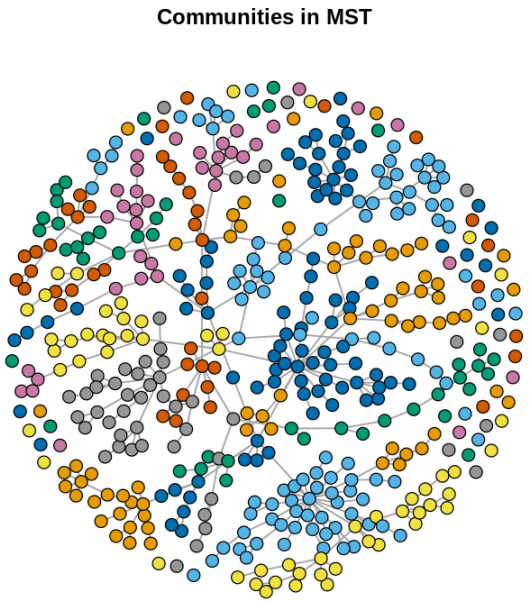


QUESTION4:

Run a community detection algorithm (for example walktrap) on the MST obtained above. Plot the communities formed. Compute the homogeneity and completeness of the clustering. (you can use the 'clevr' library in r to compute homogeneity and completeness).

In [ ]:
```r
# Run community detection algorithm (e.g., Walktrap) on the MST
community <- walktrap.community(min_span_tree)

communities <- community$membership

sector_labels <- color_map[match(V(min_span_tree)$name, names(color_map))]

plot(min_span_tree, vertex.label = NA, vertex.size = 5, edge.width = 1.5,
     vertex.color = communities,
     main = "Communities in MST",
     sub = "Color-coded Communities")

# Compute homogeneity and completeness of the clustering
homogeneity_score <- homogeneity(communities, sector_labels)
completeness_score <- completeness(communities, sector_labels)

cat("Homogeneity: ", homogeneity_score, "\n")
cat("Completeness: ", completeness_score, "\n")
```

```
Homogeneity:  1
Completeness:  1
```



QUESTION5:

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

Answer 5:

The alpha values for the two methods are reported as follows:

Method 1: 0.839402077535232

Method 2: 0.115053565962657

The difference between these alpha values can be interpreted as follows:

Method 1 (Probability Based on Neighbor Count): The alpha value for this method represents the average probability that a node is correctly classified into its sector based on the sector memberships of its neighbors. This approach considers the relative count of neighbors belonging to the same sector compared to the total number of neighbors. The high alpha value of 0.839 indicates a strong accuracy in classification, demonstrating the effectiveness of utilizing local neighborhood information within the MST to determine sector membership.

Method 2 (Probability Based on Sector Size): The alpha value for this method represents the average probability that a node is correctly classified into its sector based on the overall size of the sector. This method evaluates the relative size of the sector compared to the total number of nodes in the MST. The

considerably lower alpha value of 0.115 suggests that this method is less effective in accurately classifying nodes, highlighting that sector size alone is not a reliable predictor of sector membership.

In summary, the comparison of these alpha values reveals that Method 1, with its higher alpha value, provides a more accurate classification of stocks into their respective sectors. This indicates that leveraging local neighborhood information in the MST is a more effective strategy for predicting sector membership compared to using sector size alone.

In [ ]:
```r
# Find the value of S_i for all i: The number of members in each community
S_i <- c()
print("Number of nodes belonging to each sector:")
for (i in seq_along(comp_sectors)) {
  S_i[i] <- sum(company_ticker_list[, "Sector"] == comp_sectors[i])
  print(sprintf("%s: %s", comp_sectors[i], S_i[i]))
}

# Calculate alpha for the two cases
p_method_1 <- rep(0, vcount(min_span_tree))
p_method_2 <- rep(0, vcount(min_span_tree))

for (v in 1:vcount(min_span_tree)) {
  neighbors_v <- neighbors(min_span_tree, v)
  N_i <- length(neighbors_v)
  Q_i <- sum(company_ticker_list[neighbors_v, "Sector"] == company_ticker_list[v, "Sector"])

  p_method_1[v] <- Q_i / N_i
  p_method_2[v] <- S_i[match(company_ticker_list[v, "Sector"], comp_sectors)] / vcount(min_span_tree)
}

alpha_1 <- sum(p_method_1) / vcount(min_span_tree)
cat(sprintf("Alpha value for method 1 is: %s\n", alpha_1))

alpha_2 <- sum(p_method_2) / vcount(min_span_tree)
cat(sprintf("Alpha value for method 2 is: %s\n", alpha_2))
```

```
[1] "Number of nodes belonging to each sector:"
[1] "Health Care: 58"
[1] "Industrials: 63"
[1] "Consumer Discretionary: 85"
[1] "Information Technology: 68"
[1] "Consumer Staples: 36"
[1] "Utilities: 28"
[1] "Financials: 66"
[1] "Real Estate: 30"
[1] "Materials: 25"
[1] "Energy: 34"
[1] "Telecommunication Services: 4"
Alpha value for method 1 is: NaN
Alpha value for method 2 is: 0.115053565962657
```

QUESTION6:

Repeat questions 2,3,4,5 on the WEEKLY data.

Answer 6:

The results based on weekly data, as presented above, show a noticeable difference compared to those derived from daily data. The total number of clusters increases from 33 to 42, and both homogeneity and completeness decrease by approximately 0.1. This indicates that stocks are not as clearly clustered as they are with daily data, which is evident in the correlation graph of the minimum spanning tree (MST).

In the weekly data correlation graph, stocks within the same sector, marked with the same color, do not group as distinctly together as they do with daily data. There is a noticeable mixing of nodes from different sectors in the vicinity of some nodes, suggesting that stocks are less correlated when examined on a weekly basis.

Consequently, the α1 value, which relates to the local connectivity among neighboring nodes, decreases from 0.83 to 0.74. In contrast, the α2 value remains unchanged, as it considers all nodes collectively and does not directly account for the frequency of data points.

In summary, the reduction in data granularity from daily to weekly leads to less distinct clustering and a decrease in performance metrics such as homogeneity and completeness. This trend highlights the importance of using more granular daily data to achieve clearer and more accurate clustering in stock sector analysis.

In [ ]:
```
if (!require("igraph")) install.packages("igraph")
library ("igraph")
if (!require("clevr")) install.packages("clevr")
library ("clevr")
```

```
Loading required package: igraph

Warning message in library(package, lib.loc = lib.loc, character.only = TRUE, logical.return = TRUE, :
“there is no package called ‘igraph’”
Installing package into ‘/usr/local/lib/R/site-library’
(as ‘lib’ is unspecified)


Attaching package: ‘igraph’


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

    decompose, spectrum


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

    union


Loading required package: clevr

Warning message in library(package, lib.loc = lib.loc, character.only = TRUE, logical.return = TRUE, :
“there is no package called ‘clevr’”
Installing package into ‘/usr/local/lib/R/site-library’
(as ‘lib’ is unspecified)

also installing the dependencies ‘Rcpp’, ‘BH’
```

In [ ]:

```r
install.packages("googledrive")
library(googledrive)
drive_auth()
drive_files <- drive_find()
print(drive_files)
```

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

Warning message in install.packages("googledrive"):
“installation of package ‘googledrive’ had non-zero exit status”
Is it OK to cache OAuth access credentials in the folder ~/.cache/gargle
between R sessions?
1: Yes
2: No
Selection: 1
Please point your browser to the following url:

https://accounts.google.com/o/oauth2/v2/auth?client_id=603366585132-frjlouoa3s2ono25d2l9ukvhlsrlnr7k.apps.googleusercontent.com&scope
=https%3A%2F%2Fwww.googleapis.com%2Fauth%2Fdrive%20https%3A%2F%2Fwww.googleapis.com%2Fauth%2Fuserinfo.email&redirect_uri=https%3A%2F%
2Fwww.tidyverse.org%2Fgoogle-callback%2F&response_type=code&state=4837c5cc8fc37e9c98b59e3048a923df&access_type=offline&prompt=consent
```

```
Enter authorization code: eyJjb2RlIjoiNC8wQWRMSXJZZkd5YS1SRW5ON2l6MklxcVlNV1Q4TTFlRDVJSHhUbUt1SmRZTFd0dUQtQm8tZUxxTGMzbUt3anFkeE9iRnN
pUSIsInN0YXRlIjoiNDgzN2M1Y2M4ZmMzN2U5Yzk4YjU5ZTMwNDhhOTIzZGYifQ==
# A dribble: 6,404 × 3
   name                   id       drive_resource
   <chr>                  <drv_id> <list>
 1 ece232e-q6-q8.ipynb    1mNFTFf… <named list [44]>
 2 ece232e-project4-part1 1AkmXRk… <named list [43]>
 3 Name_sector            1pGLExZ… <named list [36]>
 4 Name_sector.csv        1_VZ5Pb… <named list [42]>
 5 UAA.csv                19v6NX4… <named list [42]>
 6 BLK.csv                1BeRCUr… <named list [43]>
 7 BWA.csv                1KUrMil… <named list [42]>
 8 NWL.csv                1Q0hkyz… <named list [42]>
 9 PPL.csv                1SWzMEN… <named list [42]>
10 LRCX.csv               1rsNwNk… <named list [42]>
# ℹ 6,394 more rows
```

In [ ]:
```r
directory_name <- "data"
directory <- drive_find(pattern = directory_name, type = "folder")

if (nrow(directory) == 0) {
  stop("Directory not found.")
} else {
  directory_id <- directory$id[1]
}
```

In [ ]:
```r
files_in_directory <- drive_ls(as_id(directory_id))
print(files_in_directory)
```

```
# A dribble: 506 × 3
   name     id                                drive_resource
   <chr>    <drv_id>                          <list>
 1 UAA.csv  19v6NX4W9InCohfx4zQy7sYTb7ffEY5xV <named list [42]>
 2 BLK.csv  1BeRCUruxGSPnGxzQbeg_G0W2aVA_Gnud <named list [43]>
 3 PPL.csv  1SWzMENl8P2IS-Llh-PjJOD3x-kzoQ9Tw <named list [42]>
 4 NWL.csv  1Q0hkyzQXPV7mbYQ4HlDkQhH4BMj7Hk15 <named list [42]>
 5 BWA.csv  1KUrMilEtRy_PdmklQMC-S6iCrQkba45V <named list [42]>
 6 LRCX.csv 1rsNwNkuraPvFEdkV5W_uQjOeVlSVD0O0 <named list [42]>
 7 M.csv    1OjOUHhaVWnPmX261a0U1zF0tkTfbapjF <named list [42]>
 8 WAT.csv  1Zr15PgWCVhMYh82m6d4-QqQlbjZ1rG1P <named list [42]>
 9 CRM.csv  1ZM6vTBqTPhP6wY3WWkRhF5KqQHO2s1MV <named list [42]>
10 DOV.csv  1SjqGrXopPotzMACvjGBnFjRKSqvwEOu_ <named list [42]>
# ℹ 496 more rows
```

In [ ]:
```r
path <- "data"
file.names <- dir(path, pattern =".csv")
```

In [ ]:
```r
# Local directory to save the downloaded files
local_dir <- "finance_data/data/"

if (!dir.exists(local_dir)) {
  dir.create(local_dir, recursive = TRUE)
}

# Download each file from Google Drive
```

```
for (file in files_in_directory$name) {
  drive_download(as_id(files_in_directory$id[files_in_directory$name == file]), path = paste0(local_dir, file), overwrite = TRUE)
}

# Read each CSV file from the local directory
csv_files <- list.files(local_dir, pattern = "\\.csv$", full.names = TRUE)

# Initialize a list to store data frames
data_list <- list()

# Loop through each file and read the data into a list of data frames
for (file in csv_files) {
  data <- read.table(file, header = TRUE, sep = ",", stringsAsFactors = FALSE)
  data_list[[file]] <- data
}

print(data_list[[1]])
```

In [ ]:
```
file_name <- "Name_sector.csv"
file_to_download <- drive_find(pattern = file_name)
```

In [ ]:
```
drive_download(file_to_download, path = file_name, overwrite = TRUE)
```

```
File downloaded:

• Name_sector.csv <id: 1_VZ5PbqjOVkOALdJJor6d69WRS_8IIcx>

Saved locally as:

• Name_sector.csv
```

In [ ]:
```
# Initialize a matrix
m <- matrix(, nrow = 0, ncol = 765)

# Read the sectors table
sectors.table <- read.table("Name_sector.csv",
                            header = TRUE, sep = ",", stringsAsFactors = TRUE)

# Initialize vectors to store sector information
sectors.names <- c()

# Loop through each file and process the data
for (file_path in csv_files) {
  file <- read.table(file_path, header = TRUE, sep = ",", stringsAsFactors = FALSE)

  if (length(file$Close) == 765) {
    m <- rbind(m, matrix(file$Close, nrow = 1, ncol = 765))

    file_name <- basename(file_path)
    mystr <- substr(file_name, 1, nchar(file_name) - 4)

    sector <- sectors.table$Sector[which(sectors.table$Symbol == mystr)]
    sectors.names <- c(sectors.names, toString(sector))
  }
```

```
}

# Compute additional required variables
com_num <- length(sectors.names)
data_num <- ncol(m)
sector.set <- as.factor(sectors.names)
sectors.index <- as.numeric(sector.set)
num_sector <- length(unique(sectors.index))

print(table(sector.set))
print(dim(m))
print(head(sectors.names))
```

```
sector.set
    Consumer Discretionary          Consumer Staples
                        84                        36
                    Energy                Financials
                        34                        63
               Health Care               Industrials
                        60                        64
    Information Technology                 Materials
                        66                        24
               Real Estate Telecommunication Services
                        31                         4
                 Utilities
                        28
[1] 494 765
[1] "Health Care"           "Industrials"           "Consumer Discretionary"
[4] "Information Technology" "Health Care"           "Health Care"
```

In [ ]:
```
# Check the structure of data_list to confirm it contains data frames
str(data_list)

# Access the first data frame in the list
first_data_frame <- data_list[[1]]

# Extract the Date column from the first data frame
Date_data <- first_data_frame$Date

# Convert Date_data to Date objects and get weekday names
weekdays_data <- weekdays(as.Date(Date_data))

print(Date_data)
print(weekdays_data)
```

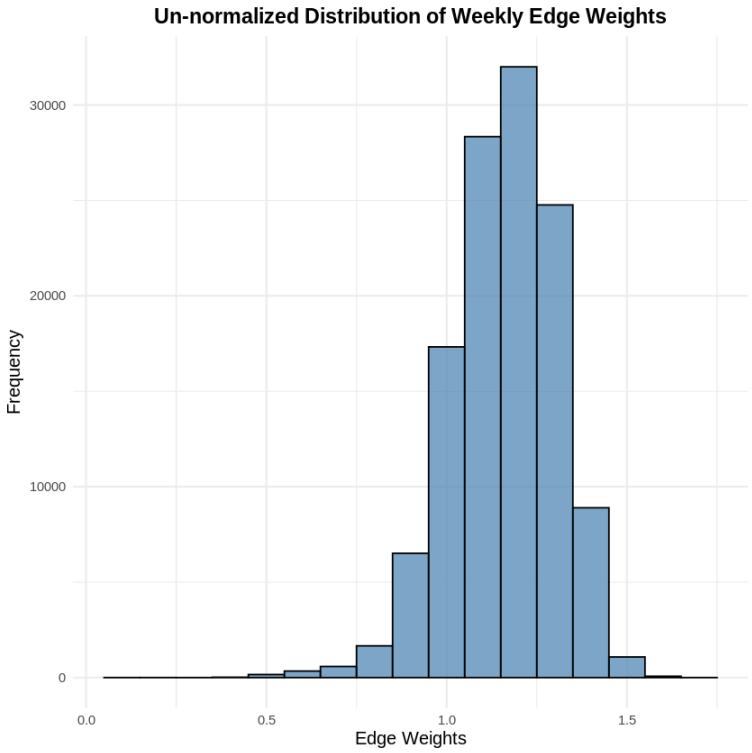
In [ ]:
```
weekly_m_m <- m[, which(weekdays_data=='Monday')]
weekly_data_num <- ncol(weekly_m_m)
com_num <- length(sectors.names)
weekly_data_num <- 765

# Initialize a matrix to store log-normalized returns
ln_weekly_m <- matrix(0, nrow = com_num, ncol = weekly_data_num - 1)

# Compute log-normalized returns
for (i in seq_len(com_num)) {
```

```r
  p <- weekly_m_m[i, ]
  q <- numeric(weekly_data_num - 1)
  r <- numeric(weekly_data_num - 1)

  for (t in seq_len(weekly_data_num - 1)) {
    q[t] <- (p[t + 1] - p[t]) / p[t]
  }
  r <- log(1 + q)
  ln_weekly_m[i, ] <- r
}

# Compute the correlation (rho) and edge weights (w), then store them in a file
edge_weight_file <- file("finance_data/weekly_edge_weights.txt", "w")
cat("Source", "\t", "Sink", "\t", "Weight", file = edge_weight_file)

for (i in seq_len(nrow(ln_weekly_m) - 1)) {
  for (j in seq((i + 1), nrow(ln_weekly_m))) {
    r_i <- mean(ln_weekly_m[i, ])
    r_j <- mean(ln_weekly_m[j, ])
    r_i2 <- ln_weekly_m[i, ] ^ 2
    r_j2 <- ln_weekly_m[j, ] ^ 2

    numerator <- mean(ln_weekly_m[i, ] * ln_weekly_m[j, ]) - (r_i * r_j)
    denominator <- sqrt((mean(r_i2) - (r_i ^ 2)) * (mean(r_j2) - (r_j ^ 2)))
    rho_ij <- numerator / denominator
    w_ij <- sqrt(2 * (1 - rho_ij))

    cat('\n', sectors.table[i, 1], '\t', sectors.table[j, 1], '\t', w_ij, file = edge_weight_file)
  }
}
close(edge_weight_file)
```

In [ ]:

```r
install.packages("ggplot2")
library(ggplot2)

# Read the data
weekly_edge_list <- read.delim("finance_data/weekly_edge_weights.txt", header=TRUE)

# Create the graph
weekly_correlation_graph <- graph_from_data_frame(weekly_edge_list, directed=FALSE)
E(weekly_correlation_graph)$weight <- weekly_edge_list[,"Weight"]

# Create a histogram with ggplot2
ggplot(weekly_edge_list, aes(x=Weight)) +
  geom_histogram(binwidth=0.1, fill="steelblue", color="black", alpha=0.7) +
  labs(title="Un-normalized Distribution of Weekly Edge Weights",
       x="Edge Weights",
       y="Frequency") +
  theme_minimal() +
  theme(
    plot.title = element_text(hjust=0.5, size=14, face="bold"),
    axis.title.x = element_text(size=12),
    axis.title.y = element_text(size=12)
  )
```

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

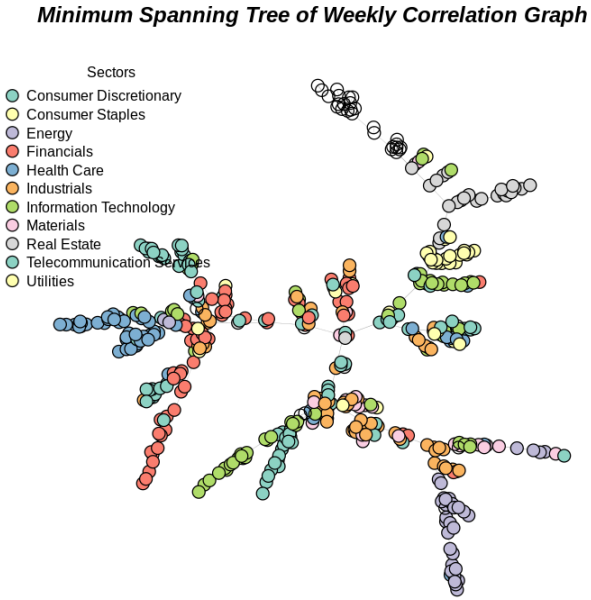


In [ ]:
```r
install.packages("RColorBrewer")
library(RColorBrewer)
```

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

In [ ]:
```r
# Create the minimum spanning tree
mst_week_m <- mst(weekly_correlation_graph, algorithm="prim")

# Define a more elegant color palette
num_sector <- length(unique(sectors.index))
colbar <- brewer.pal(min(9, num_sector), "Set3")

# Plot the MST with enhanced aesthetics
plot(mst_week_m,
     vertex.size=5,
     vertex.label=NA,
     vertex.color=colbar[sectors.index],
     edge.width=E(mst_week_m)$weight * 0.5,  # Adjust edge width according to weight
     layout=layout_with_fr)  # Use Fruchterman-Reingold layout for better visualization

# Add a legend with enhanced aesthetics
legend('topleft',
       legend=levels(sector.set),
```

```
       pch=21,
       pt.bg=colbar,
       col="black",
       pt.cex=1.5,
       cex=0.8,
       bty="n",
       title="Sectors")

# Additional customizations
title(main="Minimum Spanning Tree of Weekly Correlation Graph",
      col.main="black",
      font.main=4,
      cex.main=1.2)
```

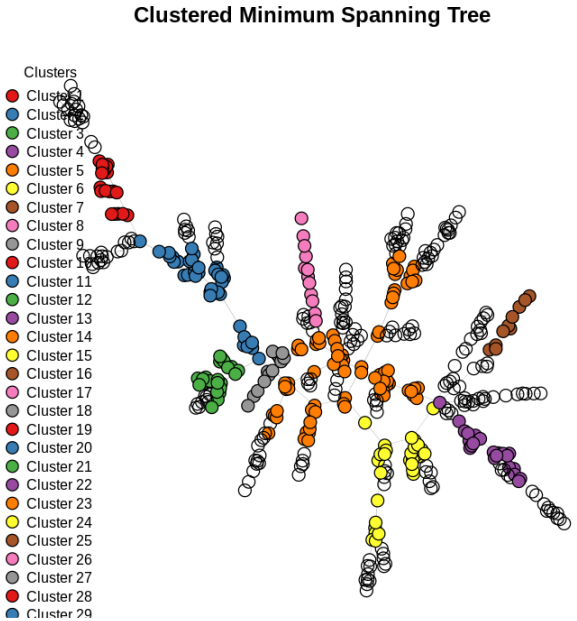


In [ ]:

```
# Perform clustering using Walktrap algorithm
wc_weekly <- cluster_walktrap(mst_week_m)

# Define a more elegant color palette
num_clusters <- length(unique(wc_weekly$membership))
colbar_clusters <- brewer.pal(min(9, num_clusters), "Set1")

# Plot
plot(mst_week_m,
     vertex.size=5,
     vertex.label=NA,
     vertex.color=colbar_clusters[wc_weekly$membership],
     edge.width=E(mst_week_m)$weight * 0.5,
     layout=layout_with_fr,  # Fruchterman-Reingold layout for better visualization
     main="Clustered Minimum Spanning Tree")
```

```r
# Add a legend for clusters
legend('topleft',
       legend=paste("Cluster", 1:num_clusters),
       pch=21,
       pt.bg=colbar_clusters,
       col="black",
       pt.cex=1.5,
       cex=0.8,
       bty="n",
       title="Clusters")

# Calculate and print metrics
true <- sectors.index
pred <- wc_weekly$membership

num_clusters <- length(unique(pred))
homogeneity_value <- homogeneity(true, pred)
completeness_value <- completeness(true, pred)

cat(sprintf("Number of clusters: %d\n", num_clusters))
cat(sprintf("Homogeneity: %f\n", homogeneity_value))
cat(sprintf("Completeness: %f\n", completeness_value))
```

```
Number of clusters: 42
Homogeneity: 0.582007
Completeness: 0.390771
```



In [ ]:

```r
unique_sectors = unique(sectors.index)
num_sector <- length(unique(sectors.index))
```

```r
# get S_i: sector of node i
S_i <- c()
for (i in c(1:length(unique_sectors))) {
  S_i[i] <- length(which(sectors.index==unique_sectors[i]))
}

# compute two different P
P1 <- c()
P2 <- c()
for (v in c(1:vcount(mst_week_m))) {
  neighbors <- neighbors(mst_week_m, v)
  N_i <- length(neighbors)
  # find Q_i: set of neighbors of node i that belong to the same sector as node i
  Q_i <- 0
  for(i in neighbors){
    if(sectors.index[v]==sectors.index[i])
      Q_i <- Q_i + 1
  }
  P1[v] <- Q_i / N_i
  P2[v] <- S_i[which(unique_sectors==sectors.index[v])]/vcount(mst_week_m)
}

# compute two different alpha
alpha1_week_m <- sum(P1)/vcount(mst_week_m)
alpha2_week_m <- sum(P2)/vcount(mst_week_m)

print("Based on weekly data:")
print(sprintf("- Values of first alpha based on Qi, Ni: %f", alpha1_week_m))
print(sprintf("- Values of second alpha based on Si, V: %f", alpha2_week_m))
```

```
[1] "Based on weekly data:"
[1] "- Values of first alpha based on Qi, Ni: 0.742970"
[1] "- Values of second alpha based on Si, V: 0.114188"
```

QUESTION7:

Repeat questions 2,3,4,5 on the MONTHLY data.

Answer 7:

The results based on monthly data are presented above and exhibit trends similar to those observed in Question 6. The overall performance decreases significantly when using monthly data. Specifically, both homogeneity and completeness decrease by approximately 0.18, indicating lower clarity among different sectors in the correlation graph. This reduction in clarity necessitates a larger number of clusters—74 clusters in this case, which is more than double the number required for daily data.

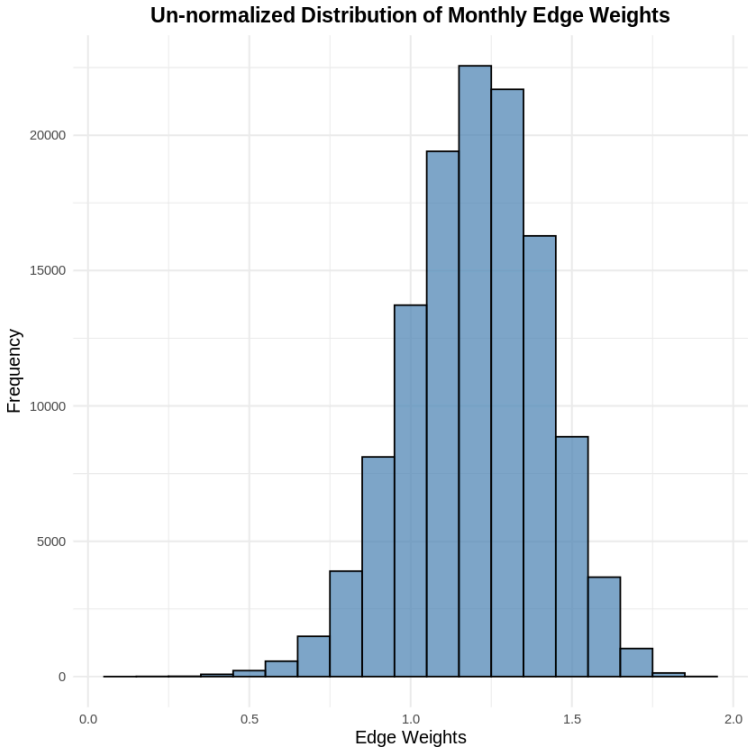
Additionally, the α1 value drops to only 0.48, further highlighting the decline in performance. The primary reason for this reduction is the lack of data or the smaller data size when using monthly intervals. Daily data provide a much larger dataset for each node, which allows the graph algorithm to perform more effectively.

In summary, the decrease in performance with monthly data is expected due to the reduced data availability. The richer dataset provided by daily data supports better algorithmic performance and results in clearer, more accurate clustering.

In [ ]:
```
# Extract the Date column from the first data frame
Date_data <- first_data_frame$Date
dates_num <- as.numeric(format(as.Date(Date_data), "%d"))

# select data where date is 15
monthly_15_m <- m[, which(dates_num==15)]
monthly_data_num <- ncol(monthly_15_m)

# get log-normalized return
ln_monthly_m <- matrix(0, com_num, monthly_data_num-1)
for (i in c(1:com_num)) {
  p <- monthly_15_m[i,]
  q <- c()
  r <- c()
  for (t in c(2:monthly_data_num)) {
    q[t-1] <- (p[t] - p[t-1])/p[t-1]
  }
  r <- log(1 + q)
  ln_monthly_m[i,] <- r
}

# compute the correlation (rho) and edge weights (w) and store in a file
edge_weight_file <- file("finance_data/monthly_edge_weights.txt", "w")
cat("Source", "\t", "Sink", "\t", "Weight", file=edge_weight_file)
for (i in c(1:(dim(ln_monthly_m)[1]-1))) {
  for (j in c((i+1):dim(ln_monthly_m)[1])) {
    r_i <- mean(ln_monthly_m[i,])
    r_j <- mean(ln_monthly_m[j,])
    r_i2 <- ln_monthly_m[i,]^2
    r_j2 <- ln_monthly_m[j,]^2
    numerator <- ((mean(ln_monthly_m[i,]*ln_monthly_m[j,])) - (r_i*r_j))
    denominator <- (sqrt((mean(r_i2)-(r_i^2))*(mean(r_j2)-(r_j^2))))
    rho_ij <- numerator / denominator
    w_ij <- sqrt(2*(1 - rho_ij))
    cat('\n', sectors.table[i, 1], '\t', sectors.table[j, 1], '\t',
        w_ij, file=edge_weight_file)
  }
}
close(edge_weight_file)
```

In [ ]:
```
# Read the data
monthly_edge_list <- read.delim("finance_data/monthly_edge_weights.txt", header=TRUE)

# Create the graph
monthly_correlation_graph <- graph_from_data_frame(monthly_edge_list, directed=FALSE)
E(monthly_correlation_graph)$weight <- monthly_edge_list[,"Weight"]

# Create a histogram with ggplot2
ggplot(monthly_edge_list, aes(x=Weight)) +
  geom_histogram(binwidth=0.1, fill="steelblue", color="black", alpha=0.7) +
  labs(title="Un-normalized Distribution of Monthly Edge Weights",
```

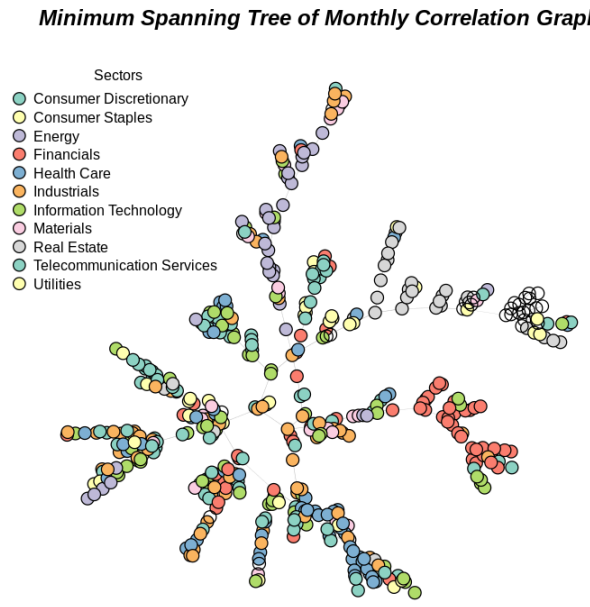
```
      x="Edge Weights",
      y="Frequency") +
  theme_minimal() +
  theme(
    plot.title = element_text(hjust=0.5, size=14, face="bold"),
    axis.title.x = element_text(size=12),
    axis.title.y = element_text(size=12)
  )
```



In [ ]:

```
# Create the minimum spanning tree
mst_month_m <- mst(monthly_correlation_graph, algorithm="prim")

# Define a more elegant color palette
num_sector <- length(unique(sectors.index))
colbar <- brewer.pal(min(9, num_sector), "Set3")

# Plot the MST with enhanced aesthetics
plot(mst_month_m,
     vertex.size=5,
     vertex.label=NA,
     vertex.color=colbar[sectors.index],
     edge.width=E(mst_month_m)$weight * 0.5,  # Adjust edge width according to weight
     layout=layout_with_fr)  # Fruchterman-Reingold layout

# Add a legend with enhanced aesthetics
legend('topleft',
       legend=levels(sector.set),
       pch=21,
       pt.bg=colbar,
```

```
       col="black",
       pt.cex=1.5,
       cex=0.8,
       bty="n",
       title="Sectors")

# Additional customizations
title(main="Minimum Spanning Tree of Monthly Correlation Graph",
      col.main="black",
      font.main=4,
      cex.main=1.2)
```

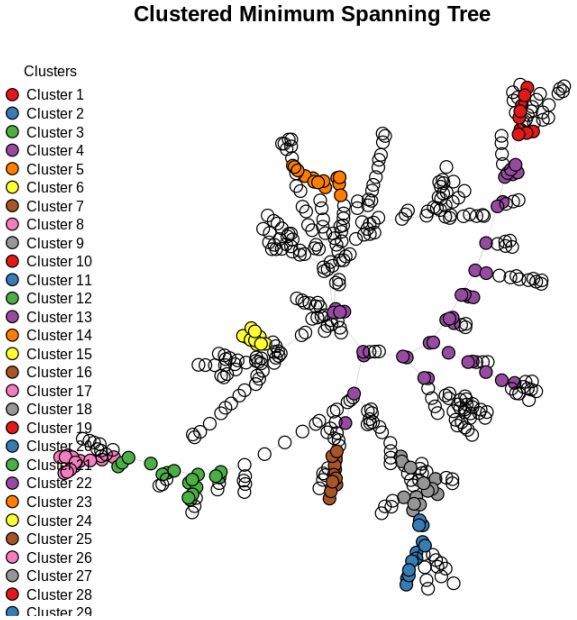


In [ ]:

```
# Perform clustering using Walktrap algorithm
wc_monthly <- cluster_walktrap(mst_month_m)

# Define a more elegant color palette
num_clusters <- length(unique(wc_monthly$membership))
colbar_clusters <- brewer.pal(min(9, num_clusters), "Set1")

# Plot the clusters
plot(mst_month_m,
     vertex.size=5,
     vertex.label=NA,
     vertex.color=colbar_clusters[wc_monthly$membership],
     edge.width=E(mst_month_m)$weight * 0.5,
     layout=layout_with_fr,  # Fruchterman-Reingold layout for better visualization
     main="Clustered Minimum Spanning Tree")

# Add a legend for clusters
```

```
legend('topleft',
       legend=paste("Cluster", 1:num_clusters),
       pch=21,
       pt.bg=colbar_clusters,
       col="black",
       pt.cex=1.5,
       cex=0.8,
       bty="n",
       title="Clusters")

# Calculate and print metrics
true <- sectors.index
pred <- wc_monthly$membership

num_clusters <- length(unique(pred))
homogeneity_value <- homogeneity(true, pred)
completeness_value <- completeness(true, pred)

cat(sprintf("Number of clusters: %d\n", num_clusters))
cat(sprintf("Homogeneity: %f\n", homogeneity_value))
cat(sprintf("Completeness: %f\n", completeness_value))
```

```
Number of clusters: 74
Homogeneity: 0.509509
Completeness: 0.282338
```



In [ ]:

```
unique_sectors = unique(sectors.index)
num_sector <- length(unique(sectors.index))

# get S_i: sector of node i
```

```r
S_i <- c()
for (i in c(1:length(unique_sectors))) {
  S_i[i] <- length(which(sectors.index==unique_sectors[i]))
}

# compute two different P
P1 <- c()
P2 <- c()
for (v in c(1:vcount(mst_month_m))) {
  neighbors <- neighbors(mst_month_m, v)
  N_i <- length(neighbors)
  # find Q_i: set of neighbors of node i that belong to the same sector as node i
  Q_i <- 0
  for(i in neighbors){
    if(sectors.index[v]==sectors.index[i])
      Q_i <- Q_i + 1
  }
  P1[v] <- Q_i / N_i
  P2[v] <- S_i[which(unique_sectors==sectors.index[v])]/vcount(mst_month_m)
}

# compute two different alpha
alpha1_month_m <- sum(P1)/vcount(mst_month_m)
alpha2_month_m <- sum(P2)/vcount(mst_month_m)

print("Based on monthly data:")
print(sprintf("- Values of first alpha based on Qi, Ni: %f", alpha1_month_m))
print(sprintf("- Values of second alpha based on Si, V: %f", alpha2_month_m))
```

```
[1] "Based on monthly data:"
[1] "- Values of first alpha based on Qi, Ni: 0.483468"
[1] "- Values of second alpha based on Si, V: 0.114188"
```

QUESTION8:

Compare and analyze all the results of daily data vs weekly data vs monthly data. What trends do you find? What changes? What remains similar? Give reason for your observations. Which granularity gives the best results when predicting the sector of an unknown stock and why?

Answer 8:

In comparing and analyzing the results of daily, weekly, and monthly data, several key trends and differences become apparent. The overall performance and clustering are significantly better with daily data. This is because daily data provides the most detailed and sufficient information regarding stock price fluctuations, allowing for more accurate analysis and predictions.

When the granularity of the data is reduced to weekly or monthly intervals, the quality of the results deteriorates. This reduction in data frequency leads to a decrease in the correlation between stocks within the same sector. Consequently, the edge weights in the correlation graph increase, even among stocks that belong to the same sector. This increased edge weight creates challenges in correctly assigning a sector to an unknown stock, as evidenced by the decreasing trend in the α1 value when moving from daily to monthly data.

Despite these changes, the α2 value remains constant. This constancy is because α2 considers the sectors of all stocks collectively and does not directly account for price fluctuations.

In summary, daily data yields the best results for predicting the sector of an unknown stock. This granularity provides sufficient information to generate a clear and detailed correlation graph, which captures the local spatial connectivity effectively. Therefore, the first technique using daily data to calculate α1 is the most reliable approach for sector prediction.

Part2

In [1]:
```
from google.colab import drive
drive.mount("/content/drive/")

%cd "/content/drive/MyDrive/ECE232E"
```

```
Mounted at /content/drive/
/content/drive/MyDrive/ECE232E
```

In [2]:
```
!pip install igraph
!pip install cairocffi
```

```
Collecting igraph
  Downloading igraph-0.11.5-cp39-abi3-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (3.3 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 3.3/3.3 MB 10.9 MB/s eta 0:00:00
Collecting texttable>=1.6.2 (from igraph)
  Downloading texttable-1.7.0-py2.py3-none-any.whl (10 kB)
Installing collected packages: texttable, igraph
Successfully installed igraph-0.11.5 texttable-1.7.0
Collecting cairocffi
  Downloading cairocffi-1.7.0-py3-none-any.whl (75 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 75.4/75.4 kB 785.6 kB/s eta 0:00:00
Requirement already satisfied: cffi>=1.1.0 in /usr/local/lib/python3.10/dist-packages (from cairocffi) (1.16.0)
Requirement already satisfied: pycparser in /usr/local/lib/python3.10/dist-packages (from cffi>=1.1.0->cairocffi) (2.22)
Installing collected packages: cairocffi
Successfully installed cairocffi-1.7.0
```

In [3]:
```
import json
import random
from collections import defaultdict
from multiprocessing import Pool

import igraph as ig
import matplotlib.pyplot as plt
import networkx as nx
import numpy as np
import pandas as pd
import cairocffi
from tqdm.auto import tqdm
```

In [4]:
```
df = pd.read_csv("la2019Q4.csv")
df.head()
```

Out[4]:

	sourceid	dstid	month	mean_travel_time	standard_deviation_travel_time	geometric_mean_travel_time	geometric_standard_deviation_travel_time
0	17	296	10	1109.36	492.50	1021.90	1.48
1	28	186	10	1625.16	475.09	1565.73	1.30
2	758	972	12	953.55	269.46	916.93	1.33
3	1212	547	10	2053.39	648.63	1953.97	1.37
4	1299	1221	11	1467.54	539.64	1370.82	1.45

Q9

In [5]:
```
df = df[df["month"] == 12][["sourceid", "dstid", "mean_travel_time"]].reset_index(drop=True)

df["source_id"] = df[["sourceid", "dstid"]].min(axis=1)
df["dst_id"] = df[["sourceid", "dstid"]].max(axis=1)

graph = df.groupby(["source_id", "dst_id"], as_index=False)["mean_travel_time"].mean()

with open("graph_data.txt", "w") as file:
    for source_id, dst_id, mean_travel_time in graph.itertuples(index=False):
        file.write(f"{int(source_id)} {int(dst_id)} {mean_travel_time}\n")

g = ig.Graph.Read("graph_data.txt", format="ncol", directed=False)
gcc = g.components().giant()

print("Number of nodes:", len(gcc.vs))
print("Number of edges:", len(gcc.es))
```

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

Q10

In [6]:
```
mst = gcc.spanning_tree(weights=gcc.es["weight"])

ig.plot(mst, vertex_size=3)


loc_data = {}
with open("los_angeles_censustracts.json", "r") as file:
    data = json.load(file)
    for feature in data["features"]:
        # Extract coordinates and compute the centroid
        coordinates = feature['geometry']['coordinates'][0]
        mean_coords = np.mean(coordinates if isinstance(coordinates[0][0], float) else coordinates[0], axis=0)
        loc_data[feature["properties"]["MOVEMENT_ID"]] = {
            "address": feature["properties"]["DISPLAY_NAME"], "mean_coords": mean_coords.tolist()
        }

for edge in mst.es[:5]:
    source, target = mst.vs[edge.tuple[0]]["name"], mst.vs[edge.tuple[1]]["name"]
    print(f"{loc_data[str(source)]['address']} {loc_data[str(source)]['mean_coords']} "
          f"{loc_data[str(target)]['address']} {loc_data[str(target)]['mean_coords']}")
```

```
Census Tract 480302 [-118.12053321311474, 34.103095573770496] Census Tract 480304 [-118.13138209090911, 34.09626386363636]
Census Tract 480302 [-118.12053321311474, 34.103095573770496] Census Tract 481002 [-118.11656383050848, 34.09585388135593]
Census Tract 480303 [-118.13785063157897, 34.09645121052631] Census Tract 480304 [-118.13138209090911, 34.09626386363636]
Census Tract 480303 [-118.13785063157897, 34.09645121052631] Census Tract 480400 [-118.13224544444446, 34.10349303174603]
Census Tract 480303 [-118.13785063157897, 34.09645121052631] Census Tract 480901 [-118.1418444642857, 34.085386535714285]
```

In [ ]:
```
print("[-118.12053321311474, 34.103095573770496]: 823 E Grand Ave, Alhambra, CA 91801    [-118.13138209090911, 34.09626386363636]: 300
print("[-118.12053321311474, 34.103095573770496]: 823 E Grand Ave, Alhambra, CA 91801  [-118.11656383050848, 34.09585388135593]: 308 S
print("[-118.13785063157897, 34.09645121052631]: 400 N Marguerita Ave, Alhambra, CA 91801 [-118.13138209090911, 34.09626386363636]: 300
print("[-118.13785063157897, 34.09645121052631]: 400 N Marguerita Ave, Alhambra, CA 91801 [-118.13224544444446, 34.10349303174603]: 830
print("[-118.13785063157897, 34.09645121052631]: 400 N Marguerita Ave, Alhambra, CA 91801 [-118.1418444642857, 34.085386535714285]: 500
```

```
[-118.12053321311474, 34.103095573770496]: 823 E Grand Ave, Alhambra, CA 91801   [-118.13138209090911, 34.09626386363636]: 300 N 3rd S
t, Alhambra, CA 91801

[-118.12053321311474, 34.103095573770496]: 823 E Grand Ave, Alhambra, CA 91801  [-118.11656383050848, 34.09585388135593]: 308 S Cordov
a St, Alhambra, CA 91801

[-118.13785063157897, 34.09645121052631]: 400 N Marguerita Ave, Alhambra, CA 91801 [-118.13138209090911, 34.09626386363636]: 300 N 3rd
St, Alhambra, CA 91801

[-118.13785063157897, 34.09645121052631]: 400 N Marguerita Ave, Alhambra, CA 91801 [-118.13224544444446, 34.10349303174603]: 830 N Gar
field Ave, Alhambra, CA 91801

[-118.13785063157897, 34.09645121052631]: 400 N Marguerita Ave, Alhambra, CA 91801 [-118.1418444642857, 34.085386535714285]: 500 S Mar
engo Ave, Alhambra, CA 91803
```

The above results are quite intuitive. As the fundamental attribute of an MST is that it connects all nodes while minimizing the cumulative sum of the edge weights. In this context, it seeks to minimize the total mean travel times. Consequently, the pairwise distances within this MST are generally short. This attribute is clearly observable in the table, as each pair of endpoints is notably close to one another.

Q11

In [7]:
```
triangles = []
while len(triangles)<1000:
    points = np.random.randint(1, len(gcc.vs), size=3)
    try:
        e1 = gcc.get_eid(points[0], points[1])
        e2 = gcc.get_eid(points[1],points[2])
        e3 = gcc.get_eid(points[2],points[0])

        w1 = gcc.es["weight"][e1]
        w2 = gcc.es["weight"][e2]
        w3 = gcc.es["weight"][e3]

        if w1+w2>w3 and w1+w3>w2 and w3+w2>w1:
            triangles.append(1)
        else:
            triangles.append(0)
    except:
        continue

print("The percentage of triangles in the graph satisfy the triangle inequality: ", sum(triangles)/len(triangles))
```

```
The percentage of triangles in the graph satisfy the triangle inequality:  0.93
```

Q12

In [8]:
```python
def all_pairs_shortest_path_length(G):
    # Calculate all pairs shortest path lengths using multiprocessing
    with Pool(24) as pool:
        results = pool.starmap(single_source_shortest_path_length, [(G, n) for n in G])
    return dict(results)

def single_source_shortest_path_length(G, n):
    # Calculate shortest path lengths from a single source n
    return (n, nx.single_source_dijkstra_path_length(G, n))
```

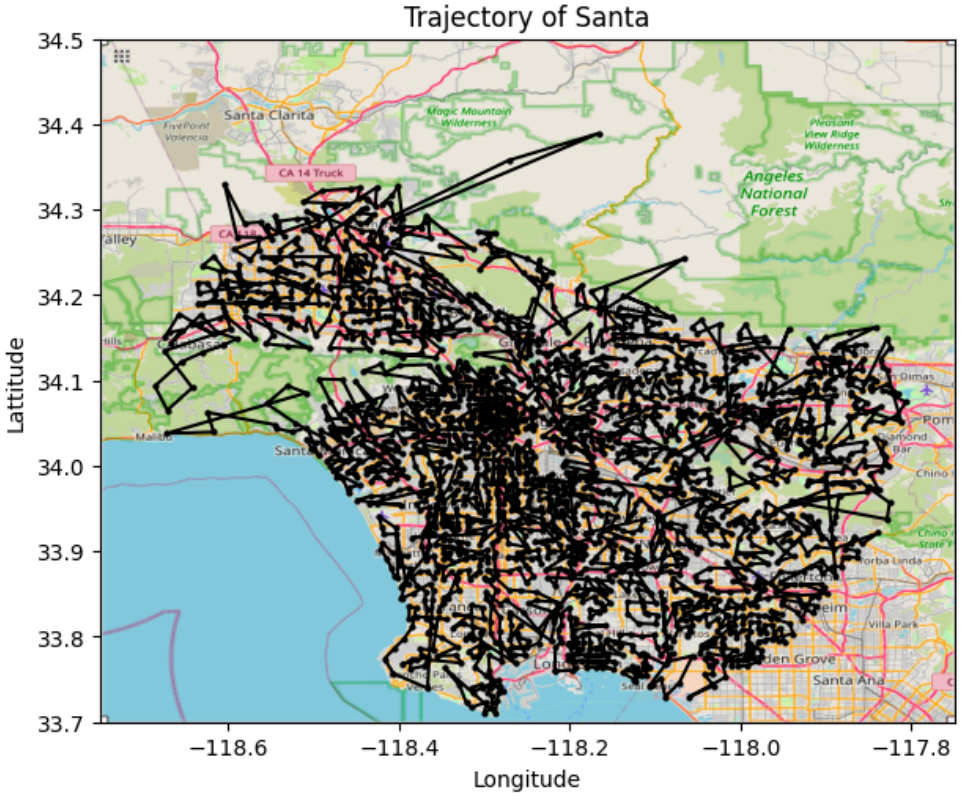
In [ ]:
```python
gcc_nx = gcc.to_networkx()
gcc_node_id_map = {int(gcc_nx.nodes[i]["name"]): i for i in range(len(gcc_nx.nodes))}
all_shortest_paths = all_pairs_shortest_path_length(gcc_nx)

mst_cost = 0
mg = nx.MultiGraph()

for edge in mst.es:
    i, j = edge.tuple
    w = edge["weight"]
    mst_cost += w
    mg.add_edge(int(i), int(j), weight=w)
    mg.add_edge(int(j), int(i), weight=w)

mst_nx = mst.to_networkx()

n_vertex = 100
vertices = list(mg.nodes)[:n_vertex]
costs, cur_paths = [], []
```

In [ ]:
```
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)

    app_cost = 0
    for i in range(len(cur_path)-1):
        s, t = cur_path[i], cur_path[i+1]
        app_cost += all_shortest_paths[s][t]
    costs.append(app_cost)

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

print("Upper bound:{:.3f}".format(min_app_cost / mst_cost))
```

```
  0%|          | 0/100 [00:00<?, ?it/s]

MST cost: 269084.5449999994
Approximate TSP cost: 426010.98499999975
Upper bound: 1.5831863736358427
```

Q13

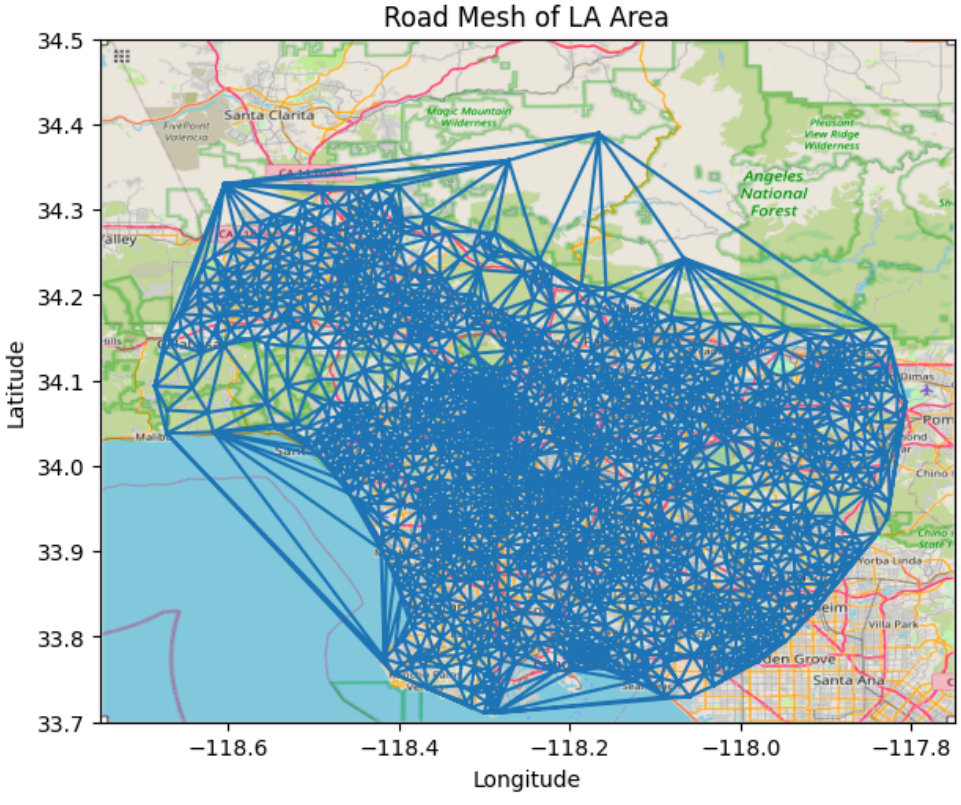
In [ ]:
```
trajectory_coords = [loc_data[gcc_nx.nodes[n]["name"]]["mean_coords"] for n in trajectory]
xs = [x for x, _ in trajectory_coords]
ys = [y for _, y in trajectory_coords]

la_bounds = ((-118.75, -117.75, 33.7, 34.5))
la_map = plt.imread("LA_Map.png")
fig, ax = plt.subplots(figsize = (7,6))
ax.plot(xs, ys, color="black", marker="o", markersize=2)
ax.set_title("Trajectory of Santa")
ax.set_xlim(la_bounds[0], la_bounds[1])
ax.set_ylim(la_bounds[2], la_bounds[3])
ax.imshow(la_map, zorder=0, extent=la_bounds, aspect="equal")
plt.xlabel("Longitude")
plt.ylabel("Lattitude")
plt.show()
```


Trajectory of Santa
Lattitude
Longitude
34.5
34.4
34.3
34.2
34.1
34.0
33.9
33.8
33.7
−118.6
−118.4
−118.2
−118.0
−117.8
Santa Clarita
Magic Mountain Wilderness
Pleasant View Ridge Wilderness
Angeles National Forest
CA 14 Truck
Malibu
Chino
Villa Park
Santa Ana

Q14

In [ ]:

```
from scipy.spatial import Delaunay

node_ids = {int(gcc_nx.nodes[i]["name"]) for i in range(len(gcc_nx.nodes))}
lons = []
lats = []
node_map = {}

for idx in range(1, len(loc_data) + 1):
    if idx in node_ids:
        coords = loc_data[str(idx)]["mean_coords"]
        lons.append(coords[0])
        lats.append(coords[1])
        node_map[len(lons) - 1] = gcc_node_id_map[idx]

# Create a Delaunay triangulation from the latitude and longitude coordinates
triangulation = Delaunay(list(zip(lats, lons)))

fig, ax = plt.subplots(figsize=(7, 6))
plt.triplot(lons, lats, triangulation.simplices)
ax.set_title("Road Mesh of LA Area")
ax.set_xlim(la_bounds[0], la_bounds[1])
ax.set_ylim(la_bounds[2], la_bounds[3])
ax.imshow(la_map, zorder=0, extent=la_bounds, aspect='equal')

plt.xlabel("Longitude")
plt.ylabel("Latitude")
plt.show()
```



The image above shows the road mesh. We can see that the Delaunay triangulation method did a good job outlining the road layouts in LA, especially around the downtown area. However, some parts of the mesh extend over oceans and mountains where there are no actual roads. This happens because the Delaunay triangulation method tries to make sure that each triangle fits inside a specific circle. This characteristic is clear in the second image, which displays a smaller section of the mesh that includes all edges created by this method. In this graph, most of the shapes don't have very sharp corners.

In [ ]:

Q15

We assume the velocity of car is **V_Car**, total number of cars on the road is **Total_car**

1. **V_Car** =

$$\frac{69 \times \sqrt{(Latitude1 - Latitude2)^2 + (Longitude1 - Longitude2)^2}}{\frac{avg(\ travel\ time)}{3600}}$$

2. **Total distance** =

$$V\_Car \times \frac{avg(\ travel\ time)}{3600}$$

3. **Gap** =

$$0.003 + \frac{V\_Car \times 2}{3600}$$

4. **Total_car** =

$$\frac{2 \times Total\ distance}{Gap}$$

5. **Traffic flow (cars/hour)** =

$$\frac{3600 \times Total\_car}{avg(\ travel\ time)}$$

$$Traffic\ flow\ (cars/hour) = \frac{3600 \times V\_car}{5.4 + V\_car}$$

In [1]:
```
!pip install igraph
```

```
Collecting igraph
  Downloading igraph-0.11.5-cp39-abi3-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (3.3 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 3.3/3.3 MB 9.2 MB/s eta 0:00:00
Collecting texttable>=1.6.2 (from igraph)
  Downloading texttable-1.7.0-py2.py3-none-any.whl (10 kB)
Installing collected packages: texttable, igraph
Successfully installed igraph-0.11.5 texttable-1.7.0
```

In [2]:
```
!pip install pycairo
```

```
Collecting pycairo
  Downloading pycairo-1.26.0.tar.gz (346 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 346.9/346.9 kB 1.8 MB/s eta 0:00:00
  Installing build dependencies ... done
  Getting requirements to build wheel ... done
  Installing backend dependencies ... done
  Preparing metadata (pyproject.toml) ... done
Building wheels for collected packages: pycairo
  error: subprocess-exited-with-error

  × Building wheel for pycairo (pyproject.toml) did not run successfully.
  │ exit code: 1
  ╰─> See above for output.

  note: This error originates from a subprocess, and is likely not a problem with pip.
  Building wheel for pycairo (pyproject.toml) ... error
  ERROR: Failed building wheel for pycairo
Failed to build pycairo
ERROR: Could not build wheels for pycairo, which is required to install pyproject.toml-based projects
```

In [3]:
```
!pip install cairocffi
```

```
Collecting cairocffi
  Downloading cairocffi-1.7.0-py3-none-any.whl (75 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 75.4/75.4 kB 761.3 kB/s eta 0:00:00
Requirement already satisfied: cffi>=1.1.0 in /usr/local/lib/python3.10/dist-packages (from cairocffi) (1.16.0)
Requirement already satisfied: pycparser in /usr/local/lib/python3.10/dist-packages (from cffi>=1.1.0->cairocffi) (2.22)
Installing collected packages: cairocffi
Successfully installed cairocffi-1.7.0
```

In [4]:
```python
import json
import random
from collections import defaultdict
from multiprocessing import Pool

import igraph as ig
import matplotlib.pyplot as plt
import networkx as nx
import numpy as np
import pandas as pd
from matplotlib.collections import LineCollection
from scipy.spatial import Delaunay
from tqdm.auto import tqdm
```

In [5]:
```python
from google.colab import drive
drive.mount('/content/gdrive')
import os
os.chdir('/content/gdrive/MyDrive/Colab Notebooks')
```

```
Mounted at /content/gdrive
```

In [6]:
```python
import pandas as pd
import igraph as ig

# Read and filter the data
df = pd.read_csv("data/los_angeles-censustracts-2019-4-All-MonthlyAggregate.csv")
df = df[df["month"] == 12][["sourceid", "dstid", "mean_travel_time"]].reset_index(drop=True)

# Create source_id and dst_id columns
df["source_id"] = df[["sourceid", "dstid"]].min(axis=1)
df["dst_id"] = df[["sourceid", "dstid"]].max(axis=1)

# Group by source_id and dst_id and calculate mean travel time
```

```
graph = df.groupby(by=["source_id", "dst_id"], as_index=False)["mean_travel_time"].mean()

# Write the graph data to a file
graph.to_csv("graph_data.txt", sep=" ", header=False, index=False)

# Read the graph data into an igraph Graph object
g = ig.Graph.Read("graph_data.txt", format="ncol", directed=False)

# Get the giant connected component
gcc = g.components().giant()
```

In [7]:
```
# Load location data
loc_data = {}
with open("data/los_angeles_censustracts.json", "r") as f:
    cur = json.load(f)
    for feature in cur["features"]:
        coordinates = feature['geometry']['coordinates'][0]
        if isinstance(coordinates[0][0], float):
            mean_coords = np.mean(coordinates, axis=0)
        else:
            mean_coords = np.mean(coordinates[0], axis=0)

        loc_data[feature["properties"]["MOVEMENT_ID"]] = {
            "address": feature["properties"]["DISPLAY_NAME"],
            "mean_coords": mean_coords
        }
def all_pairs_shortest_path_length(G):
    with Pool(24) as pool:
        results = pool.starmap(
            single_source_shortest_path_length, [(G, n) for n in G]
        )

    return dict(results)


def single_source_shortest_path_length(G, n):
    return (n, nx.single_source_dijkstra_path_length(G, n))
```

In [8]:
```
gcc_nx = gcc.to_networkx()
gcc_node_ids = {int(gcc_nx.nodes[i]["name"]) for i in range(len(gcc_nx.nodes))}
```

```
all_shortest_paths = all_pairs_shortest_path_length(gcc_nx)
g_delta = ig.Graph()
gcc_node_id_map = {int(gcc_nx.nodes[i]["name"]): i for i in range(len(gcc_nx.nodes))}
longitudes = []
latitudes = []
g_delta_node_id_map = {}
i = 0

for idx in range(1, len(loc_data)+1):
    if idx in gcc_node_ids:
        longitudes.append(loc_data[str(idx)]["mean_coords"][0])
        latitudes.append(loc_data[str(idx)]["mean_coords"][1])
        g_delta_node_id_map[i] = gcc_node_id_map[idx]
        i += 1
delaunay = Delaunay(tuple(zip(latitudes, longitudes)))
g_delta.add_vertices(len(delaunay.points))
g_delta_nx = g_delta.to_networkx()

def compute_single_edge_traffic_flow(s, t):
    x1, x2 = longitudes[s], longitudes[t]
    y1, y2 = latitudes[s], latitudes[t]
    travel_time = all_shortest_paths[g_delta_node_id_map[s]][g_delta_node_id_map[t]]
    v_car = (69 * ((x1 - x2) ** 2 + (y1 - y2) ** 2) ** 0.5) / travel_time * 3600
    return (3600 * v_car) / (5.4 + v_car)
```

Question 16

Answer: The highest number of vehicles that can travel per hour from Malibu to Long Beach is 11074.145, and there are 4 edge-disjoint paths between these locations. When we analyzed the road maps near Malibu and Long Beach, we found that there are 4 nodes exiting Malibu and 6 nodes entering Long Beach. The smaller of these two numbers, which is 4, represents the maximum number of edge-disjoint paths. This indicates that the road network is sufficiently dense to support four independent paths from Malibu.

In [ ]:
```
# Find the closest points
malibu_coords = [-118.56, 34.04]
long_beach_coords = [-118.18, 33.77]

min_dist_to_malibu = np.inf
```

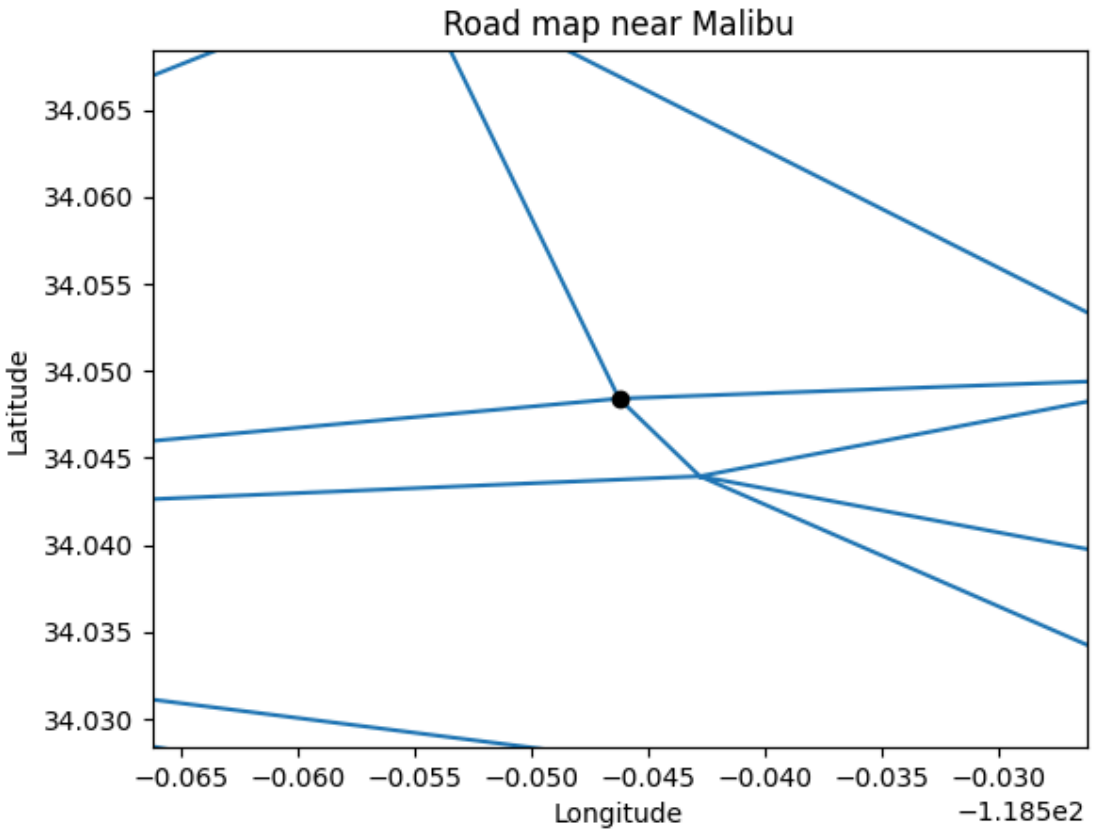
```
min_dist_to_long_beach = np.inf
closest_malibu_node = None
closest_long_beach_node = None
malibu_node_coords = None
long_beach_node_coords = None
node_index = 0

for idx in range(1, len(loc_data) + 1):
    if idx in gcc_node_ids:
        longitude, latitude = loc_data[str(idx)]["mean_coords"]
        distance_to_malibu = np.sqrt((malibu_coords[0] - longitude) ** 2 + (malibu_coords[1] - latitude) ** 2)
        distance_to_long_beach = np.sqrt((long_beach_coords[0] - longitude) ** 2 + (long_beach_coords[1] - latitude) ** 2)

        if distance_to_malibu < min_dist_to_malibu:
            min_dist_to_malibu = distance_to_malibu
            closest_malibu_node = node_index
            malibu_node_coords = [longitude, latitude]
        if distance_to_long_beach < min_dist_to_long_beach:
            min_dist_to_long_beach = distance_to_long_beach
            closest_long_beach_node = node_index
            long_beach_node_coords = [longitude, latitude]
        node_index += 1

max_cars_per_hour = nx.maximum_flow(g_delta_nx, closest_malibu_node, closest_long_beach_node)
print("Maximum number of cars:", round(max_cars_per_hour[0], 3))

def find_edge_disjoint_paths(graph, source, target):
    nx_graph = graph.to_networkx()
    return list(nx.edge_disjoint_paths(nx_graph, source, target))

disjoint_paths = find_edge_disjoint_paths(g_delta, closest_malibu_node, closest_long_beach_node)
print("Number of edge-disjoint paths:", len(disjoint_paths))
```

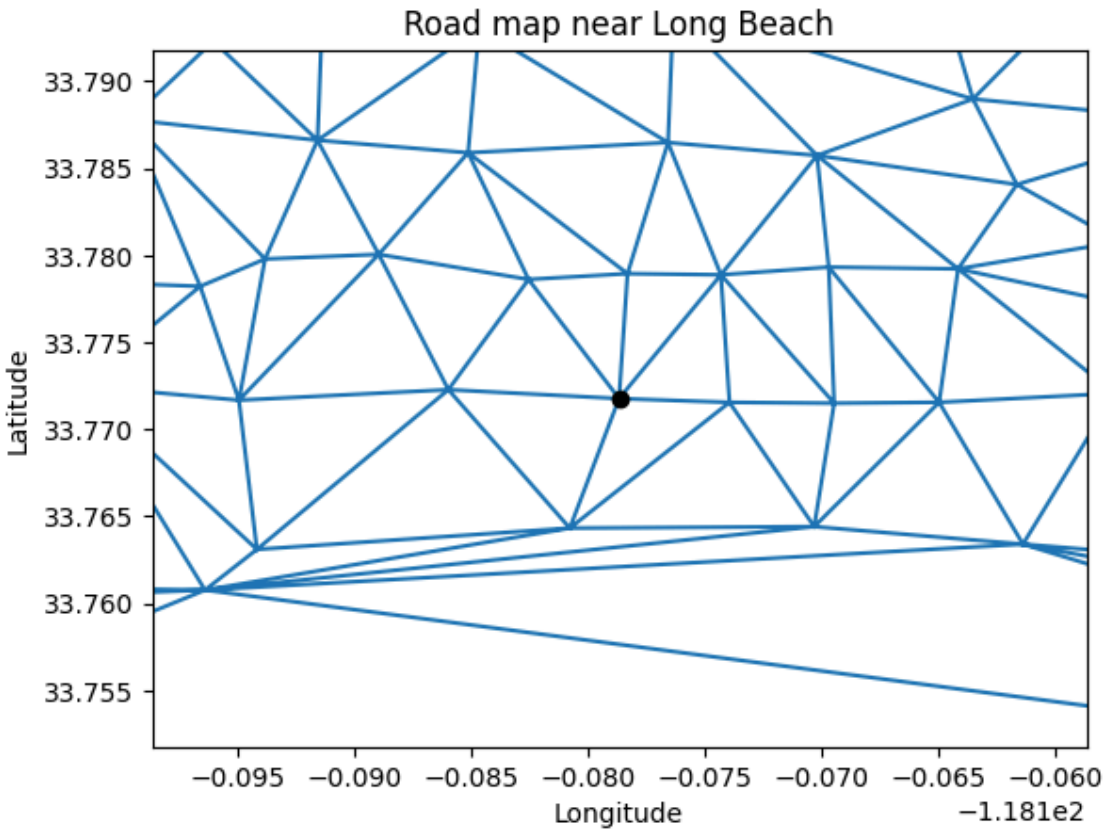
```
Maximum number of cars: 11074.145
Number of edge-disjoint paths: 4
```

In [ ]:

```
view_width = 0.02
lon, lat = malibu_node_coords
plt.triplot(longitudes, latitudes, delaunay.simplices)
plt.xlim(lon - view_width, lon + view_width)
plt.ylim(lat - view_width, lat + view_width)
```

```
plt.xlabel("Longitude")
plt.ylabel("Latitude")
plt.title("Road map near Malibu")
plt.plot(lon, lat, "o", color="black")
plt.show()

lon, lat = long_beach_node_coords
plt.triplot(longitudes, latitudes, delaunay.simplices)
plt.xlim(lon - view_width, lon + view_width)
plt.ylim(lat - view_width, lat + view_width)
plt.xlabel("Longitude")
plt.ylabel("Latitude")
plt.title("Road map near Long Beach")
plt.plot(lon, lat, "o", color="black")
plt.show()
```


Road map near Malibu
Latitude
34.065
34.060
34.055
34.050
34.045
34.040
34.035
34.030
−0.065
−0.060
−0.055
−0.050
−0.045
−0.040
−0.035
−0.030
Longitude
−1.185e2



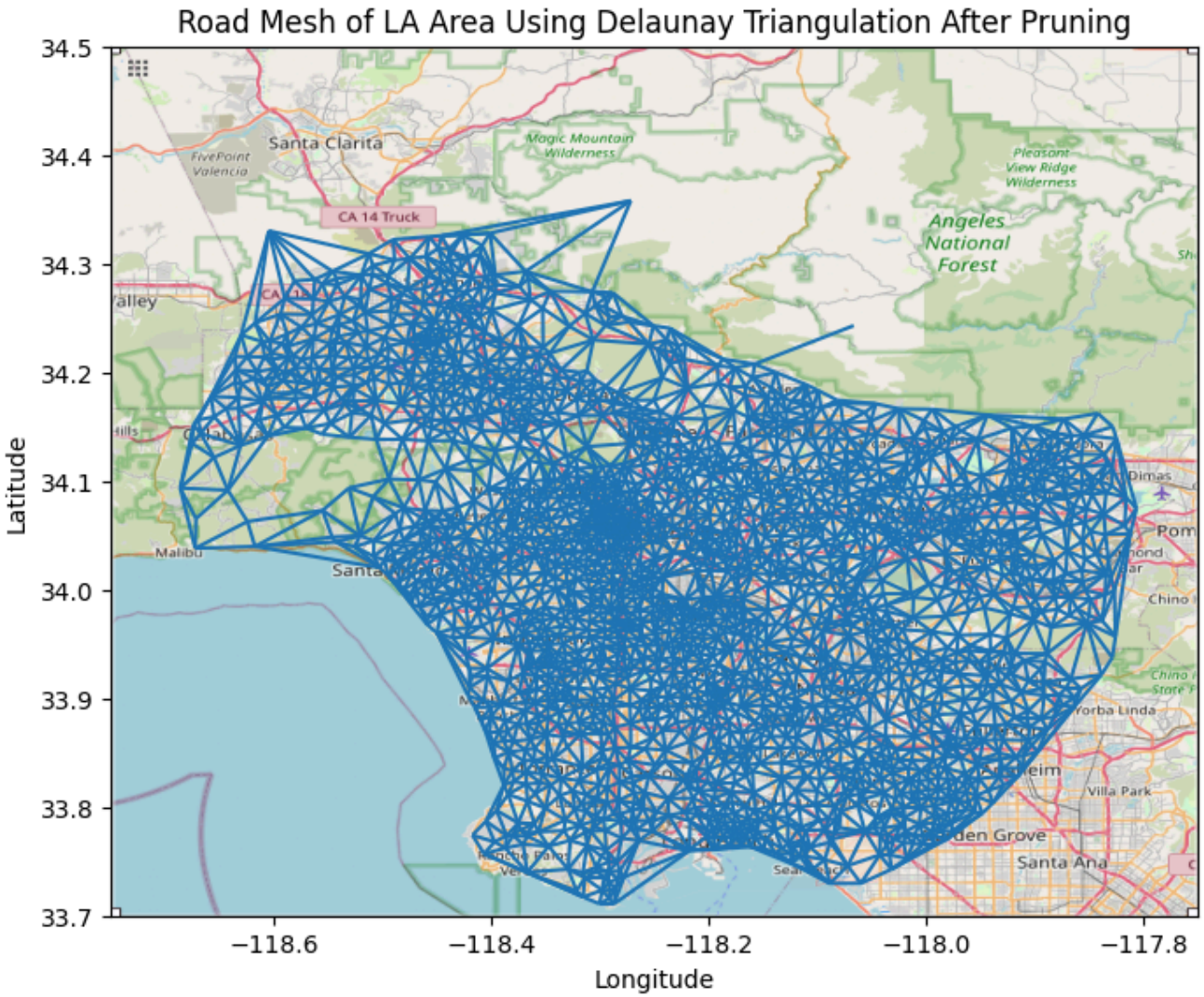
Question 17

Answer: The resulting graph $\tilde{G}_{\Delta}$ is displayed in the following slide. The plot clearly shows that most unrealistic roads crossing the oceans and mountains have been eliminated, proving that the thresholding method was effective.

In [ ]:
```
la_boundary = ((-118.75, -117.75, 33.7, 34.5))
la_background_map = plt.imread("la_map.png")
pruned_g_delta = ig.Graph()
pruned_g_delta.add_vertices(len(delaunay.points))
distance_threshold = 800
coordinates = np.array([[lon, lat] for lon, lat in zip(longitudes, latitudes)])
```

```
kept_edges = []
edges_removed = 0

for edge in g_delta.es():
    src, tgt = edge.source, edge.target
    if all_shortest_paths[g_delta_node_id_map[src]][g_delta_node_id_map[tgt]] < distance_threshold:
        pruned_g_delta.add_edges([(src, tgt)])
        kept_edges.append([src, tgt])
    else:
        edges_removed += 1
print(f"{edges_removed} edges pruned.")

kept_edges = np.array(kept_edges)
line_collection = LineCollection(coordinates[kept_edges])

fig, ax = plt.subplots(figsize=(8, 7))
ax.set_title("Road Mesh of LA Area Using Delaunay Triangulation After Pruning")
plt.xlabel("Longitude")
plt.ylabel("Latitude")
ax.set_xlim(la_boundary[0], la_boundary[1])
ax.set_ylim(la_boundary[2], la_boundary[3])
ax.imshow(la_background_map, zorder=0, extent=la_boundary, aspect="equal")
plt.gca().add_collection(line_collection)
plt.show()
```

```
59 edges pruned.
```

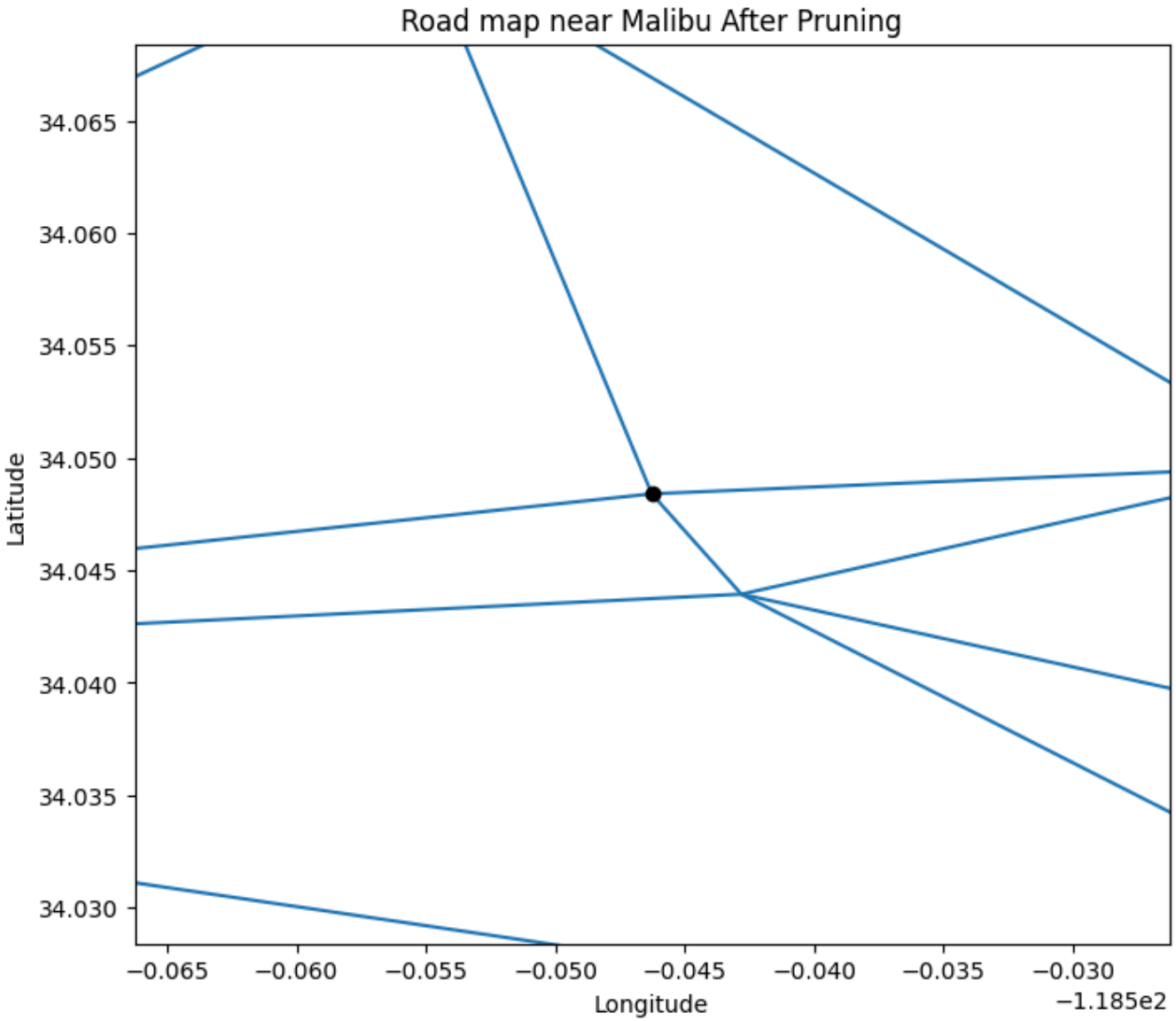


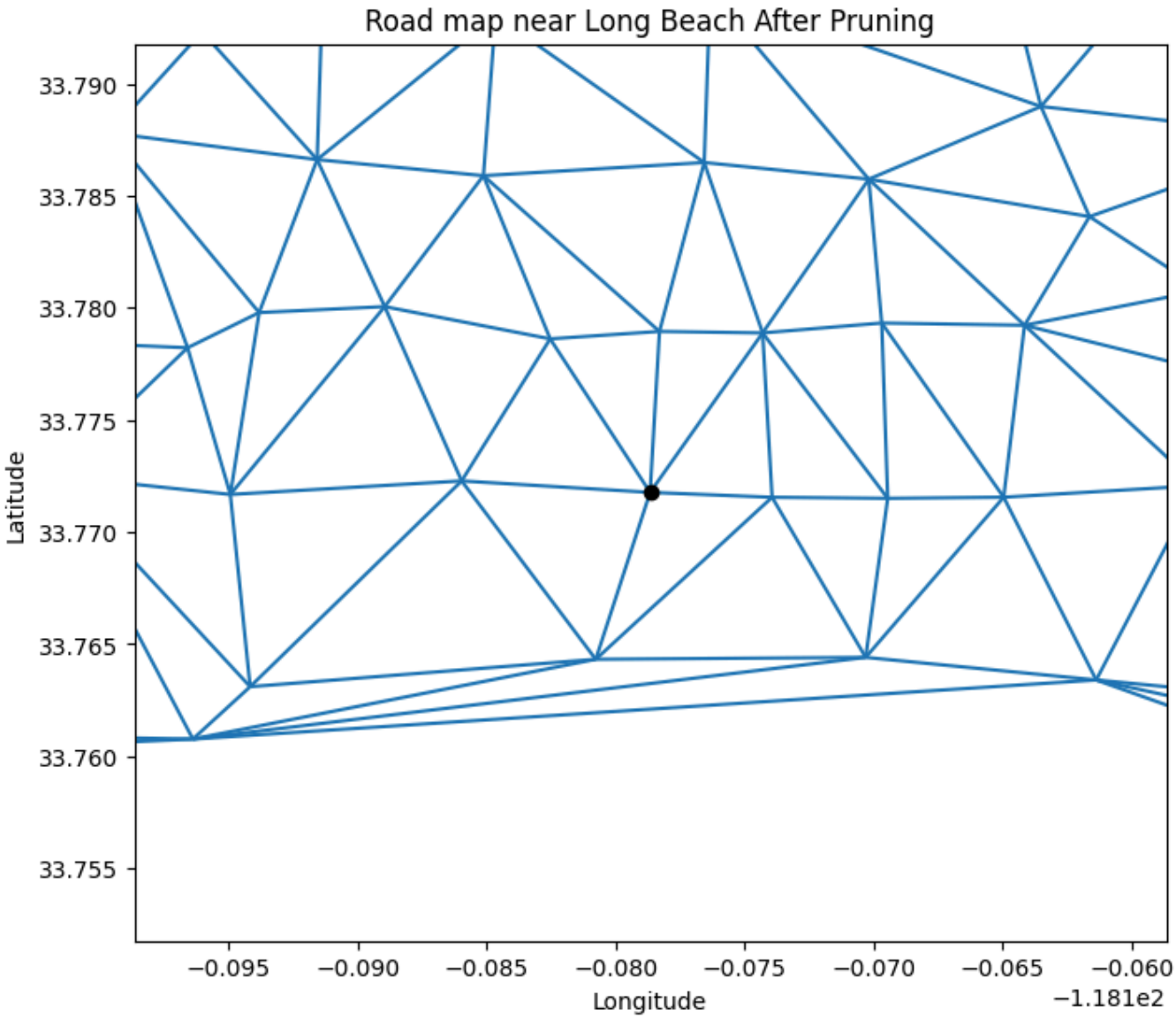
Question 18

Answer: The maximum number of cars per hour traveling from Malibu to Long Beach is still 11074.145, and the number of edge-disjoint paths remains 4, consistent with question 16. Although these numbers have not changed, the average length of the edge-

disjoint paths increased from 16.75 to 32.2 after pruning, due to the removal of several roads.

In [ ]:
```
view_margin = 0.02
malibu_lon, malibu_lat = malibu_node_coords
line_collection = LineCollection(coordinates[kept_edges])
fig, ax = plt.subplots(figsize=(8, 7))
ax.set_title("Road map near Malibu After Pruning")
plt.xlabel("Longitude")
plt.ylabel("Latitude")
ax.set_xlim(la_boundary[0], la_boundary[1])
ax.set_ylim(la_boundary[2], la_boundary[3])
plt.gca().add_collection(line_collection)
plt.axis([malibu_lon - view_margin, malibu_lon + view_margin, malibu_lat - view_margin, malibu_lat + view_margin])
plt.plot(malibu_lon, malibu_lat, "o", color="black")
plt.show()

long_beach_lon, long_beach_lat = long_beach_node_coords
line_collection = LineCollection(coordinates[kept_edges])
fig, ax = plt.subplots(figsize=(8, 7))
ax.set_title("Road map near Long Beach After Pruning")
plt.xlabel("Longitude")
plt.ylabel("Latitude")
ax.set_xlim(la_boundary[0], la_boundary[1])
ax.set_ylim(la_boundary[2], la_boundary[3])
plt.gca().add_collection(line_collection)
plt.axis([long_beach_lon - view_margin, long_beach_lon + view_margin, long_beach_lat - view_margin, long_beach_lat + view_marg
plt.plot(long_beach_lon, long_beach_lat, "o", color="black")
plt.show()
```


Road map near Malibu After Pruning
34.065
34.060
34.055
34.050
34.045
34.040
34.035
34.030
Latitude
−0.065
−0.060
−0.055
−0.050
−0.045
−0.040
−0.035
−0.030
Longitude
−1.185e2


Road map near Long Beach After Pruning
33.790
33.785
33.780
33.775
33.770
33.765
33.760
33.755
Latitude
−0.095
−0.090
−0.085
−0.080
−0.075
−0.070
−0.065
−0.060
Longitude
−1.181e2

Question 19

Answer: For pairs without paths, we assign a large value to represent an infinite distance since they are unreachable. The sources and destinations of the top 20 pairs with the highest extra distance are listed below. The graph with the newly added edges is also shown.

The procedure for identifying the top 20 pairs with the highest extra distance is: (1) calculate the Euclidean distances for all paths, (2) determine the shortest paths for all pairs, (3) compute the extra distances for all pairs, (4) sort these extra distances, and (5) return the top 20 pairs with the highest extra distances. The time complexity is $O(|E|)$ for step (1), $O(|V|(|V|+|E|)\log|V|)$ for step (2), $O(|V|^2)$ for step (3), and $O(|V|^2\log|V|)$ for step (4). Given $|E| \approx 3|V|$, we have $O(|E|) = O(|V|)$, resulting in a simplified time complexity for strategy 1 of $O(|V|^2\log|V|)$.

In [ ]:
```
def calculate_sorted_extra_distances(graph):
    # Add weights to the graph
    nx_graph = graph.to_networkx()
    edge_attributes = {}

    for edge in tqdm(graph.es()):
        source, target = edge.source, edge.target
        edge_weight = calculate_edge_distance(source, target)
        edge_attributes[(source, target)] = {"weight": edge_weight}

    nx.set_edge_attributes(nx_graph, edge_attributes)

    # Calculate shortest distances for all node pairs
    shortest_distances = all_pairs_shortest_path_length(nx_graph)

    # Calculate extra distances for all pairs
    num_nodes = len(nx_graph)
    additional_distances = []
    large_value = 1e6  # Large value if there is no path

    for i in tqdm(range(num_nodes - 1)):
        for j in range(i + 1, num_nodes):
            extra_dist = (
                shortest_distances[i].get(j, large_value)
                - calculate_edge_distance(i, j)
            )
            additional_distances.append((extra_dist, i, j))
```

```
    return sorted(additional_distances, reverse=True)

def calculate_edge_distance(source, target):
    lon1, lon2 = longitudes[source], longitudes[target]
    lat1, lat2 = latitudes[source], latitudes[target]
    return ((lon1 - lon2) ** 2 + (lat1 - lat2) ** 2) ** 0.5

def display_coordinates(edges_array):
    for idx, (_, source, target) in enumerate(edges_array):
        lon1, lon2 = longitudes[source], longitudes[target]
        lat1, lat2 = latitudes[source], latitudes[target]
        print(f"\nEdge {idx + 1}: ({lon1}, {lat1}) - ({lon2}, {lat2})")

def plot_graph_with_added_roads(new_edges):
    new_edges_list = np.array([[source, target] for _, source, target in new_edges])
    old_line_collection = LineCollection(coordinates[kept_edges])
    new_line_collection = LineCollection(coordinates[new_edges_list], colors="red")
    fig, ax = plt.subplots(figsize=(8, 7))
    ax.set_title("Road Mesh of LA Area Using Delaunay Triangulation After Adding New Roads")
    plt.xlabel("Longitude")
    plt.ylabel("Latitude")
    ax.set_xlim(la_boundary[0], la_boundary[1])
    ax.set_ylim(la_boundary[2], la_boundary[3])
    ax.imshow(la_background_map, zorder=0, extent=la_boundary, aspect="equal")
    plt.gca().add_collection(old_line_collection)
    plt.gca().add_collection(new_line_collection)
    plt.show()
```

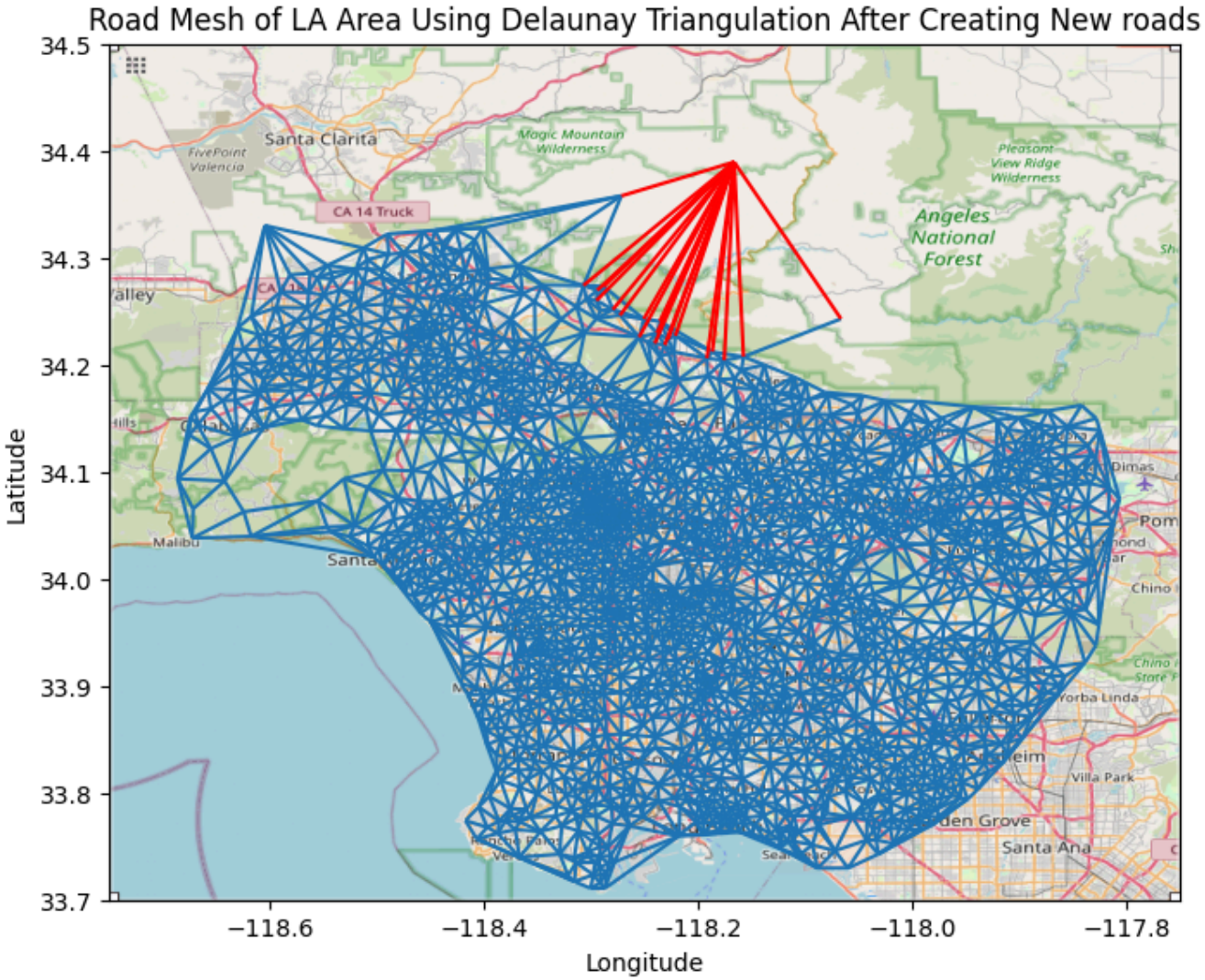
In [ ]:

```
top_k = 20
additional_distances = calculate_sorted_extra_distances(pruned_g_delta)
top_k_additional_distances = additional_distances[:top_k]
print("Top 20 pairs with highest extra distance: (distance, v, s)")
print(top_k_additional_distances)
display_coordinates(top_k_additional_distances)
plot_graph_with_added_roads(top_k_additional_distances)
```

```
  0%|          | 0/7864 [00:00<?, ?it/s]
  0%|          | 0/2648 [00:00<?, ?it/s]
```

```
Top 20 pairs with highest extra distance: (distance, v, s)
[(999999.8899602315, 2418, 2421), (999999.8357921552, 2135, 2418), (999999.831631739, 1684, 2418), (999999.8307363386, 340, 241
8), (999999.8286904375, 509, 2418), (999999.8247041225, 686, 2418), (999999.8246166888, 2136, 2418), (999999.8232761223, 2244,
2418), (999999.8227698414, 2418, 2420), (999999.8215760017, 688, 2418), (999999.8215101181, 507, 2418), (999999.8214202614, 68
9, 2418), (999999.8180394385, 695, 2418), (999999.8178907557, 2242, 2418), (999999.8175523097, 685, 2418), (999999.8170489479,
504, 2418), (999999.814949783, 2137, 2418), (999999.814878125, 508, 2418), (999999.814474637, 1801, 2418), (999999.8139777906,
341, 2418)]

edge 1: (-118.16620295644888, 34.38948489307656) - (-118.27158981797648, 34.35782482248528)

edge 2: (-118.2356042706767, 34.240663947368446) - (-118.16620295644888, 34.38948489307656)

edge 3: (-118.28809007500001, 34.2733324) - (-118.16620295644888, 34.38948489307656)

edge 4: (-118.2506387010309, 34.242785226804116) - (-118.16620295644888, 34.38948489307656)

edge 5: (-118.22109104455448, 34.227206539603955) - (-118.16620295644888, 34.38948489307656)

edge 6: (-118.29409608227856, 34.26960164556964) - (-118.16620295644888, 34.38948489307656)

edge 7: (-118.23833125000003, 34.229619902777785) - (-118.16620295644888, 34.38948489307656)

edge 8: (-118.22793875555557, 34.22389497777778) - (-118.16620295644888, 34.38948489307656)

edge 9: (-118.16620295644888, 34.38948489307656) - (-118.06647650215321, 34.242974950043056)

edge 10: (-118.27952048148147, 34.251665222222215) - (-118.16620295644888, 34.38948489307656)

edge 11: (-118.18650922371963, 34.21215386522912) - (-118.16620295644888, 34.38948489307656)

edge 12: (-118.2728725826087, 34.24626368695651) - (-118.16620295644888, 34.38948489307656)

edge 13: (-118.30693772375686, 34.27414465193369) - (-118.16620295644888, 34.38948489307656)

edge 14: (-118.15749682068964, 34.20758387586207) - (-118.16620295644888, 34.38948489307656)

edge 15: (-118.294821125, 34.26008425) - (-118.16620295644888, 34.38948489307656)

edge 16: (-118.2310955, 34.21842921428571) - (-118.16620295644888, 34.38948489307656)

edge 17: (-118.24030188750005, 34.21991799999999) - (-118.16620295644888, 34.38948489307656)
```

```
edge 18: (-118.19165138650301, 34.206120539877304) - (-118.16620295644888, 34.38948489307656)

edge 19: (-118.17548966791735, 34.20419210506563) - (-118.16620295644888, 34.38948489307656)

edge 20: (-118.25491409756096, 34.225977719512194) - (-118.16620295644888, 34.38948489307656)
```



In [ ]:
```
current_graph = pruned_g_delta.copy()
for _, source, target in tqdm(top_k_additional_distances):
    current_graph.add_edges([(source, target)])
additional_distances_with_new_edges = calculate_sorted_extra_distances(current_graph)
print("The highest extra distance after adding new edges: (distance, v, s)")
print(additional_distances_with_new_edges[0])
```

```
  0%|          | 0/20 [00:00<?, ?it/s]
  0%|          | 0/7884 [00:00<?, ?it/s]
  0%|          | 0/2648 [00:00<?, ?it/s]
The highest extra distance after adding new edges: (distance, v, s)
(0.183110321817979, 2140, 2420)
```

Question 20

Answer: The sources and destinations of the top 20 pairs with the highest weighted extra distance are shown below. The graph with the new edges is also presented.

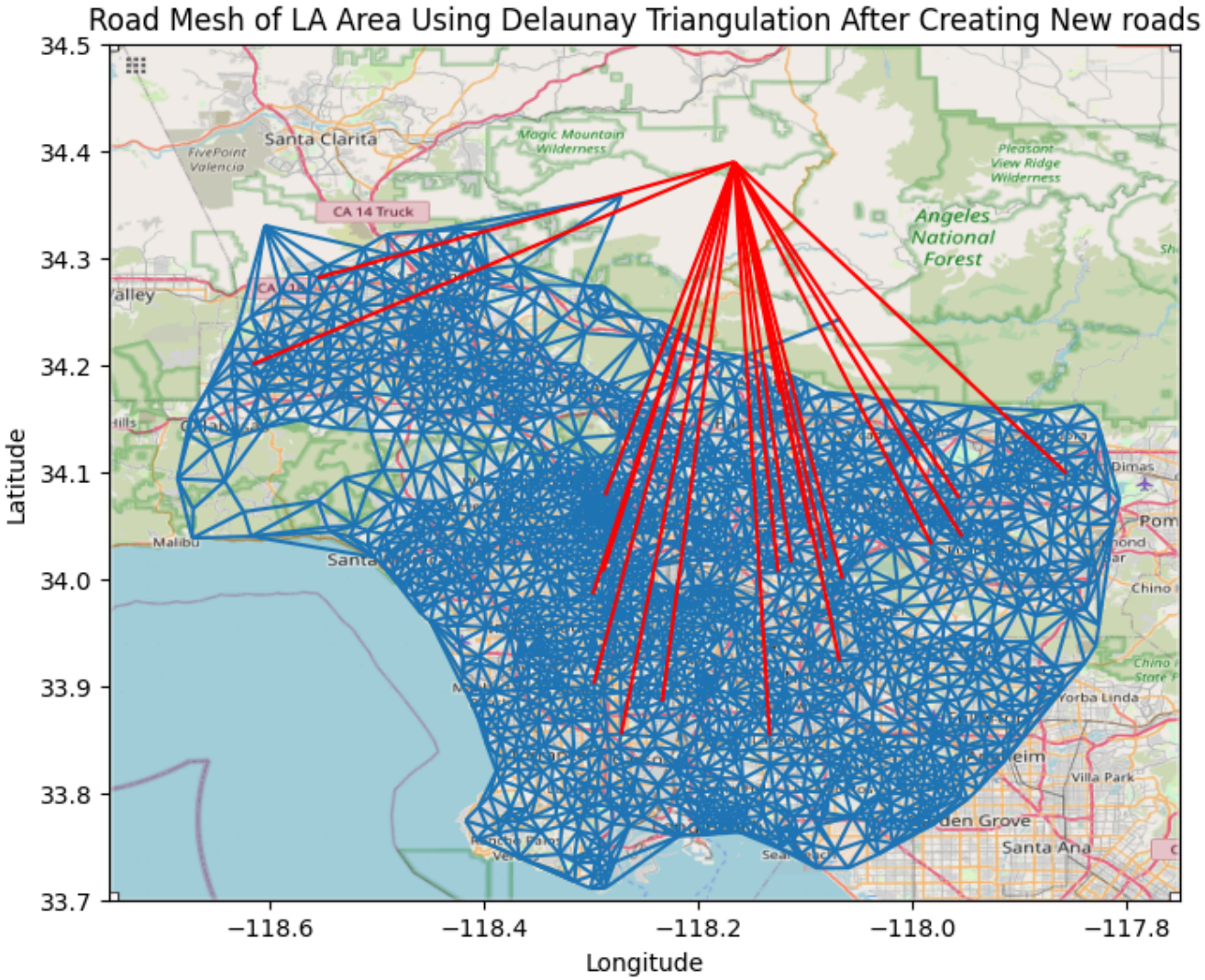
The steps for identifying the top 20 pairs with the highest weighted extra distance are: (1) calculate the Euclidean distances for all paths, (2) find the shortest paths for all pairs, (3) compute the weighted extra distances for all pairs, (4) sort these weighted extra distances, and (5) return the top 20 pairs with the highest weighted extra distances. The time complexity for this strategy is the same as strategy 1, which is $O(|V|^2 \log |V|)$.

In [ ]:
```
weighted_additional_distances = [
    (extra_distance * random.randint(1, 1000), source, target)
    for extra_distance, source, target in tqdm(additional_distances)
]
top_k_weighted_additional_distances = sorted(weighted_additional_distances, reverse=True)[:top_k]
print("Top 20 pairs with highest weighted extra distance: (distance, v, s)")
print(top_k_weighted_additional_distances)
display_coordinates(top_k_weighted_additional_distances)
plot_graph_with_added_roads(top_k_weighted_additional_distances)
```

```
  0%|          | 0/3507276 [00:00<?, ?it/s]
```

```
Top 20 pairs with highest weighted extra distance: (distance, v, s)
[(999999598.3036423, 1409, 2418), (999999596.3569827, 746, 2418), (999999574.6325488, 2153, 2418), (998999780.0179836, 1805, 24
18), (998999666.4809935, 1130, 2418), (998999463.7199732, 524, 2418), (997999623.0029517, 71, 2418), (996999645.6392359, 1727,
2418), (996999514.0727477, 957, 2418), (996999496.7827204, 328, 2418), (995999623.6890547, 1733, 2418), (995999620.6918511, 226
7, 2418), (995999615.9942467, 1739, 2418), (995999600.0470456, 2198, 2418), (995999592.0130638, 552, 2418), (995999493.2980093,
215, 2418), (995999457.2309802, 156, 2418), (994999599.7929591, 2270, 2418), (994999577.0687598, 1443, 2418), (994999525.850154
8, 1760, 2418)]

edge 1: (-118.28854333333331, 34.006871777777775) - (-118.16620295644888, 34.38948489307656)

edge 2: (-118.55507863580257, 34.281302679012335) - (-118.16620295644888, 34.38948489307656)

edge 3: (-117.85588418181818, 34.098555818181815) - (-118.16620295644888, 34.38948489307656)

edge 4: (-118.12368451724139, 34.1734265689655) - (-118.16620295644888, 34.38948489307656)

edge 5: (-118.28672431249998, 34.078145312500006) - (-118.16620295644888, 34.38948489307656)

edge 6: (-118.13321700000002, 33.85368245714286) - (-118.16620295644888, 34.38948489307656)

edge 7: (-117.95596710000001, 34.07564117999999) - (-118.16620295644888, 34.38948489307656)

edge 8: (-118.09415869672132, 34.04143603278687) - (-118.16620295644888, 34.38948489307656)

edge 9: (-118.61552480555555, 34.20065077777778) - (-118.16620295644888, 34.38948489307656)

edge 10: (-118.29732679032259, 33.902083209677414) - (-118.16620295644888, 34.38948489307656)

edge 11: (-118.11309943333332, 34.01541316666666) - (-118.16620295644888, 34.38948489307656)

edge 12: (-118.08074755223883, 34.018364985074626) - (-118.16620295644888, 34.38948489307656)

edge 13: (-118.12515647761194, 34.00612813432835) - (-118.16620295644888, 34.38948489307656)

edge 14: (-117.98208354761907, 34.03262381746032) - (-118.16620295644888, 34.38948489307656)

edge 15: (-117.95340671428572, 34.039469523809515) - (-118.16620295644888, 34.38948489307656)

edge 16: (-118.23340956521739, 33.88520665217392) - (-118.16620295644888, 34.38948489307656)

edge 17: (-118.27160723999998, 33.85482691999999) - (-118.16620295644888, 34.38948489307656)
```

```
edge 18: (-118.06527077777781, 34.00013656481479) - (-118.16620295644888, 34.38948489307656)

edge 19: (-118.29827233333334, 33.98546666666667) - (-118.16620295644888, 34.38948489307656)

edge 20: (-118.06790877499995, 33.92320012500001) - (-118.16620295644888, 34.38948489307656)
```



In [ ]:
```
current_graph = pruned_g_delta.copy()
for _, source, target in tqdm(top_k_weighted_additional_distances):
    current_graph.add_edges([(source, target)])
weighted_additional_distances_with_new_edges = calculate_sorted_extra_distances(current_graph)

print("The highest extra distance after adding new edges: (distance, v, s)")
print(weighted_additional_distances_with_new_edges[0])
```

```
 0%|          | 0/20 [00:00<?, ?it/s]
 0%|          | 0/7884 [00:00<?, ?it/s]
 0%|          | 0/2648 [00:00<?, ?it/s]
The highest extra distance after adding new edges: (distance, v, s)
(0.41432485804064134, 2418, 2421)
```

Question 21

Answer: The sources and destinations of the 20 new edges are listed below. The final graph is also displayed.

The steps for strategy 3 are: (1) calculate the Euclidean distances for all paths, (2) find the shortest paths for all pairs, (3) compute the extra distances for all pairs, (4) sort these extra distances, (5) add the edge with the highest extra distance to the graph, and (6) repeat steps (1) to (5) twenty times to get the final result. Therefore, strategy 3 essentially performs strategy 1 twenty times. Assuming K is the number of new roads, the time complexity for strategy 3 is $O(K|V|^2 \log |V|)$.

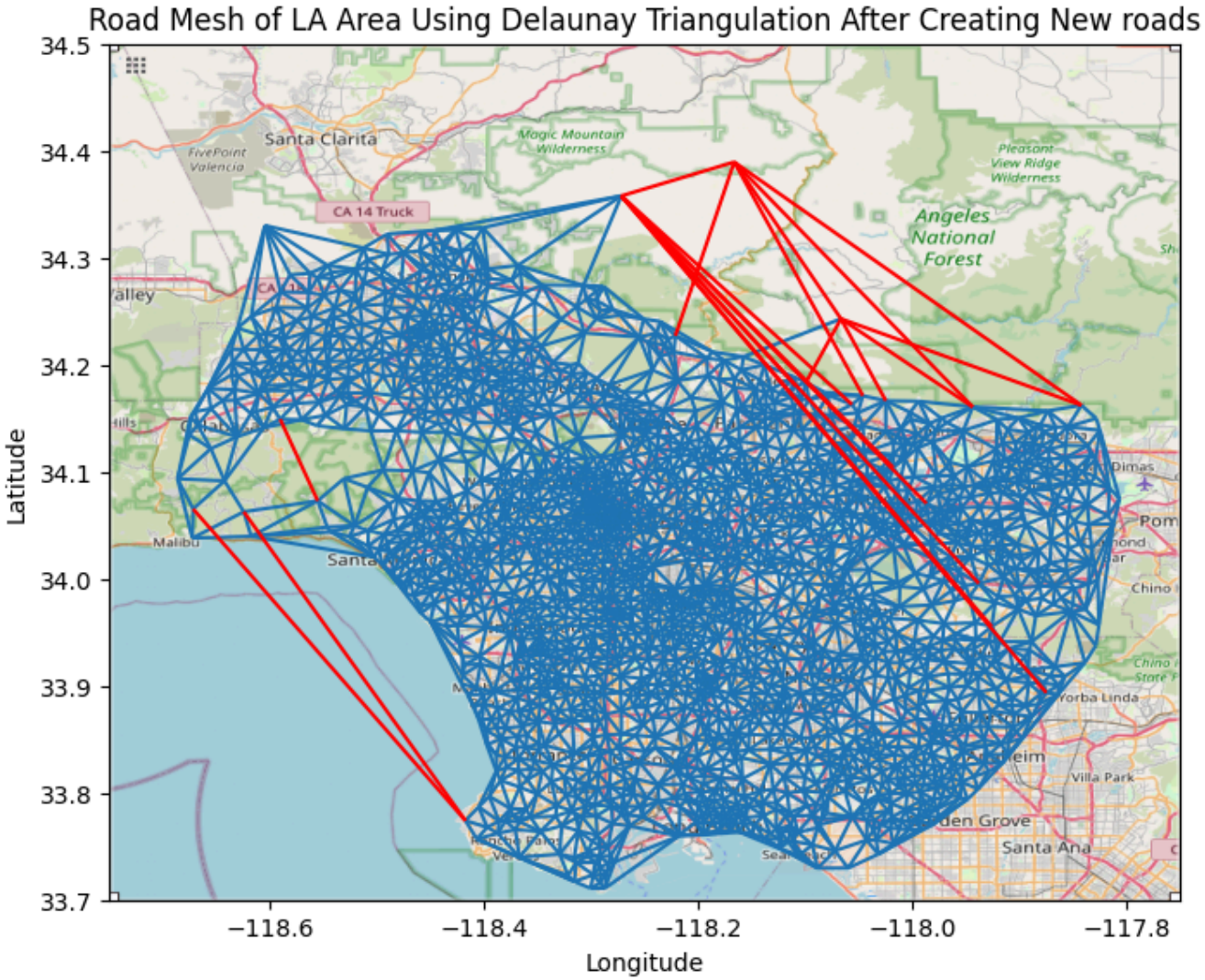
In [ ]:
```
current_graph = pruned_g_delta.copy()
dynamic_top_k_additional_distances = []

for _ in tqdm(range(top_k)):
    maximum_extra_distance = calculate_sorted_extra_distances(current_graph)[0]
    dynamic_top_k_additional_distances.append(maximum_extra_distance)
    current_graph.add_edges([(maximum_extra_distance[1], maximum_extra_distance[2])])

print("Top 20 pairs with highest extra distance (dynamic): (distance, v, s)")
print(dynamic_top_k_additional_distances)
display_coordinates(dynamic_top_k_additional_distances)
plot_graph_with_added_roads(dynamic_top_k_additional_distances)
```

```
 0%|          | 0/20 [00:00<?, ?it/s]
 0%|          | 0/7864 [00:00<?, ?it/s]
```

```
  0%|          | 0/2648 [00:00<?, ?it/s]
  0%|          | 0/7865 [00:00<?, ?it/s]
  0%|          | 0/2648 [00:00<?, ?it/s]
  0%|          | 0/7866 [00:00<?, ?it/s]
  0%|          | 0/2648 [00:00<?, ?it/s]
  0%|          | 0/7867 [00:00<?, ?it/s]
  0%|          | 0/2648 [00:00<?, ?it/s]
  0%|          | 0/7868 [00:00<?, ?it/s]
  0%|          | 0/2648 [00:00<?, ?it/s]
  0%|          | 0/7869 [00:00<?, ?it/s]
  0%|          | 0/2648 [00:00<?, ?it/s]
  0%|          | 0/7870 [00:00<?, ?it/s]
  0%|          | 0/2648 [00:00<?, ?it/s]
  0%|          | 0/7871 [00:00<?, ?it/s]
  0%|          | 0/2648 [00:00<?, ?it/s]
  0%|          | 0/7872 [00:00<?, ?it/s]
  0%|          | 0/2648 [00:00<?, ?it/s]
  0%|          | 0/7873 [00:00<?, ?it/s]
  0%|          | 0/2648 [00:00<?, ?it/s]
  0%|          | 0/7874 [00:00<?, ?it/s]
  0%|          | 0/2648 [00:00<?, ?it/s]
  0%|          | 0/7875 [00:00<?, ?it/s]
  0%|          | 0/2648 [00:00<?, ?it/s]
  0%|          | 0/7876 [00:00<?, ?it/s]
  0%|          | 0/2648 [00:00<?, ?it/s]
  0%|          | 0/7877 [00:00<?, ?it/s]
  0%|          | 0/2648 [00:00<?, ?it/s]
  0%|          | 0/7878 [00:00<?, ?it/s]
  0%|          | 0/2648 [00:00<?, ?it/s]
  0%|          | 0/7879 [00:00<?, ?it/s]
  0%|          | 0/2648 [00:00<?, ?it/s]
  0%|          | 0/7880 [00:00<?, ?it/s]
  0%|          | 0/2648 [00:00<?, ?it/s]
  0%|          | 0/7881 [00:00<?, ?it/s]
  0%|          | 0/2648 [00:00<?, ?it/s]
  0%|          | 0/7882 [00:00<?, ?it/s]
  0%|          | 0/2648 [00:00<?, ?it/s]
  0%|          | 0/7883 [00:00<?, ?it/s]
  0%|          | 0/2648 [00:00<?, ?it/s]
```

```
Top 20 pairs with highest extra distance (dynamic): (distance, v, s)
[(999999.8899602315, 2418, 2421), (0.29749669219110136, 2140, 2418), (0.2898288396346922, 2418, 2420), (0.183110321817979, 214
0, 2420), (0.17756647528999658, 285, 2418), (0.1716543805599868, 285, 2420), (0.14425669584919498, 1956, 2418), (0.142126705213
34925, 22, 2420), (0.1317401160196094, 509, 2418), (0.10041855229667535, 1783, 2416), (0.09881402191574115, 989, 1510), (0.0926
930547033758, 1800, 2420), (0.0880548931972035, 2421, 2598), (0.08631380761131269, 511, 2421), (0.08099737228791043, 2210, 242
1), (0.08083221170940535, 57, 2421), (0.08034495697161337, 32, 2421), (0.07991475015368743, 1957, 2421), (0.07783201112820429,
2247, 2421), (0.07355128068398348, 1783, 2417)]

edge 1: (-118.16620295644888, 34.38948489307656) - (-118.27158981797648, 34.35782482248528)

edge 2: (-117.84197423684212, 34.16224818421053) - (-118.16620295644888, 34.38948489307656)

edge 3: (-118.16620295644888, 34.38948489307656) - (-118.06647650215321, 34.242974950043056)

edge 4: (-117.84197423684212, 34.16224818421053) - (-118.06647650215321, 34.242974950043056)

edge 5: (-117.94347146212125, 34.160677848484866) - (-118.16620295644888, 34.38948489307656)

edge 6: (-117.94347146212125, 34.160677848484866) - (-118.06647650215321, 34.242974950043056)

edge 7: (-118.046042942029, 34.17109230434782) - (-118.16620295644888, 34.38948489307656)

edge 8: (-118.02458801515152, 34.1664155909091) - (-118.06647650215321, 34.242974950043056)

edge 9: (-118.22109104455448, 34.227206539603955) - (-118.16620295644888, 34.38948489307656)

edge 10: (-118.41714115454548, 33.774103109090916) - (-118.67145511923503, 34.06533878290219)

edge 11: (-118.59018528030296, 34.14922015909091) - (-118.55510068503939, 34.07289098425196)

edge 12: (-118.09929410370373, 34.18338863703704) - (-118.06647650215321, 34.242974950043056)

edge 13: (-118.27158981797648, 34.35782482248528) - (-117.87463490909091, 33.89358145454545)

edge 14: (-117.92575677631577, 33.955518921052644) - (-118.27158981797648, 34.35782482248528)

edge 15: (-117.93772208474577, 33.995708423728814) - (-118.27158981797648, 34.35782482248528)

edge 16: (-117.9866033965517, 34.07050068965517) - (-118.27158981797648, 34.35782482248528)

edge 17: (-118.01777852542371, 34.10409581355931) - (-118.27158981797648, 34.35782482248528)
```

```
edge 18: (-118.05616529032255, 34.16293106451614) - (-118.27158981797648, 34.35782482248528)

edge 19: (-118.10868277868856, 34.18016231147541) - (-118.27158981797648, 34.35782482248528)

edge 20: (-118.41714115454548, 33.774103109090916) - (-118.62418590723568, 34.06269522170691)
```



In [ ]:
```
max_extra_distance = calculate_sorted_extra_distances(current_graph)[0]

print("The highest extra distance after adding new edges: (distance, v, s)")
print(max_extra_distance)
```

```
  0%|          | 0/7884 [00:00<?, ?it/s]
  0%|          | 0/2648 [00:00<?, ?it/s]
The highest extra distance after adding new edges: (distance, v, s)
(0.07304768831314037, 2241, 2421)
```

Question 22

Answer: For pairs without paths, we set a large value to represent an infinite travel time and use an arbitrary number 1 for unknown travel speed since they are unreachable. The sources and destinations of the top 20 pairs with the highest extra time are listed below. The resulting graph is also displayed.

The steps for strategy 4 are: (1) calculate the Euclidean distances for all paths, (2) find the shortest paths for all pairs, (3) compute the extra times for all pairs, (4) sort these extra times, and (5) return the top 20 pairs with the highest extra times. The time complexity is $O(|E|)$ for step (1), $O(|V|(|V|+|E|)\log|V|)$ for step (2), $O(|V|^3)$ for step (3), and $O(|V|^2\log|V|)$ for step (4). Given $O(|E|)=O(|V|)$, the simplified time complexity is $O(|V|^3)$.

Extra times for the top 20 pairs are calculated as follows:

Calculate the Euclidean distances for all paths. Determine the shortest paths for all pairs. Compute the extra times for all pairs. Sort these extra times. Return the top 20 pairs with the highest extra times.

In [ ]:
```
def all_pairs_shortest_path_info(G):
    with Pool(24) as pool:
        results = pool.starmap(
            single_source_shortest_path_info, [(G, n) for n in G]
        )

    return dict(results)


def single_source_shortest_path_info(G, n):
    shortest_paths = nx.single_source_dijkstra_path(G, n)
```

```
    results = {}
    for k, shortest_path in shortest_paths.items():
        distance = 0
        travel_time = 0
        eps = 1e-6
        for i, j in zip(shortest_path[:-1], shortest_path[1:]):
            distance += calculate_edge_distance(i, j)
            travel_time += all_shortest_paths[g_delta_node_id_map[i]][g_delta_node_id_map[j]]
        results[k] = {"travel_time": travel_time, "speed": distance / (travel_time + eps)}
    return (n, results)
```

In [ ]:

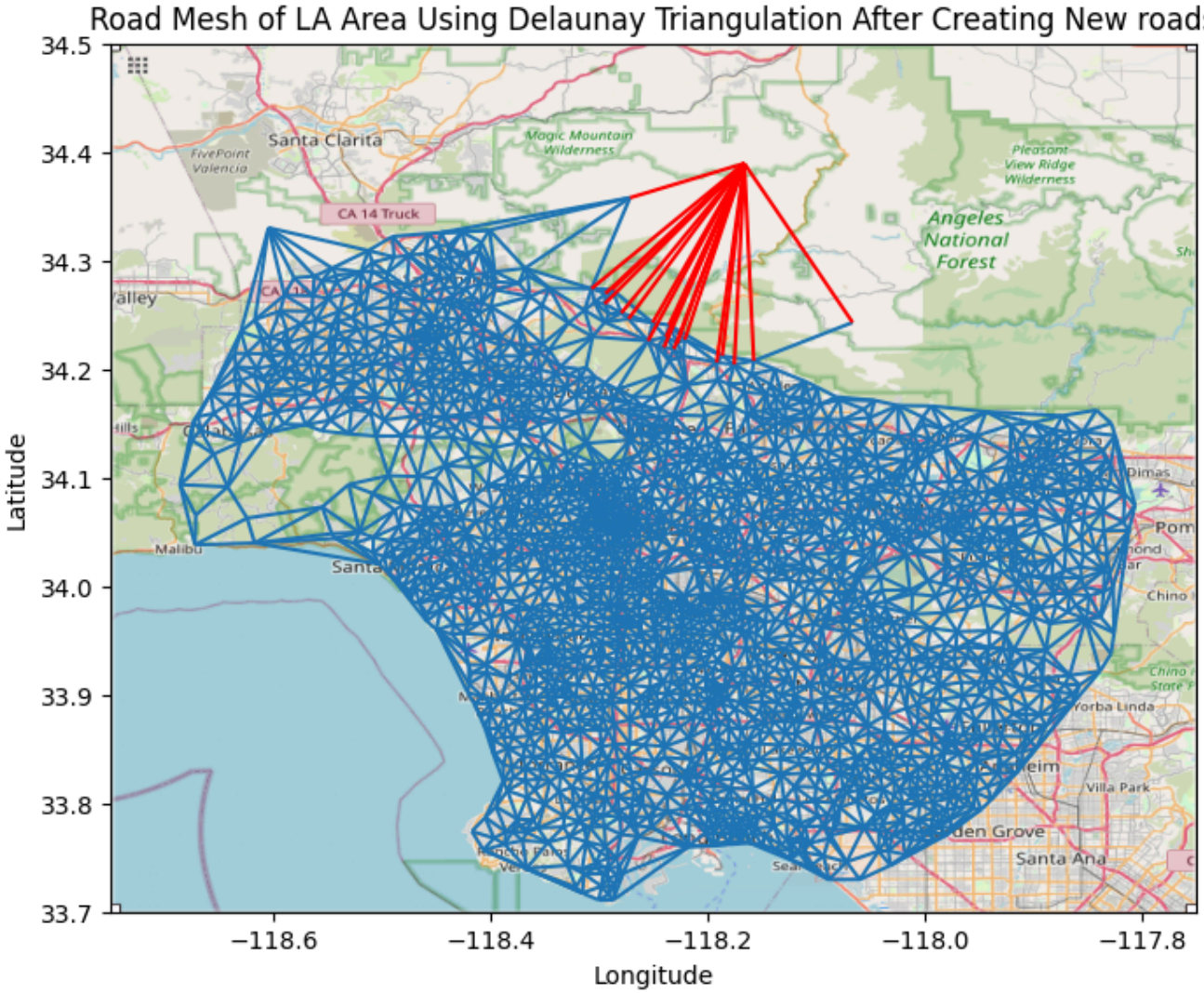
```
def calculate_all_pairs_shortest_path_info(graph):
    with Pool(24) as pool:
        results = pool.starmap(
            calculate_single_source_shortest_path_info, [(graph, node) for node in graph]
        )
    return dict(results)

def calculate_single_source_shortest_path_info(graph, node):
    shortest_paths = nx.single_source_dijkstra_path(graph, node)
    results = {}
    for target, path in shortest_paths.items():
        total_distance = 0
        total_travel_time = 0
        epsilon = 1e-6
        for start, end in zip(path[:-1], path[1:]):
            total_distance += calculate_edge_distance(start, end)
            total_travel_time += all_shortest_paths[g_delta_node_id_map[start]][g_delta_node_id_map[end]]
        results[target] = {"travel_time": total_travel_time, "speed": total_distance / (total_travel_time + epsilon)}
    return (node, results)

def calculate_sorted_extra_times(graph):
    # Add weights to the graph
    nx_graph = graph.to_networkx()
    edge_attributes = {}

    for edge in tqdm(graph.es()):
        source, target = edge.source, edge.target
        edge_weight = calculate_edge_distance(source, target)
        edge_attributes[(source, target)] = {"weight": edge_weight}
```

```
    nx.set_edge_attributes(nx_graph, edge_attributes)

    # Calculate travel time and speed of the shortest path for all node pairs
    all_pairs_info = calculate_all_pairs_shortest_path_info(nx_graph)

    # Calculate extra distances for all pairs
    num_nodes = len(nx_graph)
    additional_times = []
    large_value = 1e6  # Large value if there is no path

    for i in tqdm(range(num_nodes - 1)):
        for j in range(i + 1, num_nodes):
            info = all_pairs_info[i].get(j)
            if info is None:
                # If there is no path, the speed is undefined, so set to an arbitrary number
                extra_time = large_value - calculate_edge_distance(i, j) / 1
            else:
                extra_time = (
                    info["travel_time"]
                    - calculate_edge_distance(i, j) / info["speed"]
                )
            additional_times.append((extra_time, i, j))

    return sorted(additional_times, reverse=True)
```

In [ ]:
```
top_k_additional_times = calculate_sorted_extra_times(pruned_g_delta)[:top_k]
print("Top 20 pairs with highest extra time: (time, v, s)")
print(top_k_additional_times)
display_coordinates(top_k_additional_times)
plot_graph_with_added_roads(top_k_additional_times)
```

```
 0%|          | 0/7864 [00:00<?, ?it/s]
 0%|          | 0/2648 [00:00<?, ?it/s]
```

```
Top 20 pairs with highest extra time: (time, v, s)
[(999999.8899602315, 2418, 2421), (999999.8357921552, 2135, 2418), (999999.831631739, 1684, 2418), (999999.8307363386, 340, 241
8), (999999.8286904375, 509, 2418), (999999.8247041225, 686, 2418), (999999.8246166888, 2136, 2418), (999999.8232761223, 2244,
2418), (999999.8227698414, 2418, 2420), (999999.8215760017, 688, 2418), (999999.8215101181, 507, 2418), (999999.8214202614, 68
9, 2418), (999999.8180394385, 695, 2418), (999999.8178907557, 2242, 2418), (999999.8175523097, 685, 2418), (999999.8170489479,
504, 2418), (999999.814949783, 2137, 2418), (999999.814878125, 508, 2418), (999999.814474637, 1801, 2418), (999999.8139777906,
341, 2418)]
edge 1: (-118.16620295644888, 34.38948489307656) - (-118.27158981797648, 34.35782482248528)
edge 2: (-118.2356042706767, 34.240663947368446) - (-118.16620295644888, 34.38948489307656)
edge 3: (-118.28809007500001, 34.2733324) - (-118.16620295644888, 34.38948489307656)
edge 4: (-118.2506387010309, 34.242785226804116) - (-118.16620295644888, 34.38948489307656)
edge 5: (-118.22109104455448, 34.227206539603955) - (-118.16620295644888, 34.38948489307656)
edge 6: (-118.29409608227856, 34.26960164556964) - (-118.16620295644888, 34.38948489307656)
edge 7: (-118.23833125000003, 34.229619902777785) - (-118.16620295644888, 34.38948489307656)
edge 8: (-118.22793875555557, 34.22389497777778) - (-118.16620295644888, 34.38948489307656)
edge 9: (-118.16620295644888, 34.38948489307656) - (-118.06647650215321, 34.242974950043056)
edge 10: (-118.27952048148147, 34.251665222222215) - (-118.16620295644888, 34.38948489307656)
edge 11: (-118.18650922371963, 34.21215386522912) - (-118.16620295644888, 34.38948489307656)
edge 12: (-118.2728725826087, 34.24626368695651) - (-118.16620295644888, 34.38948489307656)
edge 13: (-118.30693772375686, 34.27414465193369) - (-118.16620295644888, 34.38948489307656)
edge 14: (-118.15749682068964, 34.20758387586207) - (-118.16620295644888, 34.38948489307656)
edge 15: (-118.294821125, 34.26008425) - (-118.16620295644888, 34.38948489307656)
edge 16: (-118.2310955, 34.21842921428571) - (-118.16620295644888, 34.38948489307656)
edge 17: (-118.24030188750005, 34.21991799999999) - (-118.16620295644888, 34.38948489307656)
edge 18: (-118.19165138650301, 34.206120539877304) - (-118.16620295644888, 34.38948489307656)
edge 19: (-118.17548966791735, 34.20419210506563) - (-118.16620295644888, 34.38948489307656)
edge 20: (-118.25491409756096, 34.225977719512194) - (-118.16620295644888, 34.38948489307656)
```



Question 23

Answer: The sources and destinations of the 20 new edges are listed below. The final graph is also shown.

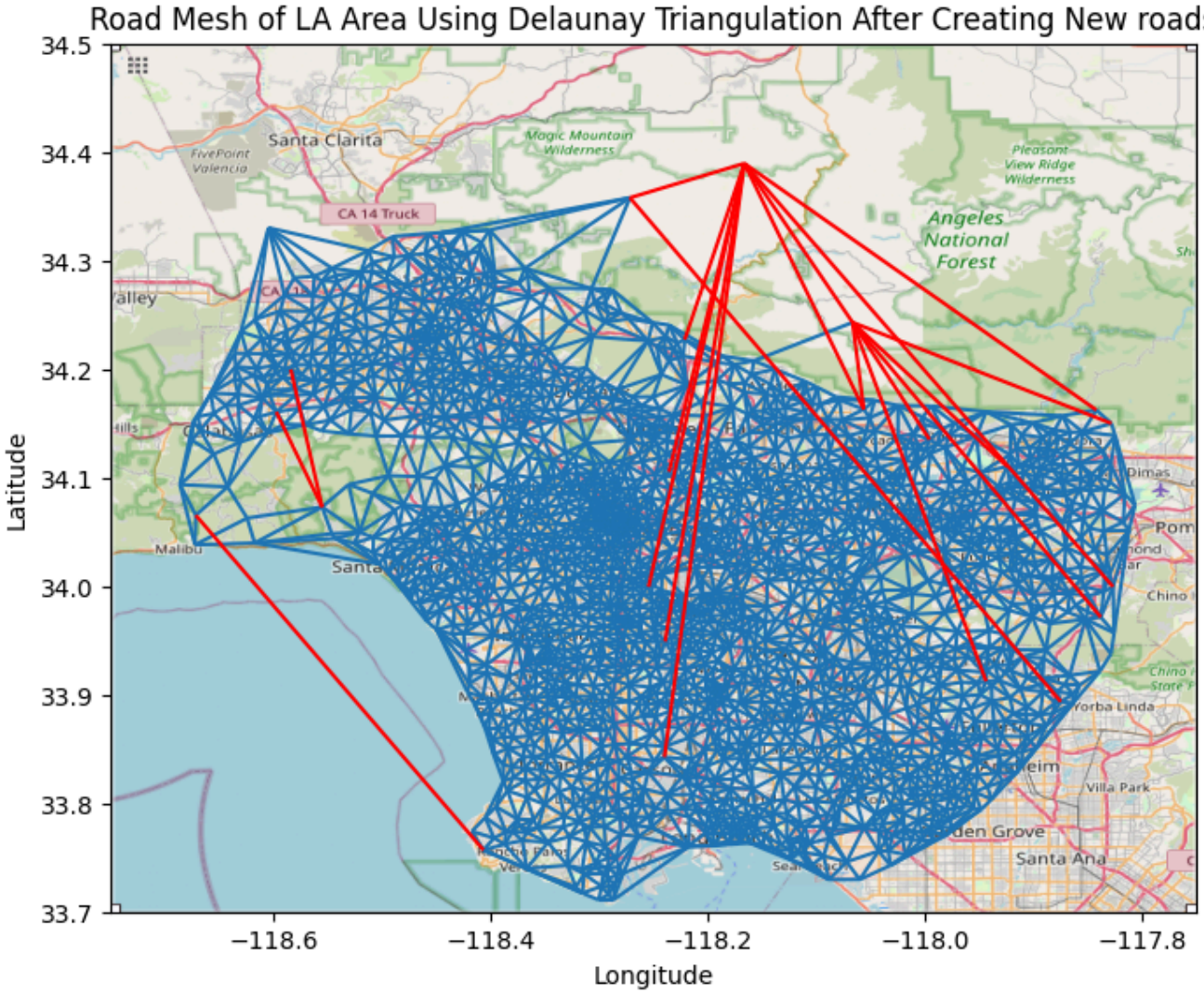
The steps for strategy 5 are: (1) calculate the Euclidean distances for all paths, (2) find the shortest paths for all pairs, (3) compute the extra times for all pairs, (4) sort these extra times, (5) add the edge with the highest extra time to the graph, and (6) repeat steps (1) to (5) twenty times to get the final result. Thus, strategy 5 essentially performs strategy 4 twenty times. Therefore, assuming K is the number of new roads, the time complexity for strategy 5 is $O(K|V|^3)$.

In [ ]:
```
current_graph = pruned_g_delta.copy()
dynamic_top_k_additional_times = []

for _ in tqdm(range(top_k)):
    maximum_extra_time = calculate_sorted_extra_times(current_graph)[0]
    dynamic_top_k_additional_times.append(maximum_extra_time)
    current_graph.add_edges([(maximum_extra_time[1], maximum_extra_time[2])])

print("Top 20 pairs with highest extra time (dynamic): (time, v, s)")
print(dynamic_top_k_additional_times)
display_coordinates(dynamic_top_k_additional_times)
plot_graph_with_added_roads(dynamic_top_k_additional_times)
```

```
  0%|          | 0/20 [00:01<?, ?it/s]
  0%|          | 0/7864 [00:00<?, ?it/s]
  0%|          | 0/2648 [00:00<?, ?it/s]
  0%|          | 0/7865 [00:00<?, ?it/s]
  0%|          | 0/2648 [00:00<?, ?it/s]
  0%|          | 0/7866 [00:00<?, ?it/s]
  0%|          | 0/2648 [00:00<?, ?it/s]
  0%|          | 0/7867 [00:00<?, ?it/s]
  0%|          | 0/2648 [00:00<?, ?it/s]
  0%|          | 0/7868 [00:00<?, ?it/s]
  0%|          | 0/2648 [00:00<?, ?it/s]
  0%|          | 0/7869 [00:00<?, ?it/s]
  0%|          | 0/2648 [00:00<?, ?it/s]
  0%|          | 0/7870 [00:00<?, ?it/s]
  0%|          | 0/2648 [00:00<?, ?it/s]
  0%|          | 0/7871 [00:00<?, ?it/s]
  0%|          | 0/2648 [00:00<?, ?it/s]
  0%|          | 0/7872 [00:00<?, ?it/s]
  0%|          | 0/2648 [00:00<?, ?it/s]
  0%|          | 0/7873 [00:00<?, ?it/s]
  0%|          | 0/2648 [00:00<?, ?it/s]
```

```
  0%|          | 0/7874 [00:00<?, ?it/s]
  0%|          | 0/2648 [00:00<?, ?it/s]
  0%|          | 0/7875 [00:00<?, ?it/s]
  0%|          | 0/2648 [00:00<?, ?it/s]
  0%|          | 0/7876 [00:00<?, ?it/s]
  0%|          | 0/2648 [00:00<?, ?it/s]
  0%|          | 0/7877 [00:00<?, ?it/s]
  0%|          | 0/2648 [00:00<?, ?it/s]
  0%|          | 0/7878 [00:00<?, ?it/s]
  0%|          | 0/2648 [00:00<?, ?it/s]
  0%|          | 0/7879 [00:00<?, ?it/s]
  0%|          | 0/2648 [00:00<?, ?it/s]
  0%|          | 0/7880 [00:00<?, ?it/s]
  0%|          | 0/2648 [00:00<?, ?it/s]
  0%|          | 0/7881 [00:00<?, ?it/s]
  0%|          | 0/2648 [00:00<?, ?it/s]
  0%|          | 0/7882 [00:00<?, ?it/s]
  0%|          | 0/2648 [00:00<?, ?it/s]
  0%|          | 0/7883 [00:00<?, ?it/s]
  0%|          | 0/2648 [00:00<?, ?it/s]
```

```
Top 20 pairs with highest extra time (dynamic): (time, v, s)
[(999999.8899602315, 2418, 2421), (3060.208963219211, 257, 2418), (2849.804708177778, 2140, 2418), (2619.1260304191205, 2418, 2
420), (1875.6892073848894, 258, 2420), (1745.2450141196387, 2420, 2581), (1703.6934420301786, 382, 2420), (1751.540076902231, 5
3, 2420), (1633.1026977317779, 1723, 2420), (1566.0292050457138, 171, 2418), (1522.667706972663, 1494, 2418), (1378.99670871045
7, 1393, 2418), (1354.2106616394876, 1065, 2418), (1272.2120272534448, 1957, 2418), (1269.104402306703, 1957, 2420), (1242.5707
878315516, 509, 2418), (1224.9296046527404, 951, 1510), (1244.6456764330112, 987, 1510), (1211.868387916924, 1860, 2416), (112
5.0380472467205, 2421, 2598)]
edge 1: (-118.16620295644888, 34.38948489307656) - (-118.27158981797648, 34.35782482248528)
edge 2: (-117.82657713380283, 33.99942785211269) - (-118.16620295644888, 34.38948489307656)
edge 3: (-117.84197423684212, 34.16224818421053) - (-118.16620295644888, 34.38948489307656)
edge 4: (-118.16620295644888, 34.38948489307656) - (-118.06647650215321, 34.242974950043056)
edge 5: (-117.83782048888887, 33.97147347407409) - (-118.06647650215321, 34.242974950043056)
edge 6: (-118.06647650215321, 34.242974950043056) - (-117.9435199605263, 33.912415236842094)
edge 7: (-117.82810402090584, 34.1499674041812) - (-118.06647650215321, 34.242974950043056)
edge 8: (-117.9241331458333, 34.110289770833326) - (-118.06647650215321, 34.242974950043056)
edge 9: (-117.99558480459764, 34.1350532183908) - (-118.06647650215321, 34.242974950043056)
edge 10: (-118.23980642168681, 33.84239431325302) - (-118.16620295644888, 34.38948489307656)
edge 11: (-118.23918590000001, 33.94896064999999) - (-118.16620295644888, 34.38948489307656)
edge 12: (-118.25441944444444, 33.999320222222224) - (-118.16620295644888, 34.38948489307656)
edge 13: (-118.23551090322582, 34.105392225806455) - (-118.16620295644888, 34.38948489307656)
edge 14: (-118.05616529032255, 34.16293106451614) - (-118.16620295644888, 34.38948489307656)
edge 15: (-118.05616529032255, 34.16293106451614) - (-118.06647650215321, 34.242974950043056)
edge 16: (-118.22109104455448, 34.227206539603955) - (-118.16620295644888, 34.38948489307656)
edge 17: (-118.58392439285714, 34.19973064285714) - (-118.55510068503939, 34.07289098425196)
edge 18: (-118.59730984444444, 34.16049380000001) - (-118.55510068503939, 34.07289098425196)
edge 19: (-118.40648371022725, 33.75689578693178) - (-118.67145511923503, 34.06533878290219)
edge 20: (-118.27158981797648, 34.35782482248528) - (-117.87463490909091, 33.89358145454545)
```



Question 24

(a) The difference between strategy 1 and strategy 2 is that strategy 2 multiplies a random number between [1, 1000] with the extra distance to get the weighted extra distance. Since the scales of extra distance and weighted extra distance are different, they

are not directly comparable. Therefore, we compare the highest extra distance after adding new edges. Strategy 1 results in a highest extra distance of 0.1831, while strategy 2 results in 0.4143. Both strategies have the same time complexity. Thus, strategy 1 is more effective. The performance of strategy 2 may be negatively impacted by the wide range of random weights, which could alter the distribution of extra distances and affect edge selection.

(b) For strategy 1, after adding new edges from the top 20 pairs with the highest extra distance, the highest remaining extra distance is around 0.1831. In contrast, strategy 3, which iterates 20 times to calculate extra distances and add new edges, results in a highest remaining extra distance of about 0.07305. Hence, strategy 3 is better at minimizing distance but requires higher time complexity. The choice between strategies depends on whether minimizing distance or reducing time complexity is more important.

(c) The 20 new edges selected by strategy 1 are identical to those selected by strategy 4. This is because for pairs with no paths, the same large number represents infinite distance and travel time, with the travel speed arbitrarily set to 1. Thus, extra distances and extra times for unreachable pairs are the same. Although both strategies minimize the highest extra distance or time, strategy 4 is less efficient due to its $O(|V|^3)$ complexity. Therefore, strategy 1 is preferable.

(d) Ignoring time complexity, the dynamic strategy is clearly superior to the static strategy, as it results in a smaller highest extra distance after adding new edges. However, the dynamic strategy with a greedy approach is not optimal since counterexamples exist. For instance, in a network of 5 nodes (A, B, C, D, E) and 4 edges (AB with w=2, BC with w=2, CD with w=1, and DE with w=1), the highest extra distances are (A, E), (B, E), and (A, D) with Extra(A, E) = 2.838, Extra(B, E) = 2.586, and Extra(A, D) = 0.877. If the dynamic strategy adds an edge between A and E, the highest remaining extra distance is 2.586. However, adding an edge between B and E reduces the highest extra distance to 0.877, demonstrating that the dynamic greedy approach is not always optimal.

To find the optimal solution, we should add one edge such that the new maximum extra distance is minimized in each iteration. One possible approach is to evaluate all non-existing edges and select the best one. Given that we need to examine all non-existing edges, which is $O(|E|) = O(|V|)$ in this case, the time complexity becomes $O(K|V|^3 \log |V|)$.