UNIVERSITÀ DEGLI STUDI DELL' AQUILA

DIPARTIMENTO DI INGEGNERIA E SCIENZE DELL' INFORMAZIONE E MATEMATICA

Corso di Laurea Magistrale in Informatica

MODELS AND ALGORITHMS FOR LOCATING CHARGING STATIONS IN URBAN AREAS

Relatore:
Prof. **Rossi Fabrizio**
Correlatore:
Dott. **De Lazzari Alberto**

CANDIDATA:
Di Egidio Laura
Matr:273860

Anno Accademico 2022/2023

Abstract

This thesis aims to address the optimization of charging station placement in urban areas, focusing on the construction of optimal models. The research begins by exploring the history of automobiles and the compelling need to transition to electric vehicles. The growing demand for sustainable transportation alternatives highlights the significance of establishing efficient charging infrastructure.

To achieve this goal, the study delves into various location models for charging stations, considering population density, traffic patterns, and existing charging infrastructure. The models are designed to maximize the accessibility and convenience of charging stations for electric vehicle owners while minimizing costs and environmental impacts.

As a case study, the city of Genoa is examined. Genoa's unique urban landscape and transportation patterns make it an interesting and challenging setting to develop effective charging station placement strategies. Through the utilization of data analysis techniques and optimization algorithms, the research investigates optimal solutions for deploying charging stations in Genoa, considering both current and future electric vehicle adoption scenarios.

The findings of this study contribute to the advancement of sustainable transportation systems by providing insights into the optimal placement of charging stations in urban areas. The proposed models and case study outcomes offer valuable guidance for policymakers, urban planners, and stakeholders involved in the development of electric vehicle infrastructure, fostering the transition towards a more sustainable and eco-friendly transportation system.

Contents

1 Introduction

One of the key challenges associated with electric vehicles is the availability and accessibility of charging infrastructure. To support the widespread adoption of electric vehicles, it is essential to establish an efficient network of charging stations that can cater to the needs of EV users. The strategic placement of charging stations plays a crucial role in ensuring the convenience and feasibility of electric vehicle usage, as drivers rely on the availability of charging facilities to meet their energy demands during journeys.
This thesis aims to address the problem of finding optimal locations for charging stations. The research will delve into various techniques and methodologies employed for the problem of location optimization, with a particular focus on the urban area of Genoa. By analyzing different approaches, evaluating their effectiveness, and comparing the obtained results, this thesis aims to contribute to the understanding of charging station location optimization and provide insights for future improvements in this domain.

The thesis is structured as follows:
In Chapter 2, we will delve into the theoretical foundations of electric cars, but we will not limit ourselves to that. We will also explore the reasons driving the adoption of electric cars and the crucial issue of sustainability. We will discuss the charging methods and types of charging. Additionally, we will analyze the increasing registration numbers and charging infrastructure in Italy. This section will provide a comprehensive overview of the reasons behind the growing adoption of electric cars and the importance of sustainability in the field of mobility.

Chapter 3 delves into the problem of finding optimal charging station locations. It investigates various techniques and algorithms used in location optimization, considering factors such as demand patterns, geographical constraints, and accessibility. The chapter provides an in-depth analysis of these methods, discussing their strengths, weaknesses, and applicability to real-world scenarios.

Chapter 4 presents the tools and technologies utilized in the development and implementation of the research. It outlines the software frameworks, data sources, and analytical approaches employed to model and analyze the charging station location problem.

Chapter 5 focuses on the case study conducted in the urban area of Genoa. It explores the specific challenges and opportunities presented by the city's characteristics and evaluates different strategies for optimizing charging station locations. The chapter provides detailed insights into the methodology employed, data collection processes, and experimental results.

Chapter 6 compares and contrasts the different approaches and techniques applied in the case study. It analyzes the performance, efficiency, and effectiveness of each method, highlighting the variations in results and the implications for charging station location optimization in similar urban areas.

Chapter 7 concludes the thesis by summarizing the findings, discussing the limitations of the study, and proposing future research directions.

By addressing the problem of finding optimal charging station locations and exploring various methodologies and techniques, this thesis contributes to the ongoing efforts in promoting the adoption of electric vehicles and advancing the development of sustainable transportation infrastructure.

2 Case study

The case study focuses on the city of Genova, the capital of the Liguria region and the largest and most populous municipality in Liguria, as well as the third largest in Northern Italy. Genova is located along the Ligurian Sea, covering an area of approximately 240 km^2, with a coastal strip stretching for about 30 km.
Currently, Genova is carrying out the "Genova Future City Map" project in collaboration with the Municipality of Genova, City Green Light (responsible for public lighting since 2020), and Wesii (a leading technological company in Europe specializing in multispectral aerial remote sensing applied to green issues).

The objective of this case study is to identify optimal locations for the installation of charging stations based on the model being examined. We will start with simple models that identify candidate sites, such as service areas or parking lots. We will then move on to the reachability model that finds optimal solutions based solely on the structure of Genova's road network. Finally, we will shift towards models based on traffic and the movement of cars between different areas of Genova. It is important to specify that the obtained results for the localization of charging stations do not take into account the sizing, hence the load capacity of the charging points. It is assumed that they are capable of meeting the daily demand required.

The models, adapted according to the case study, that we will consider are:

1. **Reachability Graph Model**: the analysis begins with the reachability graph model, as explained in the previous section. Starting from a road network, a reachability graph is created, which determines reachable points within a specified threshold for each point in the network. Once the reachability graph is obtained, the model is solved using the k-dominating-set approach with k=3, where each node not in the dominating set is adjacent to k nodes in the dominating set. For this initial model, only the road network of Genova is required.

2. **Uncapacitated Facility Location Model**: in this model, an uncapacitated facility location model, a set of candidate sites is available, and the goal is to find an optimal solution to assign each client to the site with the minimum cost. For this model, not only the road net-

work of Genova but also all the service areas in the city, considered as candidate sites, are required.

3. **Displacement-Based Models**: two displacement-based models will be analyzed: Flow Capturing and an Aggregate Arc Covering. These models rely on the road network, the areas covering different zones, and the origin-destination flows of people between zones.

In the following sections, we will examine the data retrieval process, as well as the various preprocessing and data mapping steps involved.

2.1 Open Data

The data necessary to create the models were obtained from the website of the Municipality of Genova, specifically from the thematic areas section where open data is available and categorized into groups.
In particular, the retrieved data includes the following:

- Service areas: The CSV file contains information about all service areas in the Municipality of Genova. These service areas include parking areas, refueling areas, and areas for roadside assistance. The file provides the coordinates of these areas in Monte Mario EPSG:3003 format, representing their shapes.

- Origin-Destination travel matrices: This CSV file represents the movements and trips within different zones of Genova, categorized by time intervals, purpose of travel, and the mode of transportation used. Each column in the CSV file is explained in a separate file called "descrizioneCampiSpostamenti.csv."

- Zoning shapefile: From the shapefile, a CSV file was extracted containing the names of the zones in Genova and the area of each zone represented as a polygon in Monte Mario EPSG:3003 format. This file will be useful for understanding the area and location of each zone in Genova, especially when considering movements between zones.

The CSV files containing polygon or multi polygon coordinates were preprocessed to convert them into EPSG:4326 format, which represents latitude and longitude coordinates. Once the coordinates were converted into the standard format, the files were imported into their respective Neo4j databases in order to examine various optimization models for charging stations.

In addition to the data provided by the Municipality of Genova, the download of Genova's road network was also crucial as the primary element for each model. Combining all of this information allowed for the creation of multiple data structures.
Specifically, the following data structures were developed:

- **whole_Genova**: This data structure represents Genova's road network and will be used for the Reachability Graph Model.
- **service_area**: This data structure includes service areas and Genova's road network, and it will be used for the Uncapacitated Facility Location Problem.
- **zone_point**: This data structure encompasses zone areas in Genova, movements between areas, and Genova's road network. It will be used for the Flow Based Models.

2.1.1 Preprossessing Polygon and Multi Polygon Data

As mentioned earlier, the files containing coordinate information for service areas and zones in Genova were preprocessed to retrieve coordinates in EPSG:4326 format from the original EPSG:3003 format.
The following function was executed on the CSV files "areeDiServizio.csv" for service areas and "zonizzazione.csv" for zones:

```
from shapely.geometry import MultiPolygon

source_crs = pyproj.CRS('EPSG:3003')
target_crs = pyproj.CRS('EPSG:4326')

# Build coordinate transformer
transformer = pyproj.Transformer.from_crs(source_crs, target_crs,
    always_xy=True)

for indice, riga in dataframe.iterrows():
    list_coord= []
    line = riga["geometry"]
    if isinstance(line, Polygon):
        lat_lon = []
        polygon = line

        # Polygon coordinates
        coordinates = list(polygon.exterior.coords)

        # Print coordinates
        for x, y in coordinates:
            longitude, latitude = transformer.transform(x, y)
```

```
            lat_lon.append(str(latitude) + ' ' + str(longitude))
        dataframe.at[indice, 'Coordinates'] = str(lat_lon)
    else:
        print("IS NOT A POLYGON")
```

The dataframe will store the lists of coordinates in the format: latitude, longitude.
After completing this data preprocessing step, let us delve into the detailed explanation of the three data structures that are essential for the aforementioned models.

2.2 whole_Genova database - Genova's road network

As a first step, obtaining the representation of Genova's road network was essential. After the download, the graph was saved in GraphML format and subsequently imported into Neo4j for data preprocessing.

```
Genova = ox.graph_from_place('Genova, Liguria, Italy',
                             network_type="drive")
ox.save_graphml(Genova, filepath="wholegenova.graphml",
                                gephi=True,encoding="utf-8")
```

The graph $Genova = (V, E)$ is composed of a set of nodes $v \in V$, representing intersections or roundabouts in the city, while the relationships $e \in E$ represent the roads between nodes. Before importing the graph, it is necessary to add the following configuration to the APOC.conf file:

`apoc.import.file.enabled=true`

This enables file import. Next, the graph is imported into Neo4j:

```
CALL apoc.import.graphml("wholegenova.graphml {readLabels:TRUE})
```

The graph appears as a set of interconnected nodes, where the information is present in the relationships. The following figure shows a portion of the graph in Neo4j:

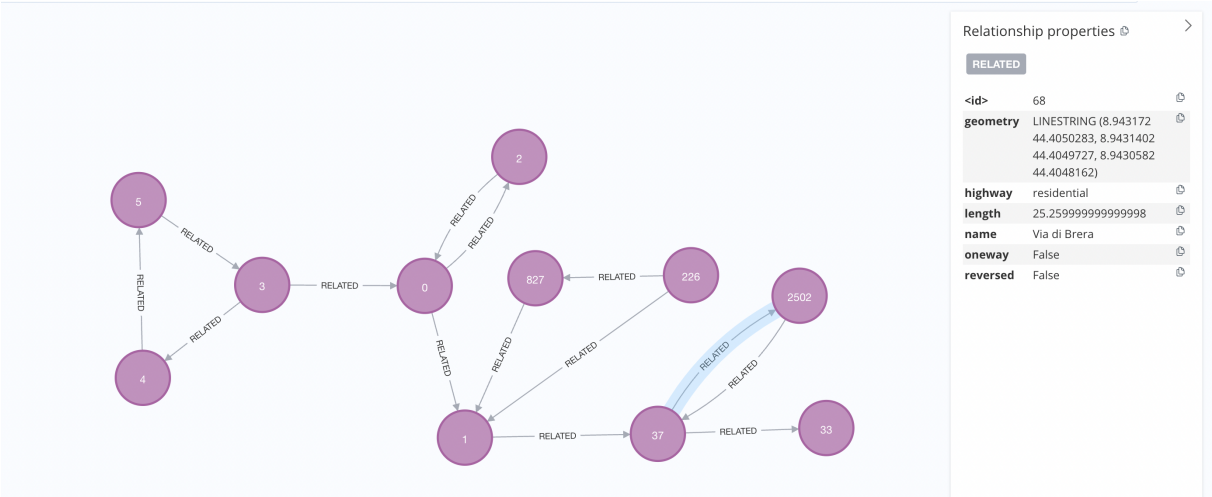


Figure 1:
Visualization of simple path of Genova in Neo4j

In the image, we can observe the properties for the selected edge. In particular, the *'geometry'* property contains a list of coordinates in string format representing the road, while the nodes at the endpoints represent the starting and destination points for that road.
Following the analysis of coordinate representation, a new property *'coordinates'* was added to each relationship, representing a list of coordinates in Neo4j's spatial point format:

```
point({longitude | x, latitude | y [, crs][, srid]})
```

Furthermore, after obtaining this list of points, the coordinates were also assigned to the origin and destination nodes. Specifically, the first coordinate of the *'coordinates'* list was assigned to the origin node, and the last coordinate was assigned to the destination node.
Query for relationships:

```
:auto MATCH (:Point)-[r:RELATED]->(:Point)
CALL {
WITH r
WITH replace(split(r.geometry, "(")[1],")","") as coordinates, r
WITH split(coordinates, ",") as coordinates, r
UNWIND coordinates as coord
WITH split(trim(coord), " ") as coord, r
WITH COLLECT (point({longitude:toFloat(coord[0]), latitude:
    toFloat(coord[1])})) as coordinates, r
SET r.coordinates = coordinates
} IN TRANSACTIONS OF 1000 ROWS
```

Query for nodes:

```
:auto MATCH (start:Point)-[r:RELATED]->(end:Point)
```

```
CALL {
WITH start, end, r
SET start.coordinates = r.coordinates[0]
SET end.coordinates = r.coordinates[-1]
} IN TRANSACTIONS OF 1000 ROWS
```

In Neomap, we can now display the points covering the area of Genova:

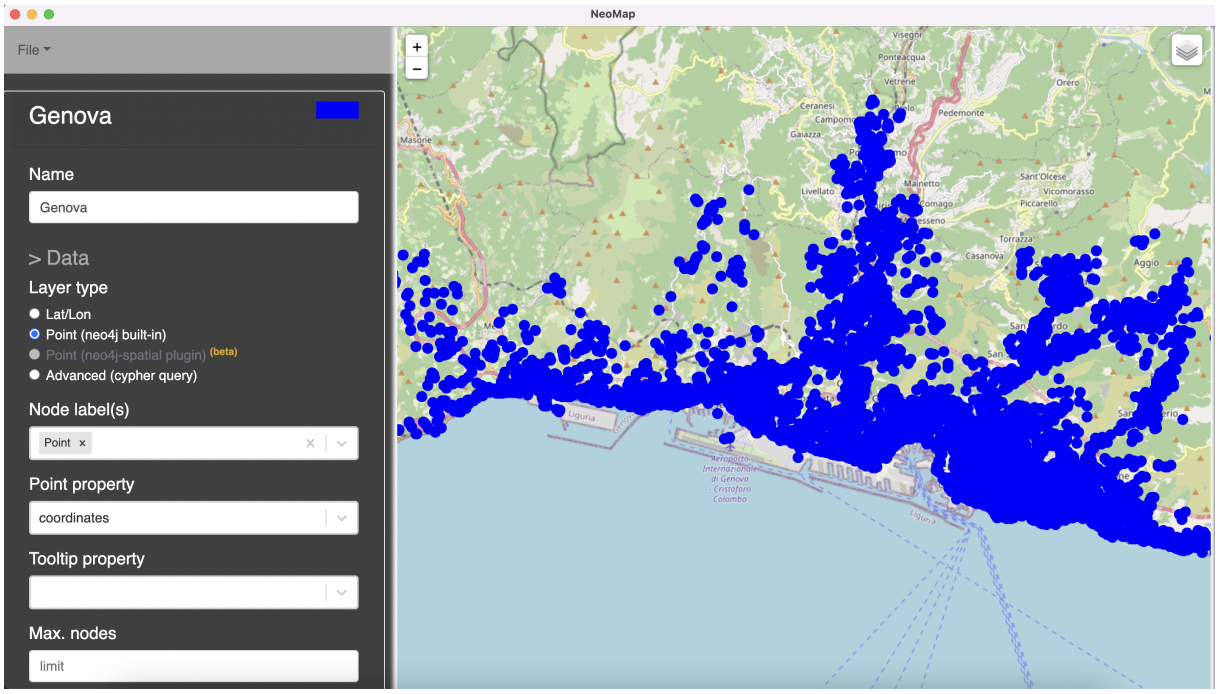


Figure 2:
Road network in whole_Genova databae

This will be represent the whole_Genova graph database useful for the Reachability Graph Model.

It is crucial to note that the network points in Genova will be considered as potential candidate locations for establishing charging stations in the models. However, since these points primarily correspond to roundabouts and intersections, they will serve as reference points where the stations should be activated in the surrounding areas.

2.3 service_area Database

The service_area database, as mentioned earlier, consists of service areas along with the road network of Genova. For the service areas, I did not focus on the entire station area but only took a single point from the list of polygon points to identify the location of the service area. Data import:

```
LOAD CSV WITH HEADERS FROM "file:///areeServizio.csv" AS line
With line
MERGE (p:Area {geometry:line.GEOMETRY})
SET p.perimetro=line.PERIMETRO, p.descrizio=line.DESCRIZIONE,
    p.superficie=line.SUPERFICIE, p.nome=line.NOME
```

After retrieving the coordinates, the (*Area*) nodes have been stored with the *points* property that contains latitude and longitude.

```
LOAD CSV WITH HEADERS FROM "file:///areeServizioLatLong.csv" AS line
With line
MATCH (a:Area)
WHERE id(a)=toInteger(line.idArea)
SET a.points = point({latitude:toFloat(line.LATITUDE),
    longitude:toFloat(line.LONGITUDE)})
```

Using the same APOC procedure as in the whole_Genova database, the road network was loaded into the service_area database, resulting in both data structures in a single database. A Cypher query was used to calculate the distance from each (*Point*) node in the road network to the (*Area*) nodes in the service areas. Subsequently, the [$DISTANCE\_STATION$] relationship was created between these pairs of nodes.

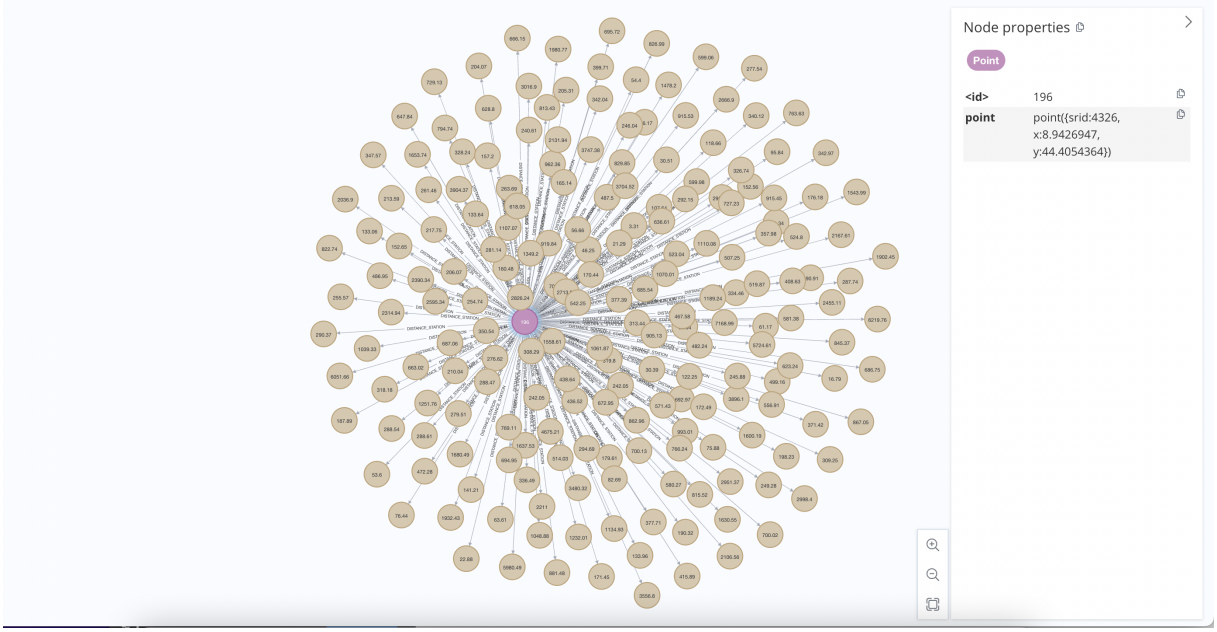


Figure 3:
Distance from Point to all Area nodes

```
MATCH (a:Area),(p:Point)
WITH a,p,distance(point({x: a.point.x, y: a.point.y, crs: 'wgs-84'}), point({x:
    p.coordinates.x, y: p.coordinates.y, crs: 'wgs-84'})) as dist
MERGE (p)-[:DISTANCE_STATION {distanza:toFloat(dist)}]->(a)
```

These distances between the (*Point*) and (*Area*) nodes will be considered in the Uncapacitated Facility Location model as the cost for a driver who is located at a specific point and needs to reach the area for recharging.

2.4 zone_point Database

To create the database in question, several steps were taken. Initially, all trips from one zone to another were imported from the matriceod_merci_viaggi.csv file as follows:

```
:auto USING PERIODIC COMMIT 1000
LOAD CSV WITH HEADERS FROM "file:///matriceod_merci_viaggi.csv" AS line
WITH line
MERGE (start:Point {codZona:toInteger(line.ORIG_COD_ZONA)})
set start.descZona=line.ORIG_DEN_ZONA
MERGE (end:Point {codZona:toInteger(line.DEST_COD_ZONA)})
set end.descZona=line.DEST_DEN_ZONA
CREATE (start)-[r:PATH_MERCI]->(end)
SET r.MEZZO_codice=toInteger(line.MEZZO_codice),
    r.NumViaggiGiornalieri=toFloat(line.NumViaggi),
r.MOTIVO_aggregato=line.MOTIVO_aggregato,r.TIPO_VIAGGIO=line.TIPO_VIAGGIO,
r.MOTIVO_codice=toInteger(line.MOTIVO_codice),
r.MEZZO_aggregato=line.MEZZO_aggregato,
r.FASCIA_ORARIA_VIAGGIO=toInteger(r.FASCIA_ORARIA_VIAGGIO),
r.MOTIVO_sistematicita=line.MOTIVO_sistematicita
```

Within each PATH relationship, the travel time slots are represented as integers 0, 1, and 2, which correspond to:

- 0: Other time slot
- 1: 6.30-9.00
- 2: 17.00-20.00

Therefore, the time slot descriptions are included within the relationship as well.

```
match (start:Zona)-[r:PATH]->(endN:Zona)
with r,
CASE r.FASCIA_ORARIA_VIAGGIO
WHEN 1 THEN "6.30-9.00"
WHEN 2 THEN "17.00-20.00"
ELSE "Altra fascia"
END as descFascia
SET r.descFascia=descFascia
```

Next, I clean the database by removing trips that were not made by cars and any trips that cross external zones of Genova:

```
MATCH p=(start:Zona)-[r:PATH]->(endN:Zona)
WHERE NOT r.TIPO_VIAGGIO="Interno" OR NOT r.MEZZO_aggregato="AUTO"
DELETE r
```

At this point, the zonification.csv file has been loaded, which contains the *geometry* field indicating the enclosed area for each zone in Genova.

```
:auto USING PERIODIC COMMIT 1000
LOAD CSV WITH HEADERS FROM "file:///zonizzazione.csv" AS line
WITH line
MERGE (z:Zona)
WHERE z.codZona == line.CODICE
MERGE (z)-[:SITUATED_IN]->(g:Geometry)
SET g.geometry = line.geometry
```

In the (*Geometry*) node, we have the coordinates expressed as polygons, which were obtained through the previous Python script to be loaded as a list of *points* nodes in Neo4j. Some zones in Genova are represented as multi polygons, so for those zones, the multi polygons were first divided into polygons using Cypher queries:

```
MATCH (g:Geometry)
WHERE g.geometry CONTAINS "MULTIPOLYGON"
with apoc.text.replace(g.geometry,"MULTIPOLYGON \(\(\(", "\(\(") as geo,g
with apoc.text.replace(geo,"\)\)\)","\)\)") as geo,g
with apoc.text.replace(geo,", \(\(",";\(\(") as geo,g
with apoc.text.replace(geo,"\(\(","POLYGON\(\(") as geo,g
with apoc.text.split(geo,";") as geo,g
unwind geo as singleG
return g.geometry as geom, singleG
```

This query splits the multi polygons into polygons and saves the query in a CSV file where each zone will have the associated polygons. Then, the Python script is executed again for these zones.
The obtained coordinates are saved in Neo4j. Specifically, for each (*Zona*) node representing the zones in Genova, there is a relationship to the (*Geometry*) node where the *linesCoordinates* property is stored as a list of coordinates in point format:

```
LOAD CSV WITH HEADERS FROM "file:///coordinates_from_Polygon.csv" as line
WITH line
WHERE line.Coordinates is not null
WITH line.Coordinates as coordinate, line
WITH REPLACE(coordinate,"'","") as coordinate,line
WITH REPLACE(coordinate,"[","") as coordinate,line
WITH split(coordinate,",") as coordinate,line
MATCH (p:Point)-[:SITUATED_IN]->(g:Geometry)
```

```
WHERE p.codZona=toInteger(line.CODICE)
UNWIND coordinate as singleC
WITH split(trim(singleC), " ") as coord, p,g
WITH collect(point({latitude:toFloat(coord[0]),longitude:toFloat(coord[1])}))
    as coordinates, p, g
SET g.linesCoordinates = coordinates
```

Visualization of all the zones in Genova after the various steps:

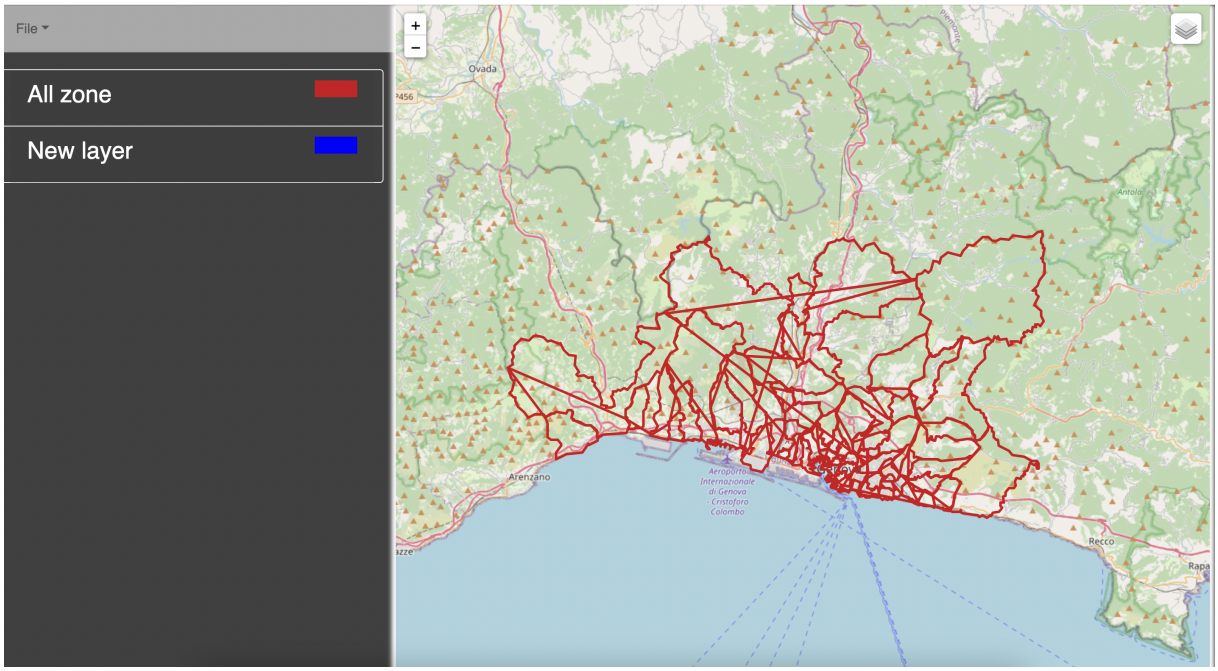


Figure 4:
all Genova's zone

The final step involved importing the road network of Genova and mapping each point node in the road network to its corresponding zone. For this last step, a Python notebook was created, which connected to the zone_point database to retrieve the coordinates of each point node and the coordinates of the zone polygons.
Database connection:

```
user = getpass('Neo4j user: ')
password = getpass('Neo4j pass: ')

uri = 'bolt://localhost:7687'
driver = GraphDatabase.driver(uri, auth=('neo4j','password'), encrypted=False)
```

Method to return the result of a Cypher query as a DataFrame:

```
def cypherQueryToDataFrame(query):
    with driver.session(database = 'neo4j') as session:
        result = session.run(query)
```

```
        return pd.DataFrame(result.data(), columns=result.keys())
```

Query to retrieve the coordinates of all the point nodes:

```
query = """
MATCH (p:Point)
Where p.coordinates is not null
RETURN p.latitude as latitude, p.longitude as longitude, id(p) as idPoint
"""

df = cypherQueryToDataFrame(query)
df
```

The coordinates are stored in a list of Points:

```
coordinate_punti = []
# Creo punti
for index, row in df.iterrows():
    coordinate_punti.append(Point(row['longitude'],row['latitude']))
```

Retrieval of coordinates for the zone nodes:

```
zone = """
MATCH (z:Zona)-[]->(g:Geometry)
WITH DISTINCT g.linesCoordinates as lines, z
UNWIND lines as line
RETURN z.descZona as nameZona, line.latitude as lat, line.longitude as long
"""

df1 = cypherQueryToDataFrame(zone)
df1
```

Create a dictionary of polygons where the key represents the zone and the value represents the coordinates:

```
zone_dict = {}
for index, row in df1.iterrows():
    zone = row['nameZona']
    lat = row['lat']
    long = row['long']
    if zone in zone_dict:
        zone_dict[zone].append((long, lat))
    else:
        zone_dict[zone] = [(long, lat)]

for i in zone_dict:
    zone_dict[i] = Polygon(zone_dict[i])
```

From here, the final mapping was performed for each point node to its corresponding zone, and the Zone-Point pair was saved in a dataframe:

```
mapping_zone_point = []
for point in coordinate_punti:
    for i in zone_dict:
        if point.within(zone_dict[i]):
            mapping_zone_point.append(i + "," + str(point))

Zona = []
Point = []
for dato in mapping_zone_point:
    parte1, parte2 = dato.split(',')
    Zona.append(parte1.strip())
    Point.append(parte2.strip())

# Creazione del dataframe
df = pd.DataFrame({'Zona': Zona, 'Point': Point})

df
```

Table 1: Mapping for each Point to Zone

	Zone	Point
4	CASTELLUCCIO	POINT (8.8051369 44.4251423)
0	S.VINCENZO	POINT (8.9426947 44.4054364)
1	S.VINCENZO	POINT (8.9423352 44.4055207)
2	S.VINCENZO	POINT (8.9428886 44.4061213)
3	S.VINCENZO	POINT (8.9431643 44.406231)
4	S.VINCENZO	POINT (8.9427624 44.4063012)
...	...	..
7488	PRATO	POINT (9.0174669 44.4523455)
7489	MULTEDO	POINT (8.8330017 44.4270671)
7490	MULTEDO	POINT (8.8326455 44.4269927)
7491	MULTEDO	POINT (8.8331272 44.4267759)
7492	CASTELLUCCIO	POINT (8.8051369 44.4251423)

After obtaining all the necessary information, the dataframe was exported to a CSV file to be loaded into Neo4j. For each (*Point*) node, a relationship [$APPARTIENE\_ALLA\_ZONA$] was created towards the zone node:

```
load csv with headers from "file:///mapping_zona_point.csv" as line
WITH line, apoc.text.replace(apoc.text.replace(line.Point,'POINT
    \(',''),'\)','') as latLong
WITH line.Zona as zona, apoc.text.split(latLong,' ') as coords
WITH zona, coords[0] as longitude, coords[1] as latitude
MATCH (p:Point)
WHERE p.coordinates.latitude = toFloat(latitude) AND p.coordinates.longitude =
    toFloat(longitude)
MATCH (z:Zona)
WHERE z.descZona = zona
MERGE (p)-[:APPARTIENE_ALLA_ZONA]->(z)
SET p.zona_di_appartenenza = zona
```

Mapping of a specific zone, "CERTOSA":

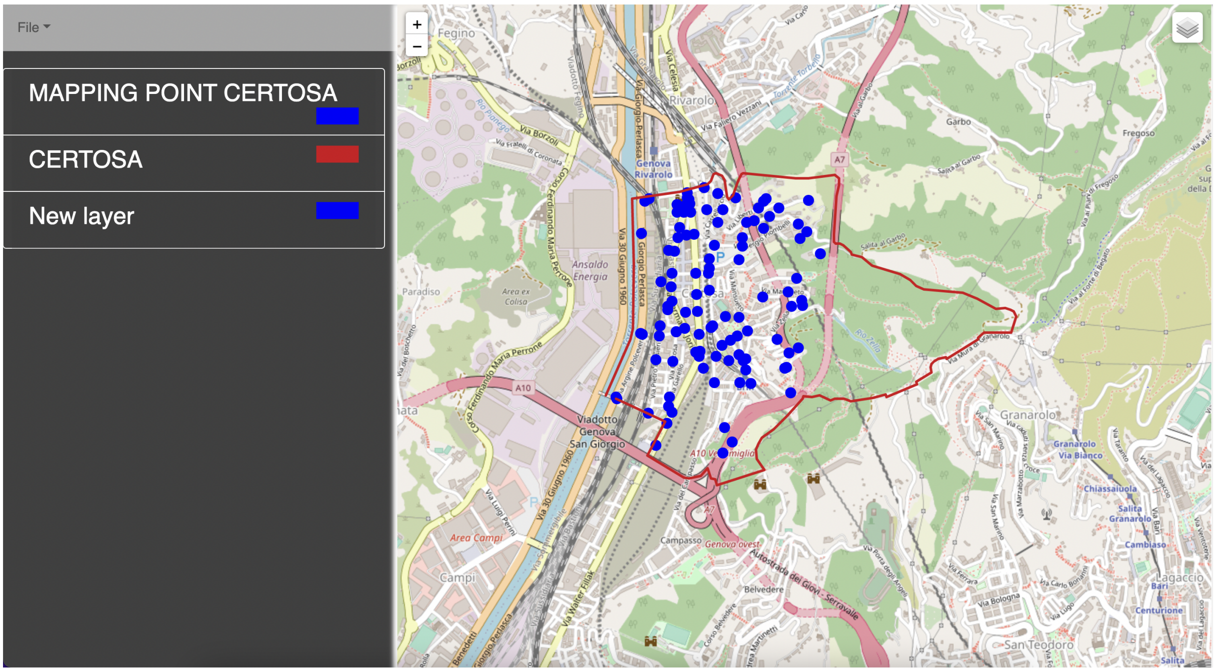


Figure 5:
Example of mapping for Certosa zone and Point

2.5 Uncapacitated Facility Location Model for Genova

As previously mentioned, the uncapacitated facility location problem, also known as the simple plant location problem, is a fundamental and extensively studied model in facility location theory. It is typically the first model considered when addressing location problems.
The uncapacitated facility location problem (UFLP) involves determining the optimal placement of an unspecified number of facilities to minimize the total cost, which includes both fixed setup costs and variable costs associated with serving customer demands from the selected facilities.
For our specific use case, we consider the set of potential locations, denoted as $N = 1, ..., n$, to represent the station area. The set of customers is denoted as $M = 1, ..., m$ and comprises all points in the road network. We focus on the following scenario: a driver needs to charge their electric vehicle and can be located at any point within the Genova road network. The objective is to find the nearest charging station. Naturally, reaching the charging station incurs a cost, which is calculated based on the distance between each point

and the station area. In this context, c_{ij} represents the distance cost from point i to charging station j.
Additionally, the station area data includes a description specifying the type of station. For example, in a service area where the driver is on a journey and needs to recharge their car as quickly as possible, a fast charging station may be installed. On the other hand, in a parking area where the driver is not in a hurry to recharge, a medium charging station could be installed. Consequently, different installation costs f_i are associated with each type of station installed, depending on its characteristics.

Type of station area:

```
typeArea = """
    MATCH (a:Area)
    WITH DISTINCT a.descrizio as typeArea
    RETURN typeArea
"""
typeArea = cypherQueryToDataFrame(typeArea)
typeArea
```

For each type of area installation cost will be:

Table 2: Type of areas

	typeArea	installationCost
0	AREA A PARCHEGGIO	1500€
1	STAZIONE DI RIFORNIMENTO CARBURANTE	2500€
2	AREA A SERVIZIO STRADALE	3500€

Assign for each facilities the installation costs:

```
Facilities = []
OpeningCosts = []
for index, row in df.iterrows():
    if row['typeArea'] == 'AREA A PARCHEGGIO':
        Facilities.append(row['Facilities'])
        OpeningCosts.append(1500)
    if row['typeArea'] == 'STAZIONE DI RIFORNIMENTO CARBURANTE':
        Facilities.append(row['Facilities'])
        OpeningCosts.append(2500)
    if row['typeArea'] == 'AREA A SERVIZIO STRADALE':
        Facilities.append(row['Facilities'])
        OpeningCosts.append(3500)

Plants = dict(zip(Facilities, OpeningCosts))
Plants
```

Retrieval all points and assign as Clients:

```
Clients = []

for index, row in Point.iterrows():
    Clients.append(row['clientPoint'])
```

The distance cost between point and station area are calculated as their distance:

```
query = """
MATCH (p:Point)-[r:DISTANCE_STATION]->(a:Area)
WHERE p.point is not null
RETURN id(p) as Clients, id(a) as Facilities, round(r.distanza,2) as costs
"""

costi = cypherQueryToDataFrame(query)
```

Table 3: distance costs from clients to facilities

	Clients	Facilities	distanceCost
0	196	29	11309.56
1	196	73	2983.52
2	196	136	8659.99
3	196	79	2744.34
4	196	133	8984.28

2.5.1 Model set up

Variables for the model are:

$$y_j = \begin{cases} 1 \text{ if facility } j \text{ is open,} \\ 0 \text{ otherwise} \end{cases}$$

$$x_{ij} = \begin{cases} 1 \text{ if point } i \text{ is served from facility} j, \\ 0 \text{ otherwise} \end{cases}$$

```
UFLP = gb.Model('FacilityLocation')

y = UFLP.addVars (Facilities, vtype=gb.GRB.BINARY, name='y')

UFLP.update()

x = UFLP.addVars (Clients, Facilities, vtype=gb.GRB.CONTINUOUS, name = 'x')

UFLP.update()
```

Objective function:

$$\min \sum_{j=1}^{n} f_j y_j + \sum_{i=1}^{m} \sum_{j=1}^{n} c_{ij} x_{ij}$$

```
UFLP.setObjective(y.prod(Plants) + x.prod(Costs), gb.GRB.MINIMIZE)
```

subject to:

1. demand of client i must be satisfied

$$\sum_{j=1}^{n} x_{ij} = 1 \text{ for } i = 1, \ldots, m$$

```
    UFLP.addConstrs((x.sum(i,'*') == 1 for i in Clients), name='Dem')
    UFLP.update()
```

2. A client can be served from facility j only if facility j is open:

$$x_{ij} \leq y_j \quad \forall i, j$$

```
    UFLP.addConstrs((x[(i,j)] <= y[j] for i in Clients for j in
        Facilities), name = 'VUB')

    UFLP.update()
```

2.5.2 Model optimize

```
UFLP.optimize()
```

Print the facilities to which each client has been assigned:

```
print('Obj: %g' % UFLP.objVal)

print ()

for v in UFLP.getVars():
    if v.x > 0.1:
        print ('%s=%g' % (v.varName, v.x), end = ' ')
```

The optimal solution found suggests opening 113 charging stations and we see how plants has to be activate:

```
active_plants = [j for j in Facilities if y[j].x > 0.5]
print(active_plants)
```

With cypher query retrieval the results and visualize on NeoMap the optimal location:

```
stazioni = """
WITH [0, 1, 2, 3, 4, 6, 8, 10, 11, 12, 14, 17, 19, 24, 26, 27, 28, 29, 31, 32,
    35, 37, 38, 43, 44, 45, 47, 48, 51, 52, 54, 61, 63, 65, 67, 68, 69, 70, 71,
    73, 75, 76, 77, 78, 79, 80, 81, 82, 84, 86, 87, 88, 90, 92, 96, 100, 103,
    107, 108, 109, 111, 112, 117, 119, 120, 124, 127, 128, 129, 132, 134, 135,
    136, 137, 138, 139, 141, 143, 144, 145, 146, 149, 151, 153, 154, 156, 157,
    158, 160, 161, 162, 163, 165, 166, 173, 174, 175, 177, 178, 179, 180, 181,
    182, 183, 184, 185, 186, 187, 189, 191, 192, 194, 195] AS station
MATCH (a:Area)
WHERE id(a) IN station
SET a:ChargingStation
RETURN a as StazioniDaAprire
"""
stazioni = cypherQueryToDataFrame(stazioni)
```

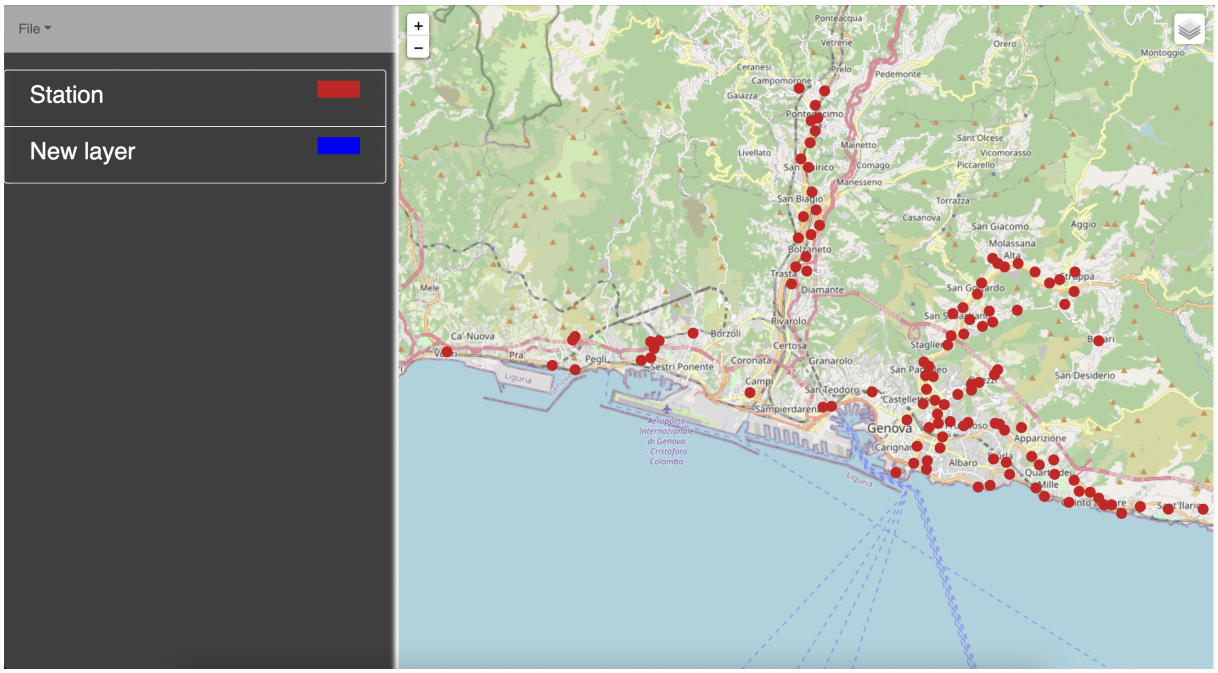


Figure 6:
Result for UFLP model

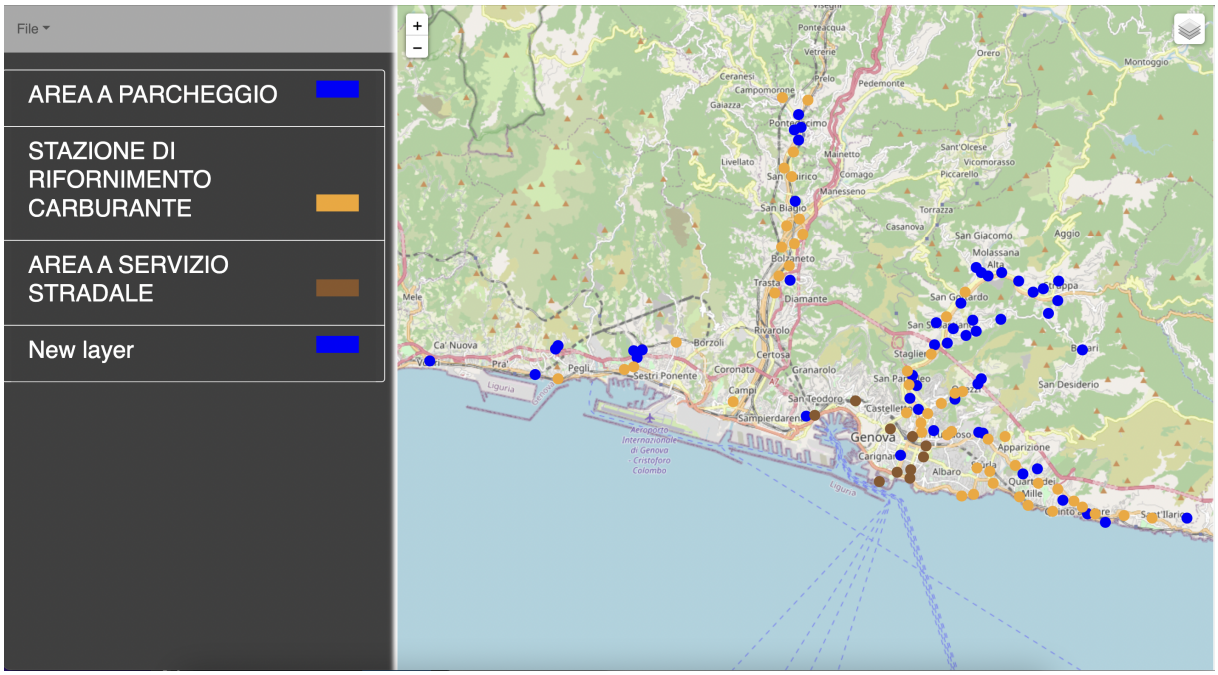


Figure 7:
Result divided by type Area

2.6 Reachability Graph Model for Genova

In the reachability graph model, we consider the road network represented by $G^s = (V^s, E^s, w : E^s \to R)$. Here, V^s and E^s denote the set of nodes and edges, respectively, and $w(e)$ represents the length of the corresponding road segment in meters. To construct the reachability graph $G_t^r = (V^s, E_t^r)$, we define edges $uv \in E_s^r$ if and only if the shortest path distance between nodes u and v is less than a specific threshold t (set to 3 km in this case).

The construction process involves the following steps:

1. Read the graph G^s

```
G = nx.read_graphml("wholegenovaExport.graphml")

for u, v, attr in G.edges(data=True):
    attr['length'] = float(attr['length']) # Convert to float value
```

2. Build a shortest paths dict from the all_pairs_dijkstra_path_length that calculate distance from each node:

```
shortest_paths = nx.all_pairs_dijkstra_path_length(G, weight='length')

shortest_paths = dict(shortest_paths)
```

Here, nx.all_pairs_dijkstra_path_length is used because we are interested in the distances rather than the actual shortest paths.

3. Construct the reachability graph by examining each node in the original graph. If the distance between node u and v is less than 3 km, we add an edge uv to the reachability graph with the corresponding weight:

```
# Initialize reachability graph as directed graph
reach_graph = nx.DiGraph()

# Iterate on each node pairs and set the distance
for node1 in G.nodes():
    for node2 in G.nodes():
        if node1 != node2:
            try:
                if shortest_paths[node1][node2]:
                    distance = shortest_paths[node1][node2] #retrieval
                        distance
                    if distance <= 3000:
                        # add edge with weight
                        reach_graph.add_edge(node1, node2, weight=distance)
            except:
                continue
```

Created the reachability graph, the dominating set was applied, first to find the minimum dominating set, and subsequently the k-dominating set with $k > 1$. The k-dominating set was modeled with different values of k to study the growth and increase of charging stations under different constraints.
Now let's examine the Minimum Dominating Set, which ensures that all nodes not in the dominating set have at least one adjacent node in the dominating set. The solve_dominating_set method utilizes the Gurobi library to formulate and solve the Dominating Set problem, returning the minimum cardinality dominating set in the given graph, if found.

2.6.1 Minimum Dominating Set

```
def solve_k_dominating_set(graph, k):
    model = gp.Model("dominating-set")

    # decision variables
    dominating_set = model.addVars(graph.nodes, vtype=GRB.BINARY,
        name="dominating_set")

    # objective function
    model.setObjective(dominating_set.sum(), GRB.MINIMIZE)
```

```
    # Constraint: Every node must either be in the dominating set or have at
        least one neighbor in the dominating set.
    for node in graph.nodes:
        successors = graph.successors(node)
        model.addConstr(dominating_set[node] +
            gp.quicksum(dominating_set[neighbor] for neighbor in successors) >=
            1)

    model.optimize()

    if model.status == GRB.OPTIMAL:
        dominating_set_solution = [node for node in graph.nodes if
            dominating_set[node].x > 0.5]
        return dominating_set_solution
    else:
        return None

solve_k_dominating_set(reach_graph, 3)
```

The solve_k_dominating_set method solves the Dominating Set problem with the objective of finding a minimum cardinality dominating set in the reachability graph.
Steps are the following:

1. Creation of decision variables: A binary variable dominating_set is created for each node in the graph. These variables represent whether a node is part of the dominating set or not.

2. Definition of the objective: The objective is to minimize the sum of the dominating_set variables, which corresponds to the cardinality of the dominating set.

3. Addition of the constraint that every node must either be in the dominating set or have at least one neighbor in the dominating set. This constraint ensures that every node is either included in the dominating set or has at least one neighbor included in the dominating set.

4. Solving the model using model.optimize(). The optimization is performed to find the optimal solution to the Dominating Set problem.

The optimal solution found suggests opening 92 charging stations.

```
Gurobi Optimizer version 10.0.1 build v10.0.1rc0 (mac64[rosetta2])

CPU model: Apple M1
Thread count: 8 physical cores, 8 logical processors, using up to 8 threads

Optimize a model with 8978 rows, 8977 columns and 6128096 nonzeros
Model fingerprint: 0x1be121d9
Variable types: 0 continuous, 8977 integer (8977 binary)
Coefficient statistics:
  Matrix range     [1e+00, 1e+00]
  Objective range  [1e+00, 1e+00]
  Bounds range     [1e+00, 1e+00]
  RHS range        [1e+00, 3e+00]
Found heuristic solution: objective 117.0000000
Presolve removed 8978 rows and 8977 columns
Presolve time: 0.91s
Presolve: All rows and columns removed

Explored 0 nodes (0 simplex iterations) in 1.10 seconds (1.37 work units)
Thread count was 1 (of 8 available processors)

Solution count 2: 92 117

Optimal solution found (tolerance 1.00e-04)
Best objective 9.200000000000e+01, best bound 9.200000000000e+01, gap 0.0000%
```

By executing a Cypher query, we can print and retrieve the nodes where the charging stations should be opened:

```
WITH
    ['n299','n408','n834','n1090','n2188','n3743','n8731','n3583','n4457','n7770',
    'n8622','n71','n3155','n6987','n4140','n701','n1517','n798','n4237','n444',
    'n4271','n3305','n926','n532','n1177','n4510','n319','n582','n323','n5513',
    'n925','n1145','n4290','n4487','n340','n5210','n709','n714','n2697','n5842',
    'n613','n5253','n6954','n5734','n5851','n6911','n8447','n635','n876','n4439',
    'n7179','n5473','n913','n4638','n5549','n5879','n5977','n3071','n7513','n1160',
    'n6661','n6674','n1095','n5520','n4866','n6284','n6311','n6538','n4758','n3291',
    'n7072','n6344','n4929','n8128','n3564','n5751','n5821','n5755','n4489','n4694',
    'n4910','n5563','n5338','n5649','n5586','n5542','n5540','n5778','n6790','n5617',
    'n5661','n5942'] as nodes
UNWIND nodes AS singleNode
WITH apoc.text.replace(singleNode,'n','') as id
MATCH (p:Point)
WHERE id(p)=toFloat(id)
SET p:Station
```

We can visualize the positions for the charging stations using Neomap:

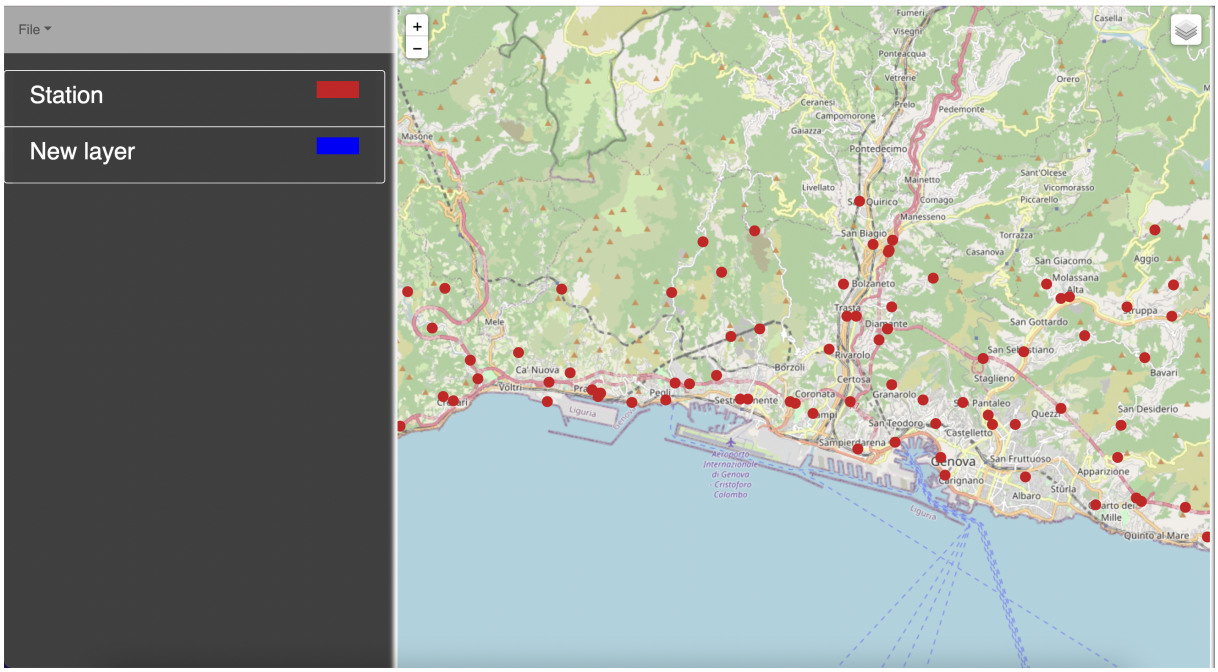


Figure 8:
result charging station with reachability graph k-dominating set

2.6.2 K-Dominating Set

o construct the k-dominating model, a new constraint was added to ensure that for every node i not in the dominating set, the sum of its neighbors j that belong to the dominating set is at least k.
A constraint is added for each node not present in the dominating set. For each node i in the list of subsets subsets.keys(), the list of neighbors is extracted. The constraint requires that the sum of variables $x[j]$ corresponding to the neighbors j is greater than or equal to k times the difference between 1 and $x[i]$. In other words, if node i is not in the dominating set ($x[i] = 0$), then at least k of its neighbors must be in the dominating set.
New model with $k = 2$ will be:

```
model = gp.Model("2DominatingSet")

nodes = list(reach_graph.nodes)
x = model.addVars(nodes, vtype=GRB.BINARY, name="x")

# Objective function: minimize the number of nodes in the dominating set
model.setObjective(gp.quicksum(x[i] for i in nodes), GRB.MINIMIZE)

# Constraint: each node not in the dominating set must have at least 2
    neighbors in the dominating set
for i in list(subsets.keys()):
    neighbors = subsets[i]
    model.addConstr(gp.quicksum(x[j] for j in neighbors) >= 2 * (1 - x[i]))
```

```
model.update()

model.write('2dominatingSet.lp')

model.optimize()
```

The model for $k = 2$ suggests opening 160 charging stations.

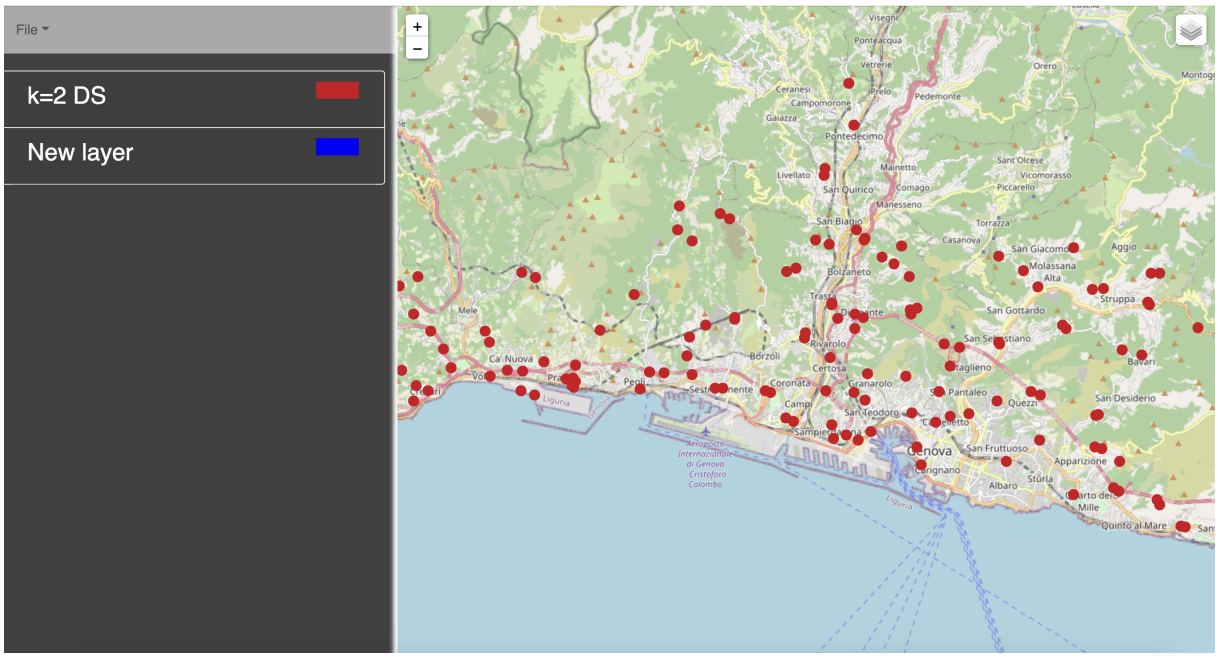


Figure 9:
Charging station for k=2 dominating set

The model was executed with different values of k, specifically $k = 3$ and $k = 4$, to observe the growth of charging stations.
For $k = 3$, it is necessary to open 221 charging stations:

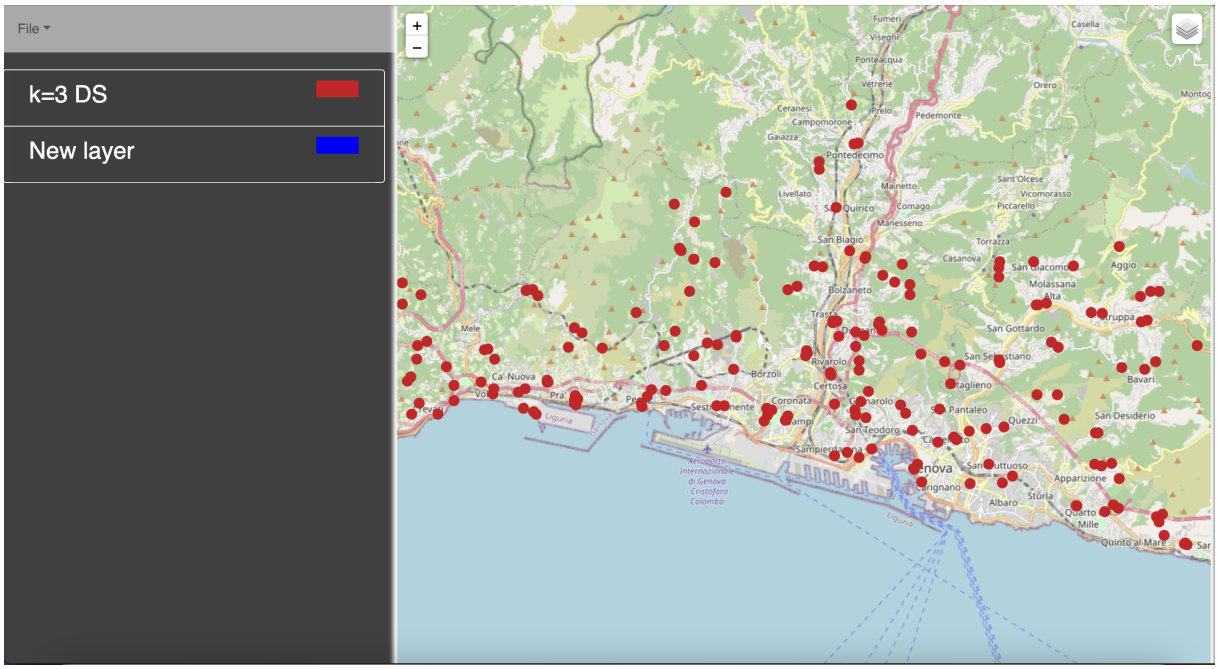


Figure 10:
Charging station for k=3 dominating set

For $k = 4$, we need to open 277 charging stations:

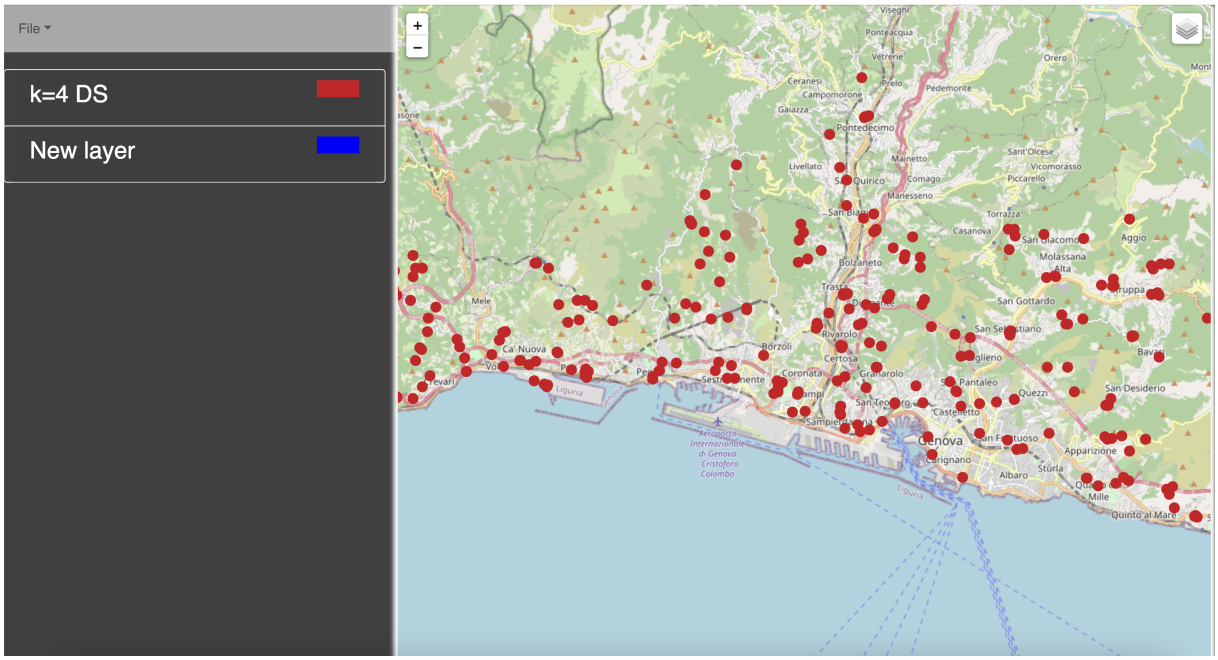


Figure 11:
Charging station for k=4 dominating set

Therefore, for different values of k, we have different numbers of charging stations to open. The obtained values are summarized in the following table:

Table 4: Results dominating set

k-value	open charging station
1	92
2	160
3	221
4	277

Results in NeoMap:

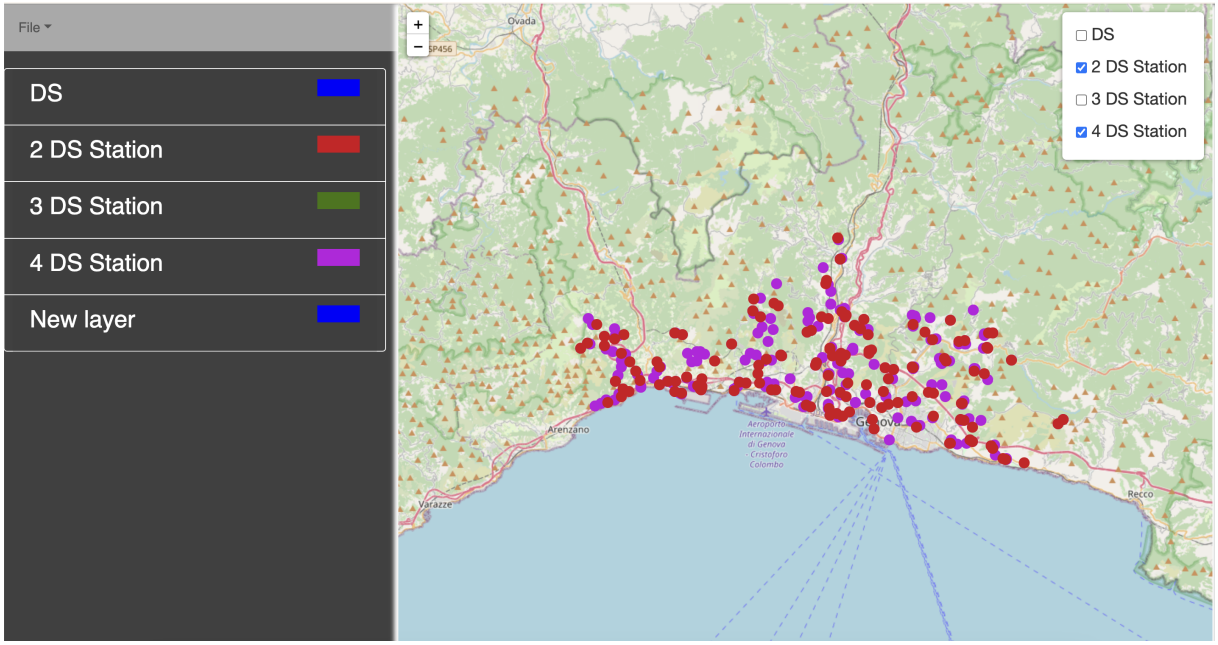


Figure 12:
compare k=2 and k=4

The proposed models identifies optimal solutions by considering the road network's structure and determining suitable locations for opening stations that can be accessed from various points within the network. However, it does not account for the busiest areas, including the flow of traffic and drivers' demands within specific zones in Genova. To address this limitation, two additional flow-based models have been implemented to consider the drivers' requests throughout the day.

2.7 Displacement-Based Models for Genova

In this section, we will analyze displacement-based models, specifically focusing on automobile movements between zones. We will primarily consider the Flow Capturing and Aggregate Arc Covering models.
For both models, we will utilize the graph *zone_point_graph.graphml*, which is derived from the zone_point database. As discussed earlier, this

graph represents the road network of Genova, with each node *Point* mapped to its respective zone, along with information on displacements between zones, including time slots and daily trip counts.

After establishing the connection to the database from the Python notebook, let's discuss some crucial considerations for the mathematical models:

1. the size of a zone is determined by the number of *Point* nodes within it, obtained through Cypher queries:

```
MATCH (z:Zona)<-[]-(p:Point)
WITH z.descZona as zona, p
RETURN zona, count(p) as mappingPoint
ORDER BY mappingPoint DESC
```

   Here are some sample results:

Table 5: Point for each zone

	Zone	mappingPoint
0	VOLTRI	261
1	BOLZANETO	244
2	MOLASSANA	200
3	S.FRUTTUOSO	172
4	PALMARO	170
...	...	..
69	MADDALENA	33
70	MOLO	19
71	CERANESI	1
72	MONTOGGIO	1
73	SERRA RICCO'	1

2. Based on the zone size, one or more nodes have been selected to simulate one or more Origin-Destination (OD) paths. The specified range for the total points in the zones (totP) is as follows:

Table 6: range for points in each zone

Range	PointToTake
$totP < 11$	1
$10 < totP < 51$	3
$50 < totP < 151$	5
$totP > 150$ 8	

```
    query = """
MATCH (z:Zona)<-[]-(p:Point)
WITH z.descZona as zone, p
```

```
RETURN zone, count(p) as totalPoint,
CASE
  WHEN count(p) < 11 THEN 1
  WHEN count(p) > 10 and count(p) < 51 THEN 3
  WHEN count(p) > 50 and count(p) < 151 THEN 5
  ELSE 8
END AS numNodesToTake
ORDER BY numNodesToTake DESC
"""

zone = cypherQueryToDataFrame(query)
zone
```

Some sample results:

Table 7: dataframe 'zone'

	zone	totalPoint	numNodesToTake
0	VOLTRI	261	8
1	PALMARO	170	8
2	PEGLI	160	8
3	SESTRI	152	8
4	S.GIOVANNI BATTISTA	162	8

3. we will examine the traffic flow between zones during all time slot:

```
flusso_traffico_1 = """
MATCH (z:Zona)<-[]-(:Point)
WITH z
MATCH path=(z)-[r:PATH]-(z1:Zona)
where (z1)<-[:APPARTIENE_ALLA_ZONA]-(:Point)
WITH DISTINCT path, r, z,z1
RETURN z.descZona as origin, z1.descZona as destination,
       round(sum(r.NumViaggiGiornalieri),0) as numTrip
order by numTrip desc
"""

ODtrip = cypherQueryToDataFrame(flusso_traffico_1)
ODtrip
```

Some sample results:

Table 8: Some results for dataframe 'ODtrip'

	origin	destination	numTrip
0	PEGLI	SESTRI	2858.0
1	SESTRI	PEGLI	2858.0
2	SESTRI	CORNIGLIANO	2375.0
3	CORNIGLIANO	SESTRI	2375.0
4	SESTRI	S.GIOVANNI BATTISTA	2178.0

4. For each origin-destination node pair, the shortest path has been calculated on the subgraph $filtered\_graph$. This subgraph is derived from the original graph G and includes only the 'RELATED' relationships of the road network in Genova.

```
G = nx.read_graphml('zone_point_graph.graphml')

filtered_graph = nx.DiGraph()
for u, v, attr in G.edges(data=True):
    if attr.get('label') == 'RELATED':
        node_attributes_u = G.nodes[u]
        node_attributes_v = G.nodes[v]
        filtered_graph.add_node(u, **node_attributes_u)
        filtered_graph.add_node(v, **node_attributes_v)
        attr['length'] = float(attr['length']) # Conversione del
            peso in un valore numerico
        filtered_graph.add_edge(u, v, **attr)
```

The uniqueness of these models lies in their ability to study the placement of electric columns by understanding user demand and, consequently, the potential gain from positioning the column in one area rather than another. Before delving into the details of the models, let us introduce some useful functions for problem creation.
For each movement between pairs of previously calculated and stored zones in the ODtrip dataframe, we randomly select n nodes, referred to as $Points$, from the origin zone, and m nodes, again referred to as $Points$, from the destination zone in order to calculate $n \times m$ paths. The paths will be computed using Dijkstra's shortest path algorithm through the function $calcola\_shortest\_path$. This function randomly selects the nodes for the origin and destination zones and then calls the algorithm from the networkX library.

```
def calcola_shortest_path(name_origine,name_destinazione,nodi_origine,
nodi_destinazione,subgraph_nodes):
    for index, row in zone.iterrows():
        if name_origine == row['zone']:
            num_path_o = int(row['numNodesToTake'])
            random_origine = random.sample(nodi_origine, num_path_o)
        if name_destinazione == row['zone']:
            num_path_d = int(row['numNodesToTake'])
            random_destinazione = random.sample(nodi_destinazione, num_path_d)


    for s in random_origine:
        for d in random_destinazione:
```

```
            try:
                print('Calcolo trip from: ' + str(s) + ' to '+ str(d))

                shortest_path = nx.shortest_path(filtered_graph, s, d,
                    weight='length', method='dijkstra')
                print(shortest_path)
                subgraph_nodes.append(shortest_path)

                subgraph_nodes.append(path)

                nodes_tot.update(path)

            except:
                continue
```

For each pair of zones, we will invoke the function *calcola_shortest_path*:

```
subgraph_nodes = []

for index, row in ODtrip.iterrows():

    nodi_origine = []
    nodi_destinazione = []

    name_origine = row['origin']
    name_destinazione = row['destination']

    pattern_o = re.compile(name_origine, re.IGNORECASE)
    pattern_d = re.compile(name_destinazione, re.IGNORECASE)

    for node, attrs in G.nodes(data=True):
        if 'zona_di_appartenenza' in attrs and
            pattern_o.search(attrs['zona_di_appartenenza']):
            nodi_origine.append(node)
        if 'zona_di_appartenenza' in attrs and
            pattern_d.search(attrs['zona_di_appartenenza']):
            nodi_destinazione.append(node)

    calcola_shortest_path(name_origine,name_destinazione,nodi_origine,
    nodi_destinazione,subgraph_nodes)
```

After calculating all the paths for each pair of zones, we will save the zones that each path traverses in a dictionary called *path_dict*. The calculation of the zones is defined by the following function, *calcolo_zone_attraversate*:

```
def calcolo_zone_attraversate(path):
    zone_attraversate = []
    for i in range(len(path) - 1):
        u=path[i]
        if filtered_graph.has_node(u) and 'zona_di_appartenenza' in
            filtered_graph.nodes[u]:
```

```
            zona = filtered_graph.nodes[u]['zona_di_appartenenza']
            if not zona in zone_attraversate:
                zone_attraversate.append(zona)
        else:
            continue
    return zone_attraversate

path_dict = dict()
numPath = 0
for i in subgraph_nodes:
    path_dict[numPath] = calcolo_zone_attraversate(i)
    numPath = numPath + 1
```

With this function, we will have the traversed zones for each path. This will allow us to calculate the total flow, which is simply the sum of the total number of trips for each pair of traversed zones. For example, considering a movement from the "PEGLI" zone to the "VOLTRI" zone, it can already be deduced that it will pass through other zones in order to reach the destination.

Figure 13:
Path from Pegli to Voltri

Assuming that the zones "CA NUOVA" and "PRA'" are traversed, the total flow from "PEGLI" to "VOLTRI" will be the sum of the column "$numTrip$" in the ODtrip dataframe.

Table 9: total flow

	origin	destination	numTrip
0	PEGLI	VOLTRI	356.0
1	PEGLI	PRA'	88.0
2	PRA'	CA NUOVA	60.0
3	CA NUOVA	VOLTRI	85.0

For this example, the total flow will be $356.0 + 88.0 + 60.0 + 85.0 = 589.0$. The function *calcolo_flusso* will be responsible for calculating the flow for each path:

```
def calcolo_flusso(zone_att, flows_dict):
total_flow = 0

if len(zone_att) == 1:
    o_d = zone_att[0]
    if o_d in flows_dict:
        total_flow = flows_dict[o_d]
    else:
        for index, row in df1.iterrows():
            if row['origin'] == o_d and row['destination'] == o_d:
                total_flow = float(row['numTrip'])
                flows_dict[o_d] = total_flow
                break
else:
    if len(zone_att) >= 3:
        for i in range(len(zone_att) - 1):
            origin = zone_att[i]
            destination = zone_att[i + 1]
            if (origin, destination) in flows_dict:
                total_flow += flows_dict[(origin, destination)]
            else:
                for index, row in df1.iterrows():
                    if row['origin'] == origin and row['destination'] ==
                        destination:
                        flow = float(row['numTrip'])
                        total_flow += flow
                        flows_dict[(origin, destination)] = flow
                        break

    if (zone_att[0], zone_att[-1]) in flows_dict:
        total_flow += flows_dict[(zone_att[0], zone_att[-1])]
    else:
        for index, row in df1.iterrows():
            if row['origin'] == zone_att[0] and row['destination'] ==
                zone_att[-1]:
                flow = float(row['numTrip'])
                total_flow += flow
                flows_dict[(zone_att[0], zone_att[-1])] = flow
```

```
                break

    return total_flow
```

2.7.1 Flow Capturing for Genova

In the flow capturing model a trip q is considered covered or captured if at least one Charging Station is opened along q. Let $filtered\_graph = (V, E)$ the graph that represents the road network where and $V = 1, ..., k$ are the intersections where charging stations can be installed and $E = 1, ..., m$ are the roads linked with nodes. The model includes binary variables x_k representing station location at every node k and binary variables y_q representing trip q coverage decisions. The objective of the model is to cover the paths while maximizing the flow. After defining the objective function and the variables, we establish constraints for path coverage and the maximum number of stations to be opened. The function $set\_node\_cover\_constraint$ ensures that each path $q \in Q$ is covered:

```
def set_node_cover_constraint(model, paths, x):
num_paths = len(paths)
num_nodes = len(x)

for i in range(num_paths):
    path = paths[i]
    expr = 0

    for node in path:
        expr += x[node.replace("n",'')]

    model.addConstr(expr >= y[i], f"NodeCoverConstraint_{i}")

set_node_cover_constraint(model, subgraph_nodes, x)
```

The constraint on the maximum number of charging stations is set as follows:

```
model.addConstr(x.sum() == 150)
```

The result of the model is as follows:

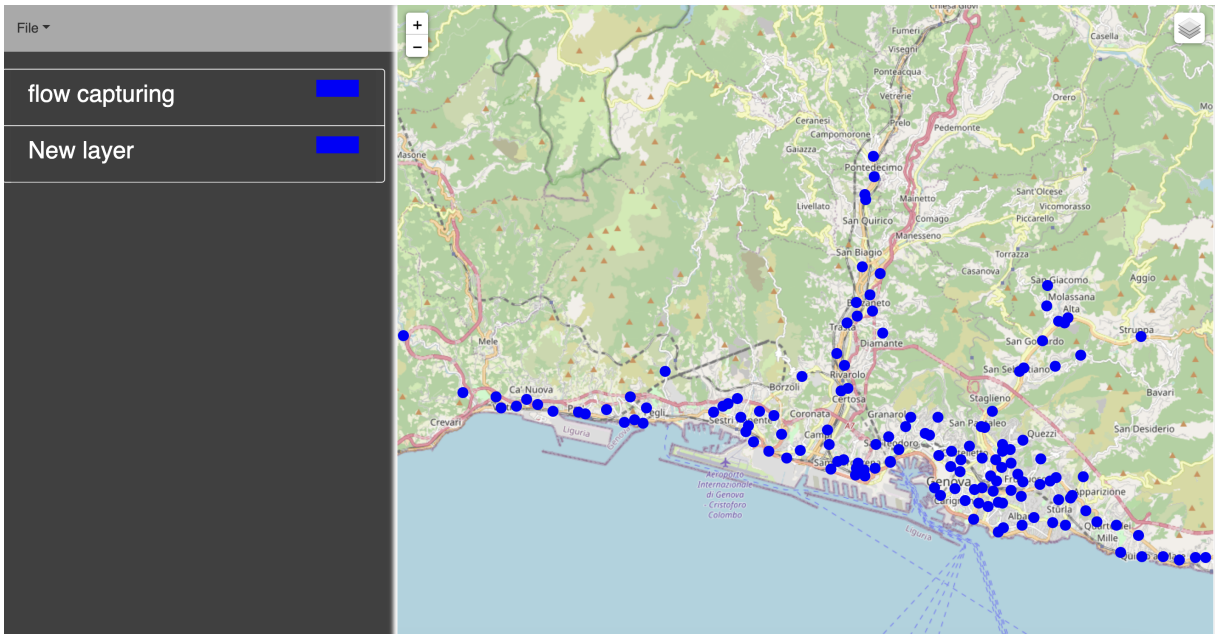


Figure 14:
Flow capturing result

It is evident from the figure that the charging stations are concentrated in the central area of Genova, while still being distributed in the outskirts of the city. This result provides us with information on the level of traffic congestion in the central area of Genova compared to the surrounding zones.

2.7.2 Aggregate Arc Covering for Genova

The last analyzed model aims to achieve two main objectives:

- Cover every trip $q \in Q$.
- All nodes in trip q where no charging station has been opened should be able to reach the open stations along q within a maximum distance of k meters.

We can consider this model as the combination of the graph reachability model and the flow capturing model. It is important that every path is covered, while at the same time, the charging stations must be reachable. This represents a positioning problem that meets the needs of drivers who may need to recharge their vehicles along the route at any given moment.

To solve this problem, an incremental heuristic was implemented because the constraint on node reachability combined with the trip coverage constraint resulted in a very large model that couldn't reach a solution. This incremental

heuristic was designed as an aggregation of results for each pair of zones at each iteration.
For example, considering the movement between the "VOLTRI-PEGLI" zones, the model is constructed, and n stations are found to be opened for all trips between these two zones. Then, another pair is considered, such as "PEGLI-SESTRI", and the stations to be opened for all trips between these two zones are determined, with the additional constraint that if there are nodes in the path from PEGLI to SESTRI where a station has already been opened, those nodes should remain active.
This solution will lead to opening new stations if a path is not yet covered or if the opened stations are insufficient to meet the reachability constraints. Conversely, it will not open new stations if the model already finds solutions with the previously opened stations.

To define the function that calculates the reachability for each node in a specific path, we first retrieve all the distances for each pair of nodes:

```
shortest_paths = nx.all_pairs_dijkstra_path_length(filtered_graph,
    weight='length')
shortest_paths = dict(shortest_paths)
```

At this point, the reachability_constraint function can access the dictionary and retrieve the distances between the nodes:

```
def reachability_constraint(path):
    tot_node = {}
    for i in range(len(path)-1):
        nodi_raggiungibili = []
        nodo_attuale = path[i]
        nodi_successivi = path[i+1:]

        for nodo in nodi_successivi:
            if shortest_paths[nodo_attuale][nodo]:
                distanza = shortest_paths[nodo_attuale][nodo]
                if distanza < 700:
                    nodi_raggiungibili.append(int(nodo.replace('n','')))
        nodi_raggiungibili.append(int(nodo_attuale.replace('n','')))

        tot_node[int(nodo_attuale.replace('n',''))] = nodi_raggiungibili
    return tot_node
```

Unlike flow capturing, each time we iterate over the ODtrip dataframe and calculate the shortest paths for each pair of nodes between the origin-destination zones, the flow will be computed for all these paths. We will create and solve the model to find the optimal solution. The *create_model* function takes as

input all the paths for the analyzed pair of zones, the nodes, and the flow for each path. It also takes *model_num*, which keeps track of the total number of models, and *tot_results*, a list that will be updated by inserting the number of activated stations during the application of the models.

```
cs_da_aprire = []

def create_model(nodi_interi,subgraph_nodes,flusso_tot,model_num,tot_results):

    model = gp.Model()

    x = model.addVars(nodi_interi, vtype=GRB.BINARY, name='x')
    y = model.addVars(len(subgraph_nodes), vtype=GRB.BINARY, name='y')

    obj_expr = gp.LinExpr()

    for i in range(len(subgraph_nodes)):
        obj_expr += flusso_tot[i] * y[i]
    model.setObjective(obj_expr, GRB.MAXIMIZE)

    activated = 0
    if model_num > 0:
        cs = list(tot_results)

        for i in cs:
            if i in x:
                activated = activated + 1
                print("CS PRESENTE: " + str(i))
                model.addConstr(x[i] == 1, "vincolo_somma")
    print("ACTIVATE CS= " +str(activated))
    num_path = 0
    for path in subgraph_nodes:
        reach_node = reachability_constraint(path)
        num_node = 0
        for i in reach_node:
            model.addConstr(gp.quicksum(x[nodo_ragg] for nodo_ragg in
                reach_node[i]) >= y[num_path],
                name=f"vincolo_reachability_{num_path}_{num_node}")
            num_node = num_node + 1
        num_path = num_path + 1

    if activated > 0:
        model.addConstr(x.sum() == activated)
    else:
        model.addConstr(x.sum() == activated + 2)

    model.update()

    model.optimize()

    results = []
```

```
    if model.status == GRB.OPTIMAL:
        for nodo in nodi_interi:
            x_value = x[nodo].x
            if x_value > 0.5:
                results.append(int(nodo))
    else:
        add_cs(nodi_interi,subgraph_nodes,flusso_tot,model_num,tot_results)

    print("MODEL COUNT: " + str(model_num) + " RESULTS: " + str(results))

    return results
```

When $model\_num == 0$, it indicates that we are creating the model for the first time, specifically considering the first row of the ODtrip dataframe. Therefore, there are no active charging stations yet.
Conversely, if $model\_num > 0$ we iterate over the list of all active charging stations. If one or more charging stations are present in the x variables, a constraint is set to indicate that these charging stations must remain active.

The *activated* counter indicates the number of charging stations that will remain active. If $activated > 0$, we will attempt to find an optimal solution without adding new charging stations. Otherwise, two additional charging stations are added using the following code:

```
if activated > 0:
    model.addConstr(x.sum() == activated)
else:
    model.addConstr(x.sum() == activated + 2)
```

When $activated > 0$, it is not guaranteed that the optimal solution can be found, as the existing charging stations may not be sufficient to meet the reachability constraint. In this case, it was necessary to define the add_cs function, which is identical to the $create\_model$ function but allows for the opening of two additional charging stations in addition to the already active stations.
The final result, obtained by aggregating the solutions of each model, is the opening of 170 charging stations distributed as follows:

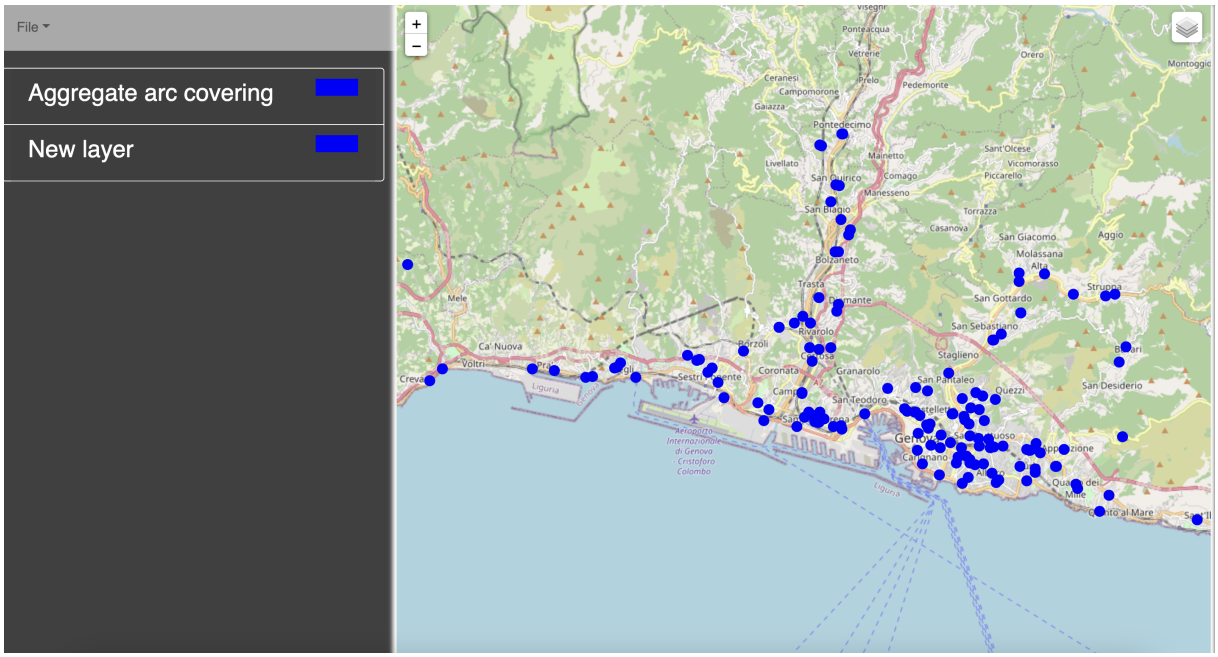


Figure 15:
Result for aggregate arc covering

It is important to consider that the algorithm finds optimal solutions for each pair of zones if and only if there is a movement between the two zones. Consequently, there may be zones that would remain uncovered by charging stations. Specifically, we can consider that the stations resulting from the previous algorithm cover the points in the network if they are located within a maximum distance of 3 km, otherwise the points are not covered. In the following figure, all the green points are covered as they can reach the stations resulting from the aggregate arc covering (shown in red). The blue points are unable to reach the stations.

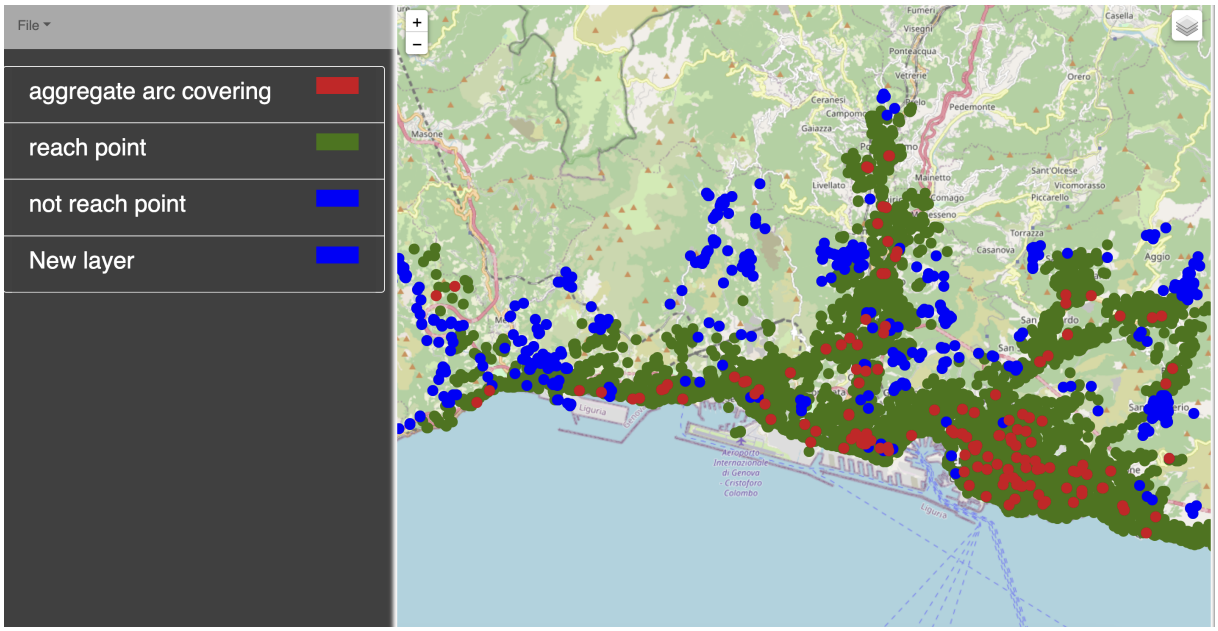


Figure 16:
covered points and not covered points from charging stations

Although there are only 521 such points, we can find a solution to cover them as well. In particular, we apply the graph reachability model among these uncovered nodes, creating a relationship between each uncovered node and another uncovered node only if their distance is less than 3 km. With this new graph obtained, we calculate the minimum dominating set and find the new stations that should be installed to cover these points. The resulting configuration is as follows:

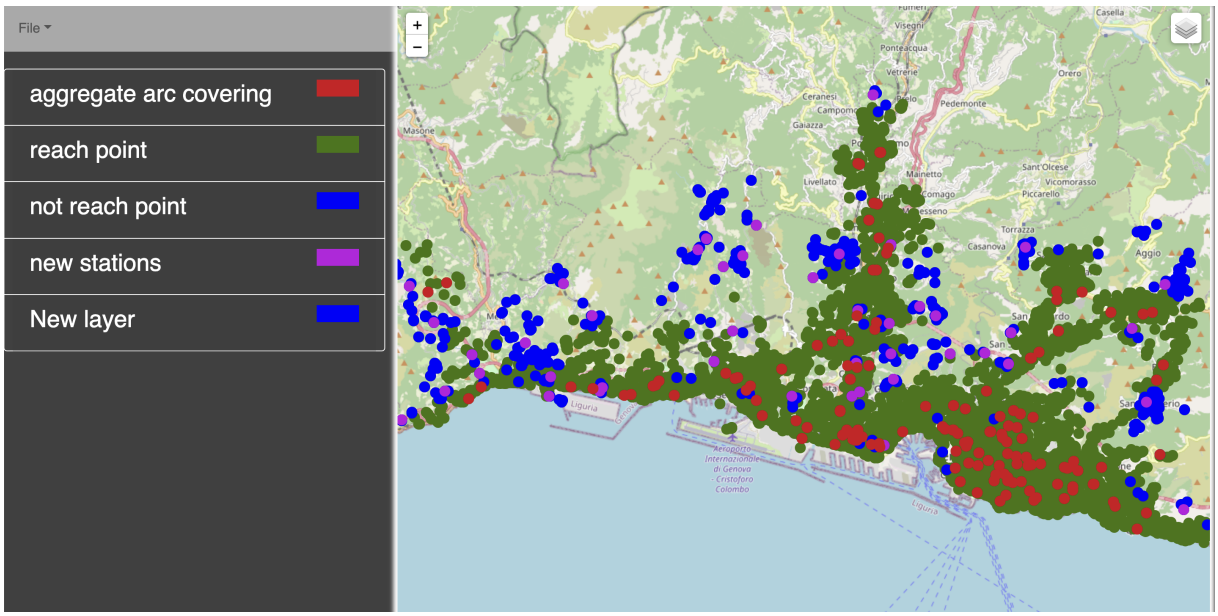


Figure 17:
all covered points with new stations

With this new solution, we can state that the stations activated using the aggregate arc covering model cover the displacement between the zones, while the stations activated subsequently are necessary to ensure even the outermost zones are covered.
The final result is as follows:

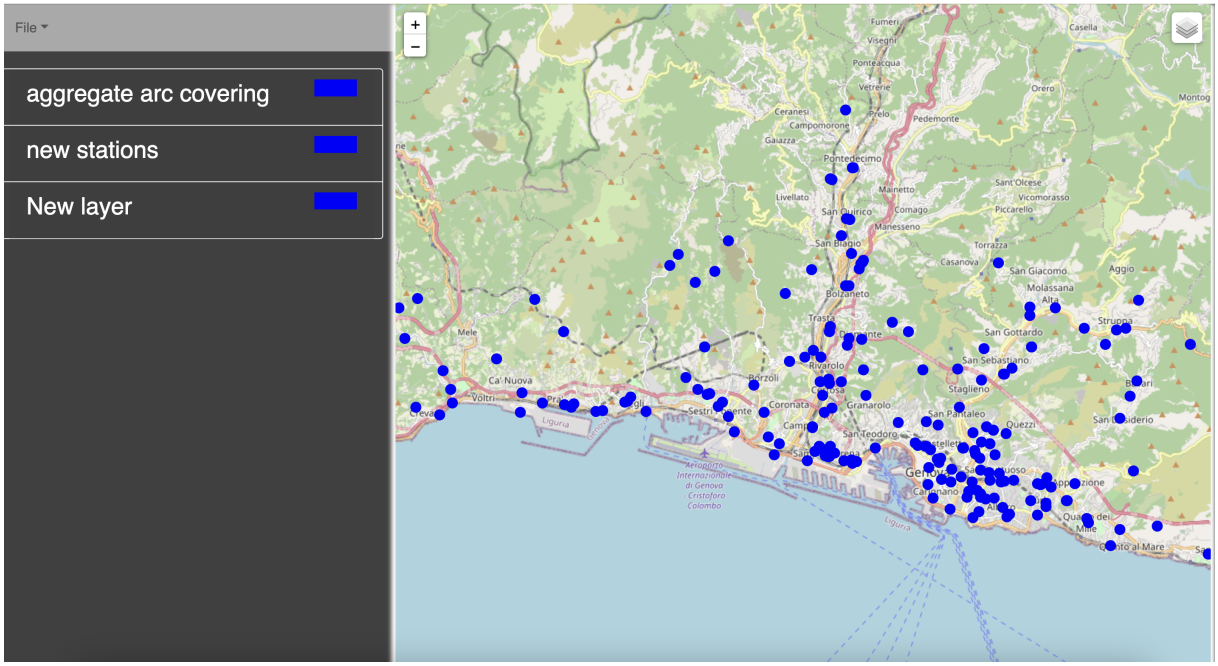
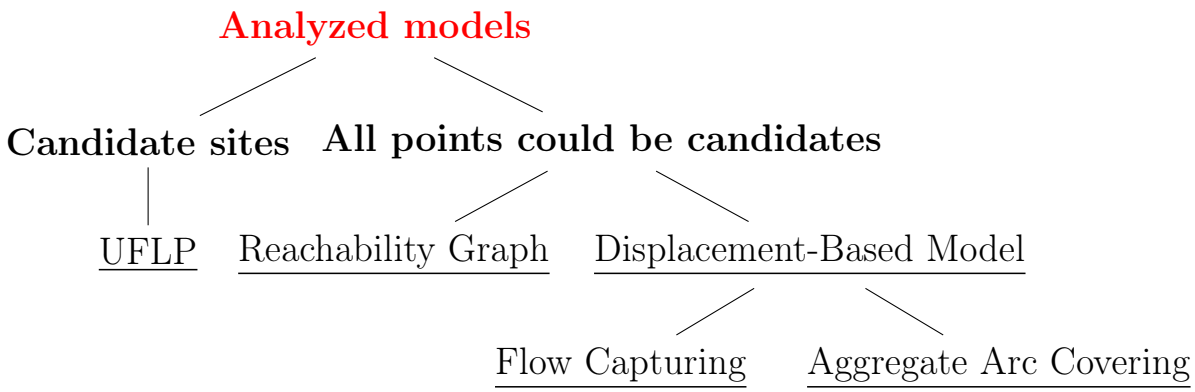


Figure 18:
Total stations to be open

3 Analysis of results

This section summarizes the models analyzed, attempting to analyze the main aspects of each.
The following is a map of the models:



The first two strands are divided into two subcategories: one characterized by the presence of identified candidate sites while for the other any node in the network could be a potential candidate.
Candidate sites occur for those problems where location is of significance; the presence of candidate sites represents additional information regarding, for example, the type of column to be installed. For example, for a service area it is assumed that there are no power grid problems, so one can think of a type of column with fast charging, which can recharge the car within a few minutes. it is useful to study it when there is a restriction on the number of candidate sites. If the candidate sites were all points in a medium-small city, we would have candidate sites on one side and potential clients on the other side; the problem would become very large and complex to solve, not computationally complicated but starts to have criticality when it becomes large. So we introduce models that do not have candidate sites but are based on the concept of reachability or traffic flow, assuming we are installing generic electric columns. This category includes graph and displacement-based reachability models, which in turn are divided into flow capturing and aggregate arc covering. As we have seen, reachability models are basically topological models; they do not consider traffic flows and tend to activate columns in a more distributed manner in the urban area. The model turns out to be computationally easy, it can be useful to give a sizing idea, on how much infrastructure is needed for 1 to k electric columns to be reached from each point in the network.

In displacement-based models we focus on the traffic between zones in Genoa, which also gives us information on how the columns will be distributed so that the paths between zones are covered. With flow capturing we activate electric pillars in the city of Genoa so that each path is covered and the flow is maximized; we are considering the situation in which electric pillars can be activated at any point in the path. it is in fact possible that the pillar is activated at the destination node of the path, so a driver starting from a point in the origin zone should be able to reach the destination point without needing to recharge on the way. However, it is very important to add the aspect on the battery limit for a driver on the road.
In this regard, the last model was analyzed, which, in addition to having the goal of covering all origin-destination paths, also has the goal of reachability. The model is a combination of the flow capturing and the reachability graph model; any node where the column has not been activated can reach the node where the column is installed for that path. Solving the model in its entirety taking into account these features generates a computational problem; linear relaxation cannot be solved either.
In order to overcome the difficulty related to the computational problem, it was thought to solve the arc covering model in an aggregate manner. Aggregate arc covering is a heuristic that combines the two models and builds the solution incrementally.
Also in this model-algorithm, we have an idea of how the columns could be distributed without start from identified allocation points, to make a service that captures the traffic flow and has quality requirements such as reachability of the columns. It turns out to be the most appropriate model for the reality we faced.
In general, each model can be a cue or give an idea of the allocation points. In the following figures we reproduce the results for the reachability graph with $k = 2$ and for the aggregate arc covering so as to have a clear visualization of the distribution of the columns.

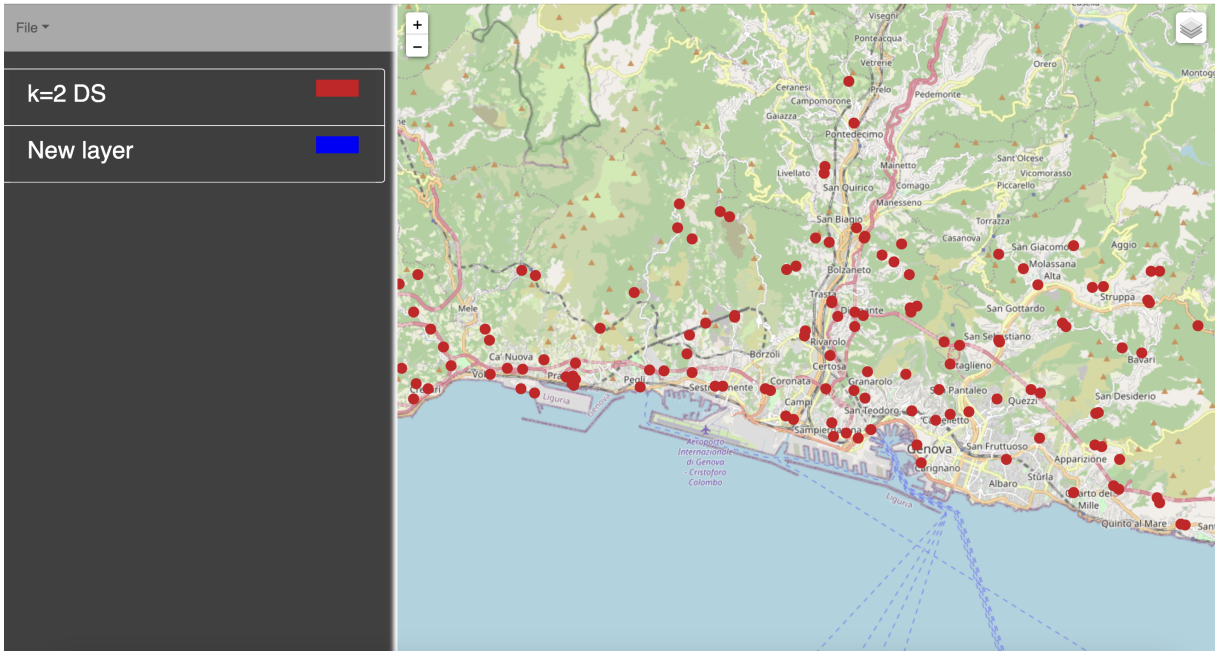


Figure 19: Result for reachability model 2-dominating-set

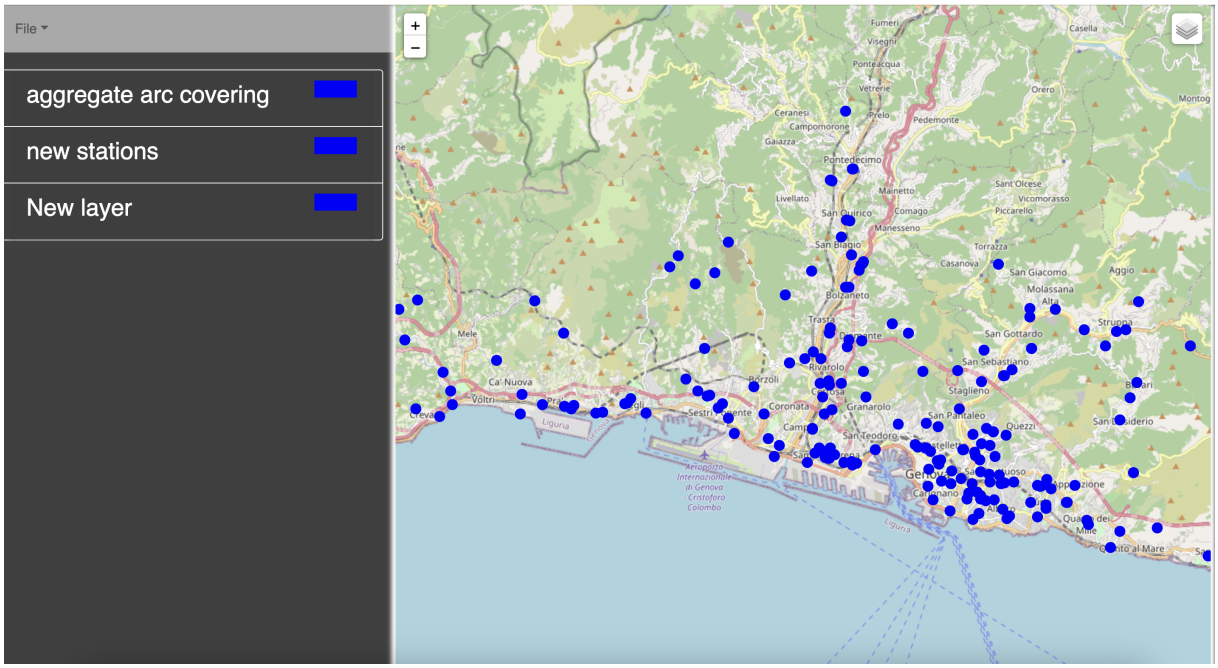


Figure 20: Result for aggregate arc covering

4 Conclusion

In this thesis, we did a general analysis of electric cars, charging types, charging modes and European standards. In addition, we focused on the situation in Italy, looked at the annual growth of electric cars and the growth of charging stations in the territory. After theoretically studying some of the location models of optimal locations for charging stations we focused on the case study, understanding how these models then find application in a real case.
As mentioned before, the obtained results for the localization of charging stations do not take into account the sizing, hence the load capacity of the charging points. It is assumed that they are capable of meeting the daily demand required. At the end, each model can give some kind of information about the distribution of the columns in the urban area of Genova. For the realization of these models we put ourselves from the point of view of the decision maker who in some way grants licenses to build a network of electric columns useful for the user, in particular the installation of the columns should not take place in the central areas of an urban area but be able to consider the suburbs as well to avoid leaving isolated areas.
A municipal decision maker or whomever could use such models, seeing the current results, fixing what already exists to figure out how to evolve the distribution network of inlets.
Hence a future development worthy of attention is the problem of **location-based pricing**, i.e., figuring out the value of a column at a certain location. Location-based pricing is a strategy used to more accurately reflect the costs of doing business in a certain area or to align better with the price sensitivities there. With this pricing method, prices are set differently depending on the locality doing the purchasing. Companies will be interested in putting the columns in a profitable location. With this type of tool it is fairly easy to determine and assign a cost in a column or figure out how much it is worth, because you can figure out how much flow it intercepts or how much flow it covers and how the traffic relates to its neighbors. In general you are able to assign a potential revenue for the company and a potential cost that a municipal operator can exercise.

Given the need for more and more sustainable measures and the obvious growth of electric cars, I think the implementation of this case study can help with any future decisions. In addition, I believe it is imperative that we can keep up with such growth and provide citizens with services that

can also stimulate other citizens to adopt new sustainable strategies, such as purchasing electric cars. Very often we citizens are reluctant to new solutions unless they are accompanied by the necessary services that can satisfy everyone. Finding optimal locations for charging stations is crucial to ensuring a convenient and efficient charging experience for EV drivers, stimulating widespread adoption of electric vehicles, and addressing concerns about vehicle range.