Delhi Technological University
(Formerly Delhi College Of Engineering)
Bawana Road, Delhi – 110042

PROJECT REPORT

Emergency Service Optimization

Algorithm Analysis and Design – CS262

Department of Applied Mathematics

SREYA MAJUMDER
2K19/MC/127
sreyamajumder_2k19mc127@dtu.ac.in

TANISHKA SINGH
2K19/MC/129
tanishkasingh_2k19mc127@dtu.ac.in

Table of Contents

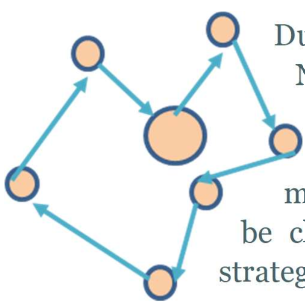
Introduction

The outbreak of coronavirus has put all countries across the globe to in a state where they need to be prepared for any kind of emergencies, one integral part of dealing with emergencies is to provide prompt and adequate help to the common people. This task is carried out by disaster management organizations with the help of police task force and hospitals in countries across the globe.

General public also need to be aware of the protocols they need to follow in times of distress. The integral part of tackling such emergencies or even day to day grievances, is how promptly and efficiently such departments respond to the citizens in need.

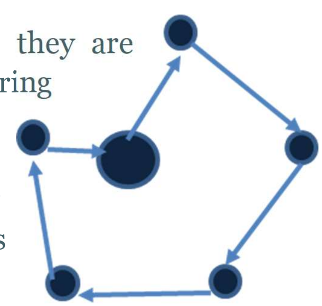
Objective

What Problem the Project Aims to Solve?



During an emergency, all authorizing centers (police stations, NDFC centers) are given the responsibility to extract or attend to a set of citizens nearby. Without proper planning, the situation becomes a mess and every second is important which might cost someone his/her life. Every such auth center should be clear which citizens they need to attend to which requires strategic allocation of an auth center to every center.

Even after every auth center is allocated certain citizens they are responsible to efficiently handle them or extract them during some natural disaster such as an earthquake. Due to limited number of extraction vehicles, it is required that response team from each of such auth center acts on extracting every citizen assigned to them such that the response team leaves the auth center, extracts every citizen and returns.



How the Project Attempts to Solve the Problem?

Problem 1: Efficiently assign each citizen an auth center

Using K-Means clustering algorithm we divide all customers into k groups, each group being head by an authorization center. Then for each cluster containing an auth center and its assigned citizens a connected graph is created.

Problem 2: Finding a path for every auth center so that the response team addresses every citizen assigned to it

Using Heuristics to calculate shortest Hamilton path (TSP) for every authorizing center for a response team to visit every citizen once and return to its center again.

Initial Data Provided for a locality

To demonstrate our project, we take 9 police stations as auth centers and 73 citizens which have requested for help in a certain emergency.

The latitude and longitudes are essential for calculating Euclidian distance which is used for clustering as well as making a connected graph for the clustering output.

latitude	longitude	identity
28.6141925	77.07154118	citizen 1
28.6994533	77.1848256	citizen 2
28.7076568	77.1755473	citizen 3
28.7032676	77.1322497	citizen 4
………	……..	……..
28.6421518	77.11606038	citizen 70
28.6396498	77.09403946	citizen 71
28.6440092	77.05447043	citizen 72
28.6384191	77.07083615	citizen 73

latitude	longitude	identity
28.7259717	77.162658	auth 1
28.6499765	77.2320588	auth 2
28.6926703	77.28354435	auth 3
28.61609	77.243048	auth 4
28.5115704	77.3026233	auth 5
28.6256914	77.10194107	auth 6
28.720002	77.220003	auth 7
28.5735343	77.1863593	auth 8
28.5012304	77.1823908	auth 9

Methodology

Performing Clustering on initial data

The first step of this project is to divide citizens into groups, where each group will be looked after by an auth center. Or basically assign some set of citizens to each auth center based on x, y coordinates.

Clustering is performed by using an algorithm named K-Means clustering.

K-Means: K-Means clustering is a method of vector quantization, originally from signal processing, that aims to partition n observations into k clusters in which each observation belongs to the cluster with the nearest mean (cluster centers or cluster centroid), serving as a prototype of the cluster.

Mathematics behind clustering

Finite set P = $\{x_1, x_2, \ldots \ldots, x_n\}$

$\boldsymbol{x_i} = (x, y)$

P: represents a finite set consisting of coordinates of the citizens which have requested for help.

k: represents the number of groups we want to divide the set P into. (number of auth centers)

Euclidean distance between 2 points: $\left\|x_i - x_j\right\|^2$

k partitions are given by C_i where i ranges from 1 to **k**

$C_1 \, U \, C_2 \, U \cdots C_k = P$ where C_i is a subset of **P**.

K-MEANS AIMS AT MINIMIZING THE FOLLOWING FUNCTION:

$$\min_{C_1 \cup C_2 \cup \cdots \cup C_k = \mathcal{P}} \sum_{i=1}^{k} \sum_{x \in C_i} \left\| x - \frac{1}{|C_i|} \sum_{x_j \in C_i} x_j \right\|^2$$

K-Means algorithm and Python Code

Algorithm

Let X = $\{x_1, x_2, x_3, \ldots \ldots, x_n\}$ be the set of data points and V = $\{v_1, v_2, v_3, \ldots \ldots, v_c\}$ be the set of centers.

1) Randomly select 'c' cluster centers.
2) Calculate the distance between each data point and cluster centers.
3) Assign the data point to the cluster center whose distance from the cluster center is minimum of all the cluster centers.
4) Recalculate the new cluster center by taking mean of every point in that cluster
5) Recalculate the distance between each data point and new obtained cluster centers.
6) If no data point was reassigned then stop, otherwise repeat from step 3)

After performing the clustering of citizens into k groups each group has a mathematically generated centroid which is shifted to the nearest auth center and that auth center is assigned all the citizens in that cluster group.

Advantages Of K-Means Clustering

K-Means clustering is a much faster way for generating such clusters and will be much more efficient when the data is much bigger.

K-Means generates exactly k groups and hence if there are k auth centers all of them will be utilized.

```
def cluster_data(df_cit, df_auth):
    km = KMeans(n_clusters=count_auth, random_state=101)
    km.fit(X=df_cit[["Lat", "Long"]])
    centers = pd.DataFrame(km.cluster_centers_, columns=["Center Lat", "Center Long"])
    centers["Cluster"] = centers.index
    df_cit["Cluster"] = km.labels_
```

Clustering Results and creating graph

Clustering assigns auth center to every citizen and the table above gives an example of a citizen from every cluster. This data is used to create k weighted and connected graphs each representing a cluster.

Graphs of different clusters are not interconnected

G_i: $Graph\ for\ the\ i^{th}$ cluster, $1 \leq i\ \leq k$

V_i: $Vertices\ in\ the\ \ i^{th}$ graph

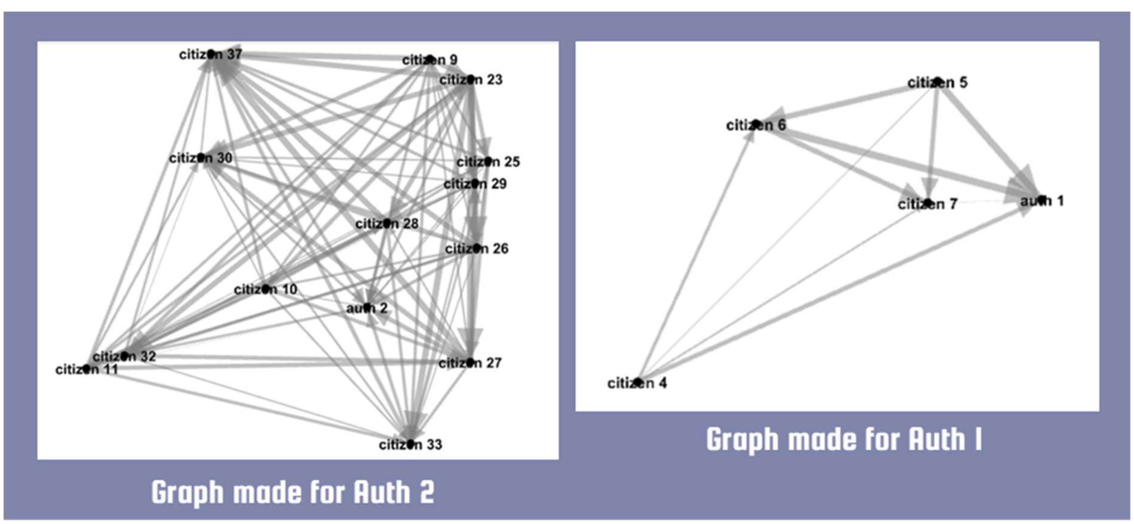
V_i = All points $\in\ \ C_i$ set + {auth center}

Graph of every cluster now represents the auth center and its assigned citizens connected by various paths.

Lat	Long	identity	level	Cluster	Center Lat	Center Long	Auth Center
28.61419	77.07154	citizen 1	1	3	28.6256914	77.10194107	auth 6
28.67979	77.19491	citizen 8	1	5	28.720002	77.220003	auth 7
28.71745	77.15087	citizen 7	1	8	28.7259717	77.162658	auth 1
28.65952	77.20501	citizen 28	1	2	28.6499765	77.2320588	auth 2
28.60014	77.22649	citizen 36	1	6	28.61609	77.243048	auth 4
28.66916	77.31227	citizen 41	1	4	28.6926703	77.28354435	auth 3
28.57416	77.19537	citizen 64	1	1	28.5735343	77.1863593	auth 8
28.56366	77.28905	citizen 53	1	0	28.5115704	77.3026233	auth 5
28.54001	77.11978	citizen 63	1	7	28.5012304	77.1823908	auth 9

Graph made for Auth 2 and Auth 1

Graphs are made for all auth centers, showing 2 examples of the graphs:



Graph made for Auth 2

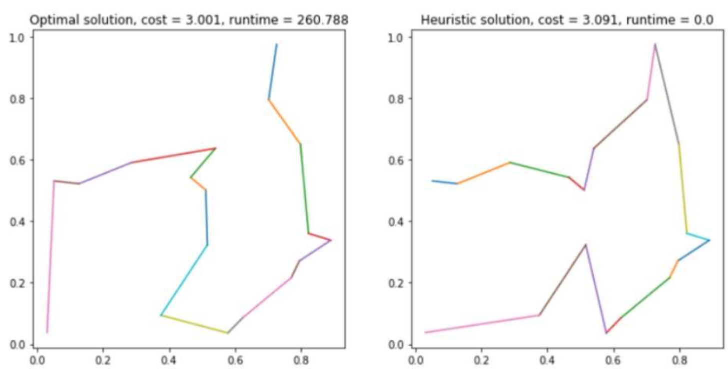
Graph made for Auth I

Applying TSP and finding the shortest Hamilton circuit

In previous step graphs for every auth center and its citizens are generated. The next motive is to find the shortest path a response team can take from the auth center, then extract/help every citizen they are assigned and return to the auth center.

To calculate the shortest Hamilton path, we use heuristic approach of TSP instead of Dynamic Programming method.

Dynamic Programming Method is more accurate, but TSP is an NP-Complete problem, and solving to optimality is highly unscalable particularly between increasing number of nodes in the graph. Hence, we use heuristic methods to compute TSP path between each citizen. Using heuristics and k different cluster graphs we compute a Hamilton circuit with the minimum cost.

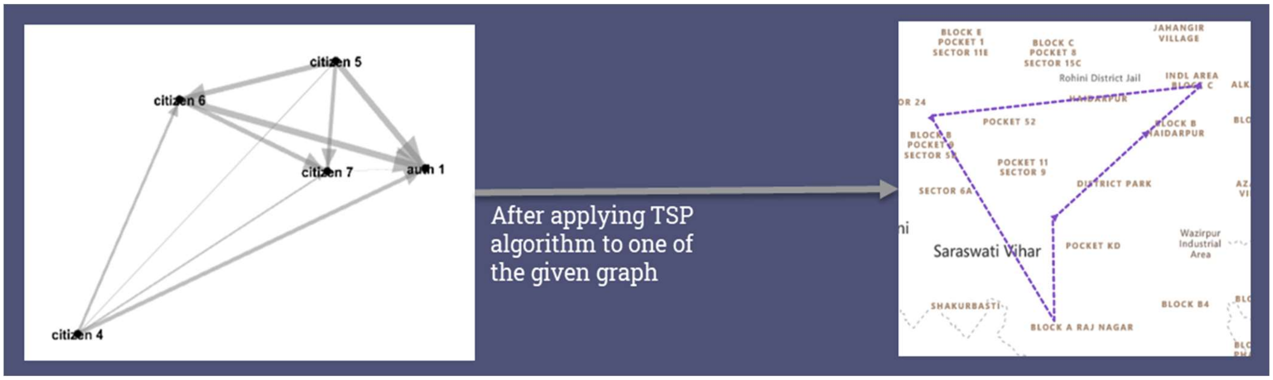


TSP calculated for a random set of 17 cities. Runtime for DP is around 270 seconds whereas Heuristics take time in milliseconds, yet the cost of the graph is almost similar.

Heuristic Algorithm for TSP

1. Select a random edge and make a subtour of it.
2. Select a city not in the subtour, having the shortest distance to any one of the cities in the subtour.
3. Find an edge in the subtour such that the cost of inserting the selected city between the edge's cities will be minimal.
4. Repeat step 2 until no more cities remain.
5. Repeat steps 1 – 4 for all edges in the graph and choose the minimum cost path.

Steps 1 – 5 are repeated for all set of auth center and their corresponding graphs.



TSP path generated and the python code used

This data is then used to create a website where each auth center can access the citizens it needs to attend to and find the path it needs to take to extract every one of them in emergencies and return back.

This project demonstrates how auth centers can efficiently manage the needy citizens as well as extract them efficiently during desperate times. To demonstrate this, a sample data was taken but for practical purposes data for the auth centers and the citizens with request help will be gathered in real time which was beyond the scope of this project.

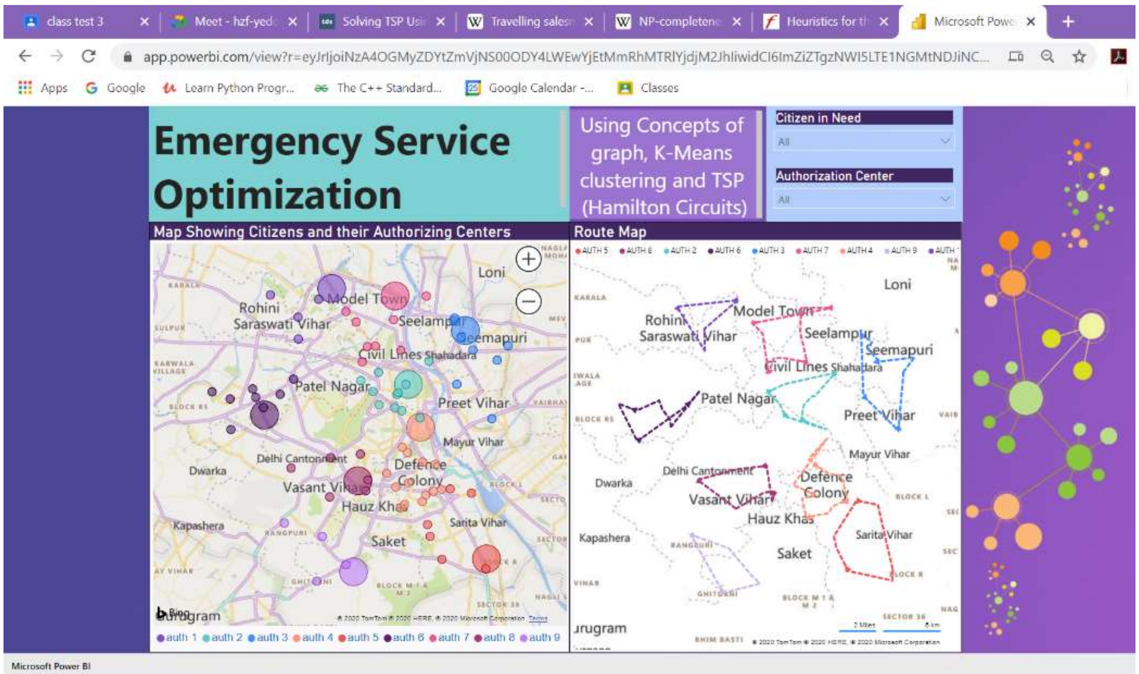
```
def find_best_path(g):
    global smallestdis, best_tsp_path
    all_tsp_paths = {}
    for source in g.nodes:
        path_calc = list(g.nodes)
        path_calc.remove(source)
        path = [source, ]
        dis, path = find_path(g, source, source, path, path_calc)
        all_tsp_paths[dis] = path
        smallestdis = list(all_tsp_paths.keys())[0]
        best_tsp_path = all_tsp_paths[smallestdis]
    for dis in all_tsp_paths.keys():
        if dis < smallestdis:
            best_tsp_path = all_tsp_paths[dis]
    return best_tsp_path


def find_path(g, gsource, source, path, path_calc, totdis=0):
    if len(path_calc) == 1:
        path.append(path_calc[0])
        path.append(gsource)
        totdis = totdis + nx.single_source_dijkstra(g, gsource, path_calc[0])[0]
        return totdis, path
    closest_node = path_calc[0]
    dis = nx.single_source_dijkstra(g, source, closest_node)[0]
  closest_node = path_calc[0]
  dis = nx.single_source_dijkstra(g, source, closest_node)[0]
  for node in path_calc:
      tempdis = nx.single_source_dijkstra(g, source, node)[0]
      if tempdis < dis:
          closest_node = node
          dis = tempdis
  path.append(closest_node)
  path_calc.remove(closest_node)
  totdis = totdis + dis
  totdis, path = find_path(g, gsource, closest_node, path, path_calc, totdis)
  return totdis, path
```

MINIMUM COST HAMILTON CIRCUITS GENERATED AFTER APPLYING TSP TO GRAPH OF EACH CLUSTER

```
['auth 5', 'citizen 59', 'citizen 58', 'citizen 43', 'citizen 45', 'citizen 51', 'citizen 53', 'auth 5']
['citizen 66', 'auth 8', 'citizen 64', 'citizen 54', 'citizen 31', 'citizen 61', 'citizen 68', 'citizen 60', 'ci
['auth 2', 'citizen 11', 'citizen 10', 'citizen 25', 'citizen 29', 'citizen 32', 'citizen 30', 'citizen 37', 'ci
['auth 6', 'citizen 69', 'citizen 71', 'citizen 14', 'citizen 73', 'citizen 72', 'citizen 1', 'citizen 70', 'cit
['citizen 41', 'citizen 20', 'citizen 21', 'citizen 42', 'citizen 19', 'auth 3', 'citizen 24', 'citizen 38', 'ci
['auth 7', 'citizen 18', 'citizen 12', 'citizen 13', 'citizen 8', 'citizen 17', 'citizen 16', 'citizen 2', 'citi
['auth 4', 'citizen 36', 'citizen 34', 'citizen 52', 'citizen 15', 'citizen 49', 'citizen 57', 'citizen 46', 'ci
['citizen 65', 'citizen 44', 'auth 9', 'citizen 62', 'citizen 63', 'citizen 65']
['citizen 5', 'citizen 4', 'citizen 7', 'auth 1', 'citizen 6', 'citizen 5']
```

Results

THIS INTERACTIVE WEBPAGE SHOWS THE CLUSTERING RESULTS AS WELL AS PATHS FOR EVERY SET OF CLUSTERS



Link: https://bit.ly/2JeRz4D

GitHub Repository Link:

https://github.com/tanishka2001/Emergency-Service-Optimization

Conclusion

Future Scope

With a team of coders developing an app which uses this project to make a portal for citizens where they can request extraction by any type of agency or by hospitals nearest to them and allow these agencies/hospitals to respond to needy citizens and help them without wasting time on planning the routes to take or which citizens to visit as that will be done by the program.

Conclusion

The world revolves around technology and the covid epidemic taught the world that it needs to be prepared for any such disasters in future. The concept demonstrated in this project can be used by various agencies or hospitals for providing maximum and quick help to needy citizens in desperate times.

References

Resources used to complete the project

- https://towardsdatascience.com/solving-tsp-using-dynamic-programming-2c77da86610d
- https://en.wikipedia.org/wiki/Travelling_salesman_problem
- http://160592857366.free.fr/joe/ebooks/ShareData/Heuristics%20for%20the%20Traveling%20Salesman%20Problem%20By%20Christian%20Nillson.pdf
- https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html
- https://networkx.org/
- https://www.javatpoint.com/discrete-mathematics-travelling-salesman-problem#:~:text=Suppose%20a%20salesman%20wants%20to,of%20cities%20allotted%20to%20him.&text=If%20we%20represent%20the%20cities,i(weight)%20is%20associated.

Appendix

Source Code:

1. Auth and citizen is the input dataset which is imported into the python file.
2. Python source code generates output excel.
3. Output excel is processed into PowerBi source file to make website.

```
import pandas as pd
import numpy as np
from sklearn.cluster import KMeans
import seaborn as sns
from networkx import *
import matplotlib.pyplot as plt


sns.set()


def distance(s_lat, s_lng, e_lat, e_lng):
    # approximate radius of earth in km
    r = 6373.0
    s_lat = s_lat * np.pi / 180.0
    s_lng = np.deg2rad(s_lng)
    e_lat = np.deg2rad(e_lat)
    e_lng = np.deg2rad(e_lng)
    d = np.sin((e_lat - s_lat) / 2) ** 2 + np.cos(s_lat) * np.cos(e_lat) * np.sin((e_lng - s_lng) / 2) ** 2
    return 2 * r * np.arcsin(np.sqrt(d))


def create_graph(df, auth, auth_lat, auth_long):
    g = nx.Graph()
    df_copy = df[(df.Centername == auth)].copy().reset_index()
    source = df_copy['identity'].tolist()
    authlist = [auth, ]
    source.extend(authlist)

    g.add_nodes_from(source)
    for index, c in df_copy.iterrows():
        g.add_edge(auth, c['identity'], weight=distance(auth_lat, auth_long, c['Lat'], c['Long']))

    for cindex, c in df_copy.iterrows():
        for cindex1, c1 in df_copy.iterrows():
            if c['identity'] == c1['identity']:
                continue
            g.add_edge(c['identity'], c1['identity'], weight=distance(c['Lat'], c['Long'], c1['Lat'], c1['Long']))
    nx.draw(g)
    plt.savefig("{}.png".format(auth))
    return g


def graph_list(centers, df_final):
    dict_of_graphs = {}
    for index, a in centers.iterrows():
        g = create_graph(df_final, a['Centername'], a['Center Lat'], a['Center Long'])
        dict_of_graphs[a['Centername']] = g
    return dict_of_graphs
```

```
def find_tsp(centers, df_tot, df_final):
    dict_g = graph_list(centers, df_final)
    df_path = pd.DataFrame(columns=['lat', 'long', 'auth_name'])
    for source, g in dict_g.items():
        path = find_best_path(g)
        print(path)
        for index in range(len(path)):
            df = {'lat': df_tot[(df_tot.identity == path[index])].iloc[0].Lat,
                  'long': df_tot[(df_tot.identity == path[index])].iloc[0].Long,
                  'auth_name': source}
            temp_df = pd.DataFrame([df])
            df_path = pd.concat([df_path, temp_df], ignore_index=True)

    return df_path


def find_best_path(g):
    global smallestdis, best_tsp_path
    all_tsp_paths = {}
    for source in g.nodes:
        path_calc = list(g.nodes)
        path_calc.remove(source)
        path = [source, ]
        dis, path = find_path(g, source, source, path, path_calc)
        all_tsp_paths[dis] = path
        smallestdis = list(all_tsp_paths.keys())[0]
        best_tsp_path = all_tsp_paths[smallestdis]
    for dis in all_tsp_paths.keys():
        if dis < smallestdis:
            best_tsp_path = all_tsp_paths[dis]
    return best_tsp_path


def find_path(g, gsource, source, path, path_calc, totdis=0):
    if len(path_calc) == 1:
        path.append(path_calc[0])
        path.append(gsource)
        totdis = totdis + nx.single_source_dijkstra(g, gsource, path_calc[0])[0]
        return totdis, path
    closest_node = path_calc[0]
    dis = nx.single_source_dijkstra(g, source, closest_node)[0]
    for node in path_calc:
        tempdis = nx.single_source_dijkstra(g, source, node)[0]
        if tempdis < dis:
            closest_node = node
            dis = tempdis
    path.append(closest_node)
    path_calc.remove(closest_node)
    totdis = totdis + dis
    totdis, path = find_path(g, gsource, closest_node, path, path_calc, totdis)
    return totdis, path


def cluster_data(df_cit, df_auth):
    km = KMeans(n_clusters=count_auth, random_state=101)
    km.fit(X=df_cit[["Lat", "Long"]])
    centers = pd.DataFrame(km.cluster_centers_, columns=["Center Lat", "Center Long"])
    centers["Cluster"] = centers.index
```

```
    df_cit["Cluster"] = km.labels_

    for index, c in centers.iterrows():
        clong = c['Center Long']
        clat = c['Center Lat']  # when you have space between the name
        ds = []
        for ind, auth in df_auth.iterrows():
            authlong = auth.Long
            authlat = auth.Lat
            distance_center = distance(clong, clat, authlong, authlat)
            ds.append(distance_center)
        idx = np.argmin(np.array(ds))

        centers.at[index, "Center Lat"] = df_auth.at[idx, "Lat"]
        centers.at[index, "Center Long"] = df_auth.at[idx, "Long"]
        centers.at[index, "Centername"] = df_auth.at[idx, "identity"]

    df = pd.merge(df_cit, centers)
    return df, centers


def get_dataframes(file_name):
    global count_auth
    df = pd.read_excel(file_name)
    for index, c in df.iterrows():
        if 'citizen' in c['identity']:
            df.at[index, "level"] = '1'
        elif 'auth' in c['identity']:
            df.at[index, "level"] = '2'
            count_auth = count_auth + 1
    df_return = df.copy()[['latitude', 'longitude', 'identity', 'level']]
    df_return = df_return.rename(columns={"longitude": "Long", 'latitude': "Lat", 'identity': "identity"})
    return df_return


file_name_cit = "D:\\College\\Sem-IV\\ADA\\Project\\citizen.xlsx"
file_name_auth = " D:\\College\\Sem-IV\\ADA\\Project\\auth.xlsx"
df_cit = get_dataframes(file_name_cit)
count_auth = 0
df_auth = get_dataframes(file_name_auth)
print(count_auth)
df_tot = pd.concat([df_cit, df_auth], ignore_index=True)
df_final, centers = cluster_data(df_cit, df_auth)
centers.drop_duplicates(subset="Centername", keep='first', inplace=True)
df_final.groupby('Centername')
df_final.to_excel("D:\\College\\Sem-IV\\ADA\\Project\\clustured dataset.xlsx")
G = nx.Graph()
df_path = find_tsp(centers, df_tot, df_final)
df_path.to_excel("D:\\College\\Sem-IV\\ADA\\Project\\output_dftest.xlsx")
```