TECHNICAL UNIVERSITY OF CRETE

Department of Electrical and Computer Engineering

A System for Traffic Data Mining and Traffic Monitoring over City Road Networks

Author: Konstantinos Stamatopoulos

Committee: Asst. Prof. Nikos Giatrakos (Supervisor)

Prof. Aikaterini Mania

Prof. Dionyssios Hristopoulos

November 2023

Table of Contents

Acknowledgments....4
Abstract....5
Chapter 1 - Introduction....6
1.1 Importance of road traffic monitoring....6
1.2 Technical difficulties in developing a viable real-time traffic mining framework....7
1.3 Overview of our traffic relationships miner....8
1.4 Thesis outline....9
Chapter 2 - Road network and traffic relationships....10
2.1 Definitions....10
2.1.1 Edges....10
2.1.2 Traffic relationships....11
2.2 Obtaining OSM data....13
2.3 Importing OSM data into PostgreSQL....13
2.4 Visualizing data on QGIS....15
2.5 Edge construction....16
2.5.1 Finding the edge points....17
2.5.2 Creating edges....18
2.5.3 Creating linestring edges....18
2.6 Conclusions & Future work....19
Chapter 3 - Traffic data collection....20
3.1 Data collection tools....20
3.1.1 Selenium....21
3.1.2 Beautiful Soup....21
3.2 Time series data collection....21
3.2.1 WebDriver setup and URL construction....22
3.2.2 Web page inspection....23
3.2.3 Threading....25
3.3 Data management....25
3.4 The time series database....26
3.5 Conclusions & Future work....29
Chapter 4 - Traffic Mining....30
4.1 Clustering algorithms and tools....30
4.1.1 DBSCAN....30
4.1.2 OPTICS....34
4.2 Divisive hierarchical clustering....37
4.3 The 3-step clustering model....39
4.3.1 Step 1....39
4.3.2 Step 2....41
4.3.3 Step 3....43
4.4 Separation of traffic relationships....45
4.5 Experimental results....46
4.6 Display of clustered edges....49
4.6.1 OSMnx....49
4.6.2 Folium....50
4.6.3 The interactive map....51
4.7 Conclusions & Future work....52
Chapter 5 - The application....54
5.1 CustomTkinter....54

5.2 Design & Functionality....56
5.3 Usage....59
5.4 Conclusions & Future Work....61
Chapter 6 - Epilogue....62

Table of Figures

Figure 1. Directed Network Graph....11
Figure 2. Propagate Definition....11
Figure 3. Split Definition....12
Figure 4. Merge Definition....12
Figure 5. Osm Database....14
Figure 6. QGIS interface....16
Figure 7. Edge Points....17
Figure 8. Page Inspection....23
Figure 9. Time series' Plot(1)....28
Figure 10. Time series' Plot(2)....28
Figure 11. Core, Border and Noise Points....32
Figure 12. Density Reachability....33
Figure 13. Reachability Distances and Reachability Plot....36
Figure 14. Divisive Hierarchical Clustering Diagram....38
Figure 15. DBSCAN Haversine Distance Results....43
Figure 16. Markers' Position....47
Figure 17. Algorithms' Accuracy Graph (1)....48
Figure 18. Algorithms' Accuracy Graph (2)....48
Figure 19. The Interactive Map....52
Figure 20. Application's Interface....59
Figure 21. Application's Results....60
Figure 22. Traffic Merge....61
Figure 23. Traffic Split....61
Figure 24. Traffic Propagation....61

Table of Listings

Text 1. Query for Nodes....14
Text 2. Query for Linestrings....15
Text 3. Selenium....22
Text 4. Beautiful Soup....24
Text 5. Step 1 Implementation....40
Text 6. Step 2 Implementation....42
Text 7. Step 3 Implementation....44
Text 8. Separation of Traffic Relationships....45
Text 9. OSMnx....50
Text 10. Folium....51
Text 11. CustomTkinter Interface....56
Text 12. Button Command (part1)....57
Text 13. Button Command (part 2)....58

Acknowledgments

As the time to present my thesis approaches, I would like to express my sincere gratitude to all the individuals who contributed their assistance throughout this challenging but fulfilling journey of mine.

Firstly, I would like to acknowledge and give my warmest thanks to my supervisor, Asst. Professor Nikos Giatrakos, for giving me the opportunity to work on this subject. His guidance, support and expertise carried me through all the stages of completing this thesis and provided me with a valuable mindset on problem solving. I would also like to thank my committee members, Professor Aikaterini Mania and Professor Dionysios Hristopoulos, for their significant contribution to my development as an electrical and computer engineer.

Last but not least, I want to give a special thanks to my family and friends, for their continuous support, understanding and encouragement during all these years. Without them, none of what I've achieved would be possible, and I wouldn’t be the person I am today.

Abstract

In recent years, the issue of traffic congestion has been steadily increasing, making driving an arduous and unfavorable experience for many of us. Despite the expansion of metropolitan areas, an effective solution to the continuously growing number of vehicles in their road networks has yet to be found. In response to this challenge, we have embarked on the design of an application that simulates traffic flow in the city of Chania, serving as a support system for traffic decision-making.

Our approach involves collecting real-time traffic data at intervals of a few minutes for each edge/road segment in the city. Subsequently, we employ a divisive hierarchical clustering model to cluster this data, extracting relationships among them, such as traffic propagation, split, and merge. This process enables us to illustrate the behavior of the road network's traffic flow throughout the day, providing a valuable tool for developing improved transportation systems and techniques to alleviate road overloading.

Chapter 1

Introduction

1.1 Importance of road traffic monitoring

Nowadays, most cities around the world suffer from traffic congestion, and everything indicates that it will continue to worsen, negatively affecting the quality of urban life. Increasing travel times, fuel consumption, and environmental pollution make traffic congestion too 'expensive' from various perspectives. However, authorities have not yet found a solid solution for it. Road works, accidents, poor transportation infrastructure, and traffic signal timing are only some of the reasons that contribute to traffic overload on certain routes. For this reason, we have decided to develop a traffic management decision support system that incorporates real-time data mining techniques to extract traffic patterns.

By collecting and analyzing data on traffic conditions, we can predict the traffic flow on the road network of the city and therefore develop strategies to reduce congestion. These strategies may include adjusting the timing of traffic lights, stationing police officers at key spots for better real-time traffic regulation, or even planning for future transportation infrastructure projects. Hence, road traffic monitoring is an incredibly useful and important tool for improving traffic flow in modern, fast-growing societies.

1.2 Technical difficulties in developing a viable real-time traffic mining framework

At first, we considered using the Google Maps APIs[1] to retrieve information about the speed and congestion of roads in a given area through the Google Maps Roads API. Undoubtedly, it is an easy-to-use API that provides more than we initially asked for. However, the drawback lies in the overwhelming costs associated with it, limiting our ability to scale and fine-tune the framework we intended to develop. For example, the Google Maps API imposes restrictions and bases its pricing policy on the number of queries it serves, prompting us to search for alternatives.

Subsequently, we explored the option of Mapbox[3], where traffic data is natively matched to OpenStreetMap[4], the source of our information for modeling the city road network. The appeal was accessing data directly, outside of Mapbox SDKs and APIs[2]. However, the downside was that the price for this service was even higher than Google's.

Eventually, we discovered that obtaining real-time traffic data under a pricing policy would not make our intended solution neither economically feasible nor scalable from smaller cities to metropolitan areas. That's why we decided to create our own database by 'scraping' traffic data from Google Maps. The idea is to collect information about traffic features (average speed, travel time, etc.) per road network edge and treat these data as time series. This approach allows us to analyze and extract relationships between the road segments of the network.

1.3 Overview of our traffic relationships miner

The main objective of this project is to discern traffic relationships among road segments within the network, encompassing phenomena like traffic propagation, split, and merge. This pursuit is influenced by the framework introduced in [23]. The traffic of a particular edge can be propagated into one of its outgoing edges or be split into multiple ones. Conversely, the traffic of an edge may result from merged traffic from its incoming edges. Detecting these relationships requires defining similarities between the time series of different road segments/edges.

To accomplish this, we adopted an approach initially proposed in [5], leveraging a divisive hierarchical clustering algorithm with three distinct steps. After each step, clusters are further split into sub-clusters.

In Step 1, we utilize shape-based clustering distance, which detects edges whose traffic increases and decreases at the same rate based on the Euclidean distance of their corresponding normalized time series. In Step 2, we employ structure-based clustering distance to identify neighboring edges. Finally, in Step 3, we apply value-based distance clustering to identify time series with similar values, based on their Euclidean distance.

To ensure our implementation is accessible to the general public, we've consolidated its functionality into a single, user-friendly application. We crafted a modern Graphical User Interface (GUI) that allows users to select the month, day, time span, time interval for the time series data, and the traffic relationships they are interested in. Once these selections are made, the program retrieves the desired time series data, clusters them, and displays the results on an interactive map using OpenStreetMap tiles.

Each choice made by the end-user holds significant importance, as the output varies based on different parameter combinations. For instance, traffic relationships on a Saturday will differ from those on a Monday in the same month, and a Friday in August will exhibit distinctions from a Friday in October, especially when the time span refers to rush hours versus the rest of the day. In summary, we aimed to design a minimal and user-friendly application where every option serves a purpose.

1.4 Thesis outline

This section outlines the following chapters' description of this thesis:

- In Chapter 2, we elaborate on the design of the road network and topology of the city under study. In our case this is the city center of Chania for which we construct the database of edges on which we later perform traffic mining.
- In Chapter 3, we develop an algorithm which is used for scraping traffic information from Google Maps' web application and constructs our time series database.
- In Chapter 4, we develop our traffic mining framework that clusters road edges based on their time series through a hybrid 3-step clustering model. We further showcase the effectiveness of our approach against other base line alternatives.
- In Chapter 5, we explain the design of the GUI of our traffic mining application which produces the final result according to the user's preferences.
- Chapter 6 includes conclusive remarks and reviews the accomplishments of this thesis.

Chapter 2

Road network and traffic relationships

In this chapter, we will elucidate the process of obtaining and manipulating the necessary road data from **OpenStreetMap**. Subsequently, we loaded this data into a **PostgreSQL**[6] database, enabling us to model the road network of Chania city as a directed graph. Using the information extracted from this database, we constructed the edges of the network, which will serve as the foundation for our examination of traffic relationships in the subsequent sections.

2.1 Definitions

Before delving into the practical aspects of this thesis, we must first establish the definition of an edge in the road network. Additionally, we will elucidate the underlying logic behind traffic relationships, specifically propagation, split, and merge.

2.1.1 Edges

In our model, we define an edge as the road segment that connects two nodes, which we term as 'edge points'. These edge points are nodes in the road network that connect two or more road segments. To simplify, consider an edge as the portion of the road from one turn to another. When an object enters the edge, it travels the entire length of the edge until it reaches a point where it must decide whether to turn or continue straight. This concept is illustrated in the directed network graph example below.

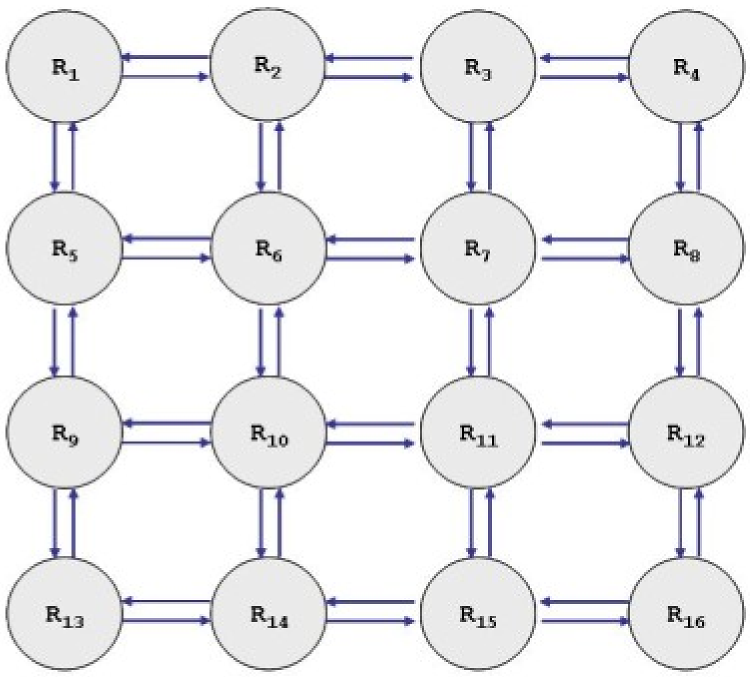


Figure 1. Directed Network Graph[5]

Here the edge points/nodes are depicted as regions (R_1, R_2, ...) and the blue arrows form the directed edges of the hypothetical road network.

2.1.2 Traffic relationships

In order to be able to analyze and examine the traffic flow, we established three different traffic relationships between the edges of the network[5]:

- **traffic propagation**

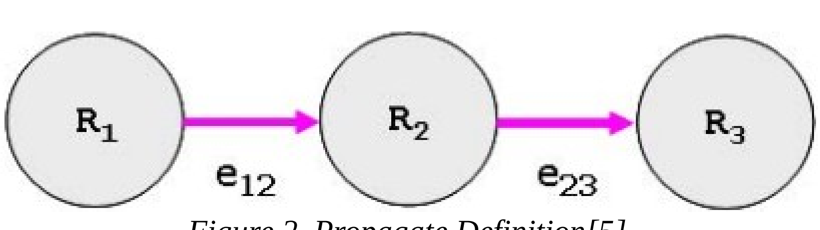


Figure 2. Propagate Definition[5]

edge e_{12} propagates its traffic into edge e_{23}

- **traffic split**

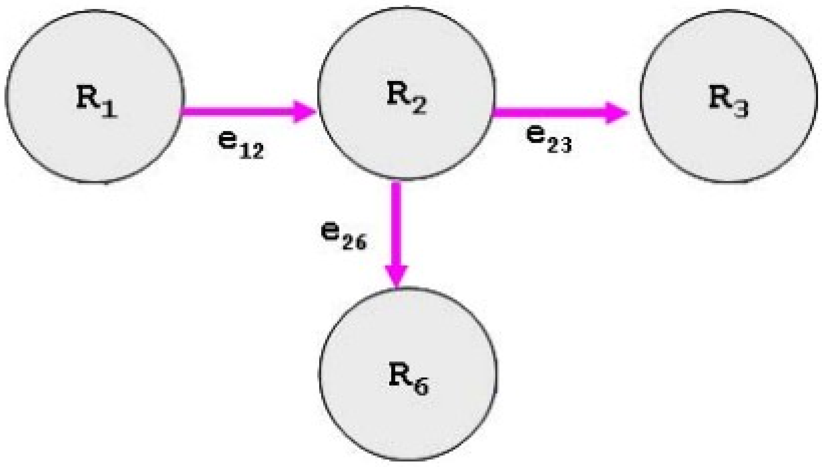


Figure 3. Split Definition[5]

edge e_{12} splits its traffic into edge e_{23} and edge e_{26}

- **traffic merge**

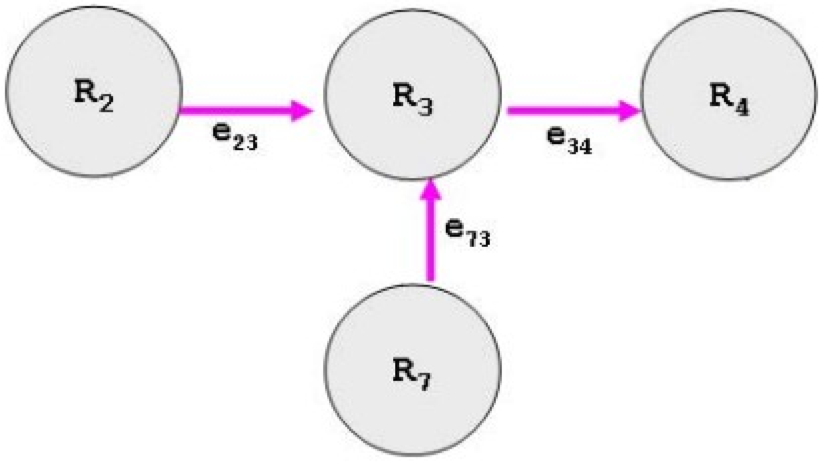


Figure 4. Merge Definition[5]

edge e_{23} and edge e_{73} merge their traffic into edge e_{34}

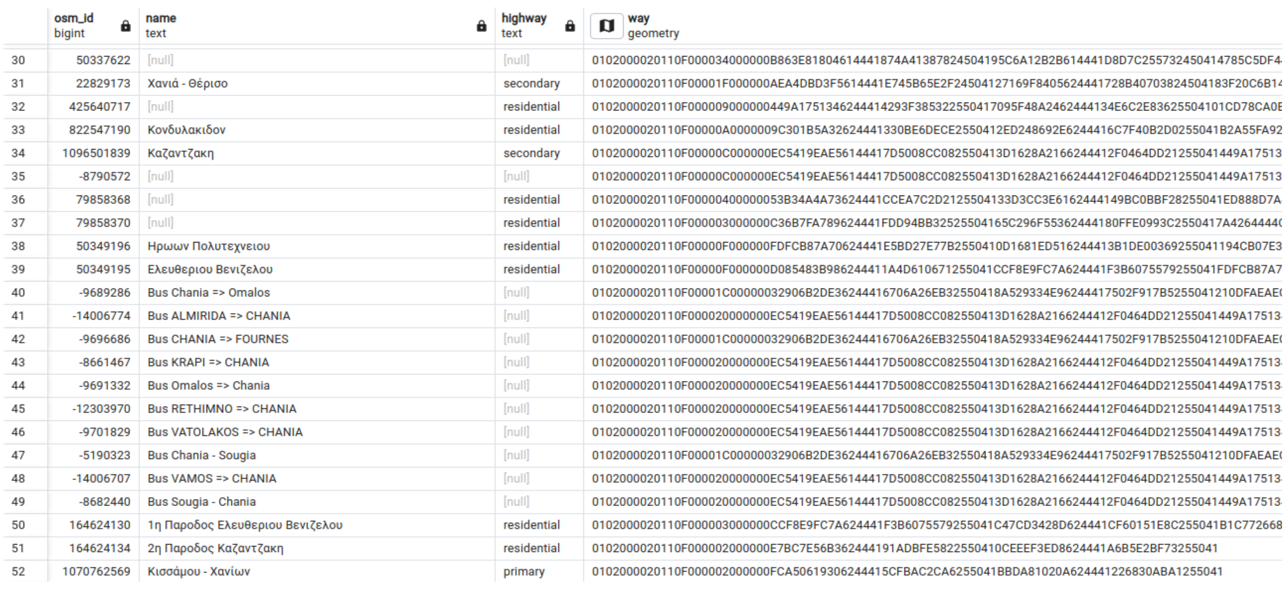
2.2 Obtaining OSM data

The initial step in our project involved designing the road network for the city of Chania. Following research, we identified OpenStreetMap as a valuable resource, as it is an open geographic database offering updated topological data for nearly every city or region worldwide. To obtain the spatial road network data, users can navigate to [4], search for their desired region using the search bar, and then click the export button.

For our specific case, where we required a smaller area as a 'test map' for better comprehension, we manually selected the center of Chania city and exported the data in OpenStreetMap (.osm) files using the same process.

2.3 Importing OSM data into PostgreSQL

Once we obtained the raw OpenStreetMap (OSM) data, our next step was to convert them into a usable and readable format. To achieve this, we downloaded the **osm2pgsql** tool[7], which facilitated the import of the .osm file into a PostgreSQL database through the command line. Following this process, we utilized an interface, specifically **pgAdmin**[8], to store and manipulate the pertinent data extracted from the database through queries.



	osm_id bigint	name text	highway text	way geometry
30	50337622	[null]	[null]	0102000020110F000034000000B863E81804614441874A41387824504195C6A12B2B614441D8D7C255732450414785C5DF4
31	22829173	Χανιά - Θέρισο	secondary	0102000020110F00001F000000AEA4DBD3F5614441E745B65E2F24504127169F8405624441728B40703824504183F20C6B14
32	425640717	[null]	residential	0102000020110F000009000000449A17513462444142 93F38532255041709 5F48A2462444134E6C2E83625504101CD78CA0E
33	822547190	Κονδυλακιδον	residential	0102000020110F00000A0000009C301B5A32624441330BE6DECE2550412ED248692E6244416C7F40B2D0255041B2A55FA92
34	1096501839	Καζαντζακη	secondary	0102000020110F00000C000000EC5419EAE56144417D5008CC082550413D1628A2166244412F0464DD21255041449A17513
35	-8790572	[null]	[null]	0102000020110F00000C000000EC5419EAE56144417D5008CC082550413D1628A2166244412F0464DD21255041449A17513
36	79858368	[null]	residential	0102000020110F00000400000053B34A4A73624441CCEA7C2D2125504133D3CC3E6162444149BC0BBF28255041ED888D7A
37	79858370	[null]	residential	0102000020110F000003000000C36B7FA789624441FDD94BB32525504165C296F55362444180FFE0993C2550417A4264444
38	50349196	Ηρωων Πολυτεχνειου	residential	0102000020110F00000F000000FDFCB87A70624441E5BD27E77B2550410D1681ED516244413B1DE00369255041194CB07E3
39	50349195	Ελευθεριου Βενιζελου	residential	0102000020110F00000F000000D085483B986244411A4D610671255041CCF8E9FC7A624441F3B6075579255041FDFCB87A7
40	-9689286	Bus Chania => Omalos	[null]	0102000020110F00001C00000032906B2DE36244416706A26EB32550418A529334E96244417502F917B5255041210DFAEAE
41	-14006774	Bus ALMIRIDA => CHANIA	[null]	0102000020110F000020000000EC5419EAE56144417D5008CC082550413D1628A2166244412F0464DD21255041449A17513
42	-9696686	Bus CHANIA => FOURNES	[null]	0102000020110F00001C00000032906B2DE36244416706A26EB32550418A529334E96244417502F917B5255041210DFAEAE
43	-8661467	Bus KRAPI => CHANIA	[null]	0102000020110F000020000000EC5419EAE56144417D5008CC082550413D1628A2166244412F0464DD21255041449A17513
44	-9691332	Bus Omalos => Chania	[null]	0102000020110F000020000000EC5419EAE56144417D5008CC082550413D1628A2166244412F0464DD21255041449A17513
45	-12303970	Bus RETHIMNO => CHANIA	[null]	0102000020110F000020000000EC5419EAE56144417D5008CC082550413D1628A2166244412F0464DD21255041449A17513
46	-9701829	Bus VATOLAKOS => CHANIA	[null]	0102000020110F000020000000EC5419EAE56144417D5008CC082550413D1628A2166244412F0464DD21255041449A17513
47	-5190323	Bus Chania - Sougia	[null]	0102000020110F00001C00000032906B2DE36244416706A26EB32550418A529334E96244417502F917B5255041210DFAEAE
48	-14006707	Bus VAMOS => CHANIA	[null]	0102000020110F000020000000EC5419EAE56144417D5008CC082550413D1628A2166244412F0464DD21255041449A17513
49	-8682440	Bus Sougia - Chania	[null]	0102000020110F000020000000EC5419EAE56144417D5008CC082550413D1628A2166244412F0464DD21255041449A17513
50	164624130	1η Παροδος Ελευθεριου Βενιζελου	residential	0102000020110F000003000000CCF8E9FC7A624441F3B6075579255041C47CD3428D624441CF60151E8C255041B1C772668
51	164624134	2η Παροδος Καζαντζακη	residential	0102000020110F000002000000E7BC7E56B362444191ADBFE5822550410CEEEF3ED8624441A6B5E2BF73255041
52	1070762569	Κισσάμου - Χανίων	primary	0102000020110F000002000000FCA50619306244415CFBAC2CA6255041BBDA81020A624441226830ABA1255041

Figure 5. Osm Database

The essential spatial data for modeling our road network is found in the *planet_osm_line* table. This table encompasses comprehensive details about the network, and our specific focus was on three key columns: *osm_id*, *highway*, and *way*. The *osm_id* column contains the unique identifier for each node on the map, the *highway* column specifies the type of the road (e.g., living_street, footway, residential), and the *way* column provides geometry information about the nodes, including their coordinates and how they are interconnected.

By utilizing the appropriate queries, we successfully extracted either all the nodes of the map as points or their linked format, known as linestrings, where nodes of the same id are connected in a line. However, in both cases, we had to establish certain criteria based on the 'highway' column to exclude routes that are not accessible by cars. The queries we employed for this purpose are as follows:

- **Points/nodes**

```sql
SELECT osm_id, st_asText(((ST_Dumppoints (st_transform(way, 4326))).geom))
FROM planet_osm_line
WHERE highway!='NULL' AND highway!='unclassified' AND highway!='footway' AND
highway!='steps' AND highway!='service' AND highway!='path' AND highway!
='track' AND highway!='living_street' AND highway!='pedestrian';
```

Text 1. Query for Nodes

- **Linestrings**

```
SELECT osm_id, st_asText((st_transform(way, 4326))) as linestrings
FROM planet_osm_line
WHERE highway!='NULL' AND highway!='unclassified' AND highway!='footway' AND
highway!='steps' AND highway!='service' AND highway!='path' AND highway!
='track' AND highway!='living_street' AND highway!='pedestrian';
```

Text 2. Query for Linestrings

To obtain the longitude and latitude format, we needed to convert the current Reference System to the World Geodetic System, identified by SRID 4326. The PostGIS[9] function *st_transform(way, 4326)* returns a geometric variant of the 'way' column, and the *st_asText()* function provides the Well-Known Text (WKT) representation of that geometry without SRID metadata. This allowed us to obtain the data in linestring format. To extract the points along every linestring and their individual coordinates, we also used the *ST_Dumppoints()* function.

2.4 Visualizing data on QGIS

Considering the enormous size of data in these databases, we sought a way to visualize them for better understanding before making our selection. Initially, we created a PostGIS extension on our PostgreSQL database. Subsequently, we downloaded a Geographic Information System application, specifically the **QGIS**[10] application. QGIS allowed us to visualize our data on an OpenStreetMap tile and verify querying results, such as the selection of data types to be excluded in the 'highway' column, using its Query Builder module. The interface of the application is depicted in the following image, with the green lines on the map representing the data from the selected table in the PostgreSQL database.

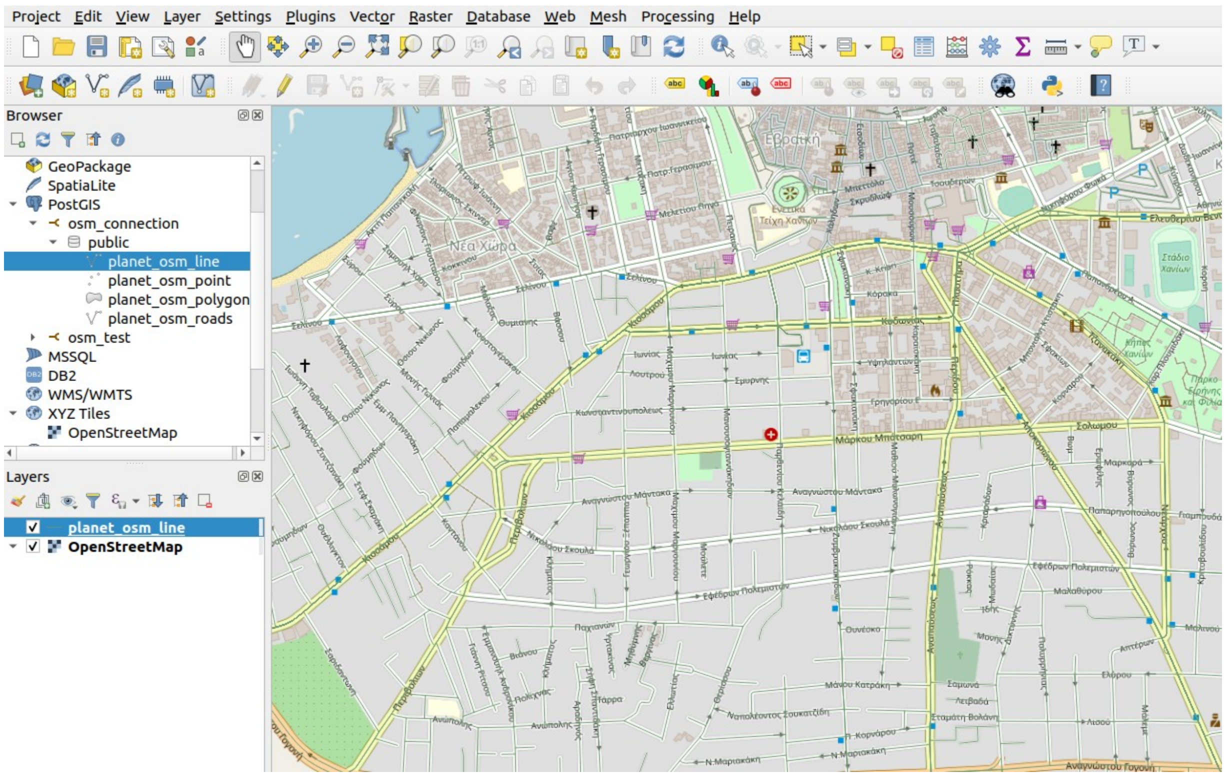


Figure 6. QGIS interface

2.5 Edge construction

At this stage, we had two CSV files. The first contained all the points/nodes on the map with their IDs, and the second contained linestrings, which are the linked form of those points/nodes, along with their unique IDs. Our primary task was to identify and store the points that serve as edge points, namely those where an edge starts or ends. This step was crucial for the subsequent process of edge identification.

2.5.1 Finding the edge points

To read and handle the data in those CSV files, we utilized the **pandas** library[11] in Python, an incredibly powerful and user-friendly open-source tool for data analysis and manipulation. With the help of some built-in functions, particularly those for identifying duplicates, we managed to extract the edge points from the first file. Simply put, if a point occurs more than once with different IDs in that file, it indicates an edge point, as mentioned in Section 2.1.1, signifying a connection between two or more road segments/edges. To validate our approach, we stored these edge points in another file and visualized them as markers using the **gmplot** library[12] in Python.

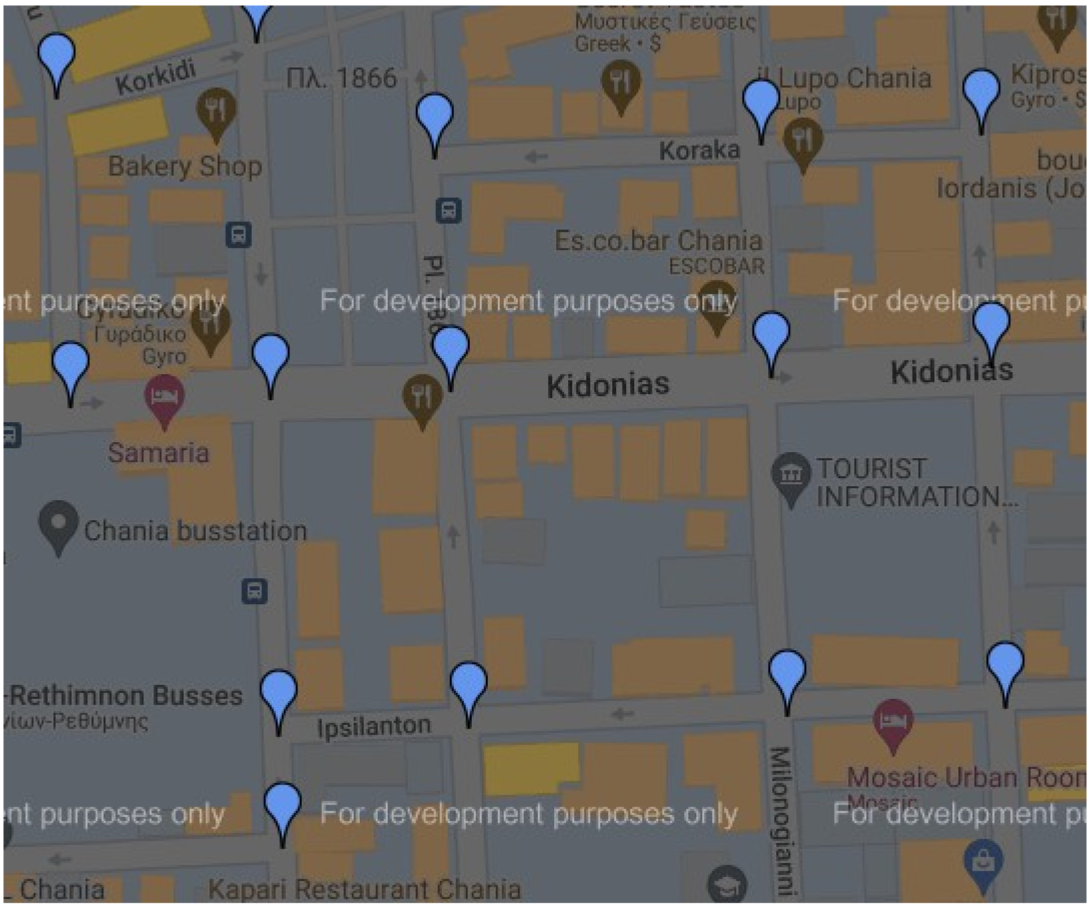


Figure 7. Edge Points

2.5.2 Creating edges

Having identified the edge points of the city’s road network, our next step was to create its edges. One way to achieve this is by taking each edge point from the new file created in Section 2.5.1 and sequentially searching for it in the list of nodes obtained from the database. Subsequently, if we find such a node, we check the ID of the next node in the file. If the IDs are the same, it indicates that we have an edge starting from the current (edge) point and ending at the next point in the list. Otherwise, we have an edge starting from the previous point and ending at the current (edge) point. While this method may not be optimal in terms of time and resources, it is practical. We acknowledge the potential for improvement in this aspect of our edge extraction technique as future work.

To validate the created edges, we executed the script for data mining, which we will analyze later. Unexpectedly, despite the correctness of our logic, we encountered an issue due to the short length of edges in a city like Chania. This limitation prevented us from obtaining accurate measurements for average speed and travel time in our time series database. Google Maps' minimum travel time is 1 minute, and it does not provide information at the granularity of seconds, which was essential for our needs. Consequently, we considered assigning more than one actual edge of the network to a single edge ID. This approach aimed to enhance differentiation between edges for improved accuracy in our data representation.

2.5.3 Creating linestring edges

Due to the mentioned limitation at the end of Section 2.5.2, we decided to incorporate the second file from the osm database, which contains linestrings of the road network. A linestring is essentially a sequence of nodes, positioned one after the other, that when connected, forms a line (a sequence of edges) on the map. Following a similar logic, we extracted each edge point from the file created in the initial step and sequentially searched for it in the list of linestrings. If we found an edge point in any set of points (linestring), we considered the entire linestring as an edge. This approach allowed us to construct our database of hyper-edges, which will be utilized for traffic data mining.

2.6 Conclusions & Future work

One of the most challenging aspects of this thesis was finding free and updated spatial data to model the city's road network. Initially, we considered using Google Maps and its APIs, given its reputation as a constantly evolving service and the distinction of being a premier mapping and routing provider. However, in today’s digital landscape, data is often as valuable as currency, playing a pivotal role in business success. While Google occasionally provides data for application development, developing a fully-fledged application for real-world business typically involves fees for its services. Nevertheless, our goal was to create an application that would be free for both the developer and the user. To achieve this, we explored and developed our own alternative data harvesting solution.

Fortunately, after conducting research, we discovered OpenStreetMap, which supplied us with all the geographical data necessary for the project. Additionally, we were pleasantly surprised to find that all coordinate information is seamlessly compatible with Google Maps. This alignment will prove particularly beneficial in the upcoming data mining discussions.

As a part of future work, we recommend considering the use of original edges if, at some point, Google Maps starts providing travel time for short-length road segments with granularity in seconds. However, developers should be aware that this approach will require corresponding resources for the automated process of data scraping, as the number of real edges will be significantly larger than the hypothetical ones used in this implementation.

Chapter 3

Traffic data collection

In this chapter, we will present the tools we used to scrape the web application of **Google Maps** in order to create our own database of time series for the edges of the road network. Additionally, we will explain the techniques employed to obtain these measurements on a regular basis.

3.1 Data collection tools

It is evident that web scraping becomes essential in data science when there is no alternative source to acquire the necessary data for analysis. In our scenario, we have already generated a file containing the edges. Now, our objective is to gather real-time traffic data for each of these edges at regular intervals, enabling the creation of a time series database for subsequent clustering. Achieving this requires an automated process wherein we retrieve information such as *distance, travel time,* and *delay* from the source to the destination point/node of each edge.

This specific data, among others, can be obtained from services like Google Maps, which we consider to be the most up-to-date and accurate for the information we seek. After researching various scraping methods, we opted for a combination of **Selenium**[13] and **Beautiful Soup**[14] in Python to extract and process this data efficiently.

3.1.1 Selenium

For our automated data mining routine, we selected Selenium due to its status as a free, open-source framework widely employed for automated scraping across various platforms and browsers. It goes beyond being merely a tool, serving as a suite of software that facilitates the development of automation scripts for web applications by offering powerful built-in functions. In essence, all that is required is to download the updated driver for the preferred browser and then set the path for Selenium's ‘WebDriver’ tool. This enables direct communication with the browser, allowing us to focus on extracting the desired data from the rendered HTML document generated by each web page.

3.1.2 Beautiful Soup

Now that we have addressed the automation process, our focus is on reading and extracting information about travel time, distance, and delay for each iteration in the file of edges. Whenever a Google Maps page loads, it generates an HTML document that encompasses the data essential for scraping and creating our time series database. To achieve this, we opted for Beautiful Soup, a Python library specifically designed for parsing HTML information from web pages. It allows us to isolate the classes and titles of interest through specific commands, facilitating the extraction of the required data.

3.2 Time series data collection

Several key factors required careful consideration before proceeding with the modeling of our data mining algorithm. These factors include the correct URL format, the identification and selection of relevant parts within the HTML of the web page, and the establishment of a mechanism to sustain this process continuously for the desired duration. In the following sections, we will delve into the challenges we encountered in addressing these aspects.

3.2.1 WebDriver setup and URL construction

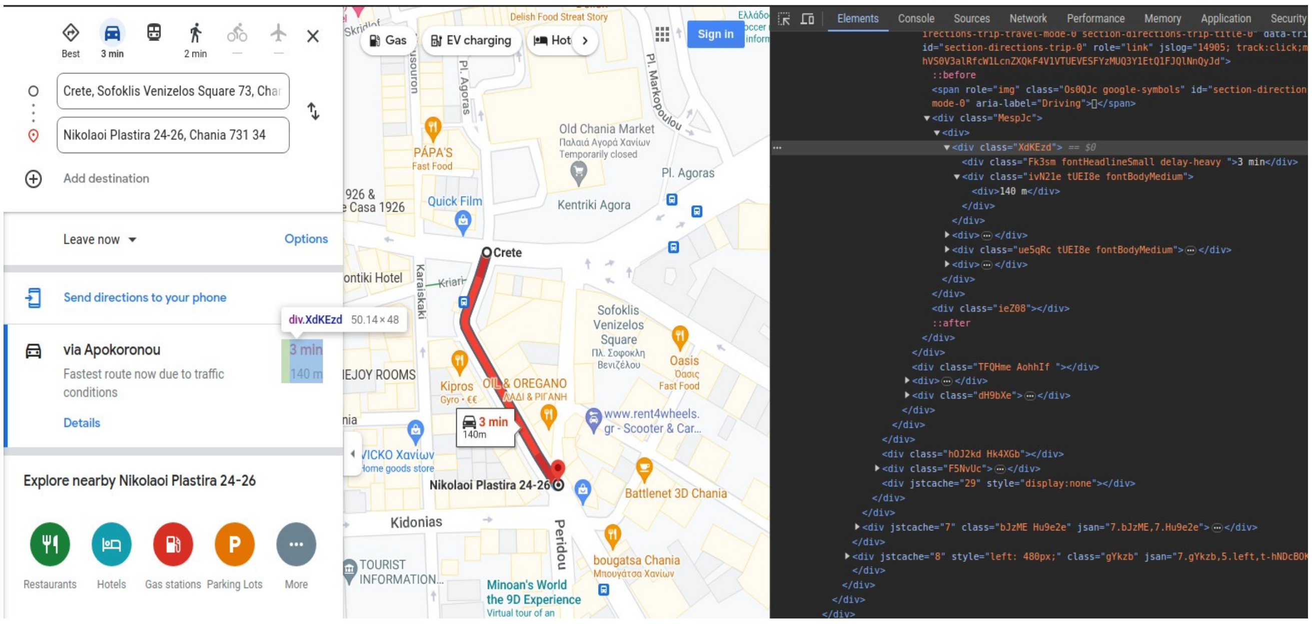
For our automated process with Selenium, we opted for **Google Chrome** as the browser of choice. To set up Selenium's WebDriver tool, we downloaded the driver for the current version of the browser, allowing seamless communication. Additionally, to minimize reCaptcha verifications, we employed a fake user agent. Constructing the URL was the next step, using the pair of coordinates for each edge to retrieve the necessary information in drive mode. It's worth noting that Selenium's capabilities enabled us to effortlessly handle the "cookie consent" page that appears at the beginning of each session by automatically locating and clicking the appropriate button. The following code snippet illustrates the implementation of these steps:

```python
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.chrome.options import Options
from selenium.webdriver.chrome.service import Service
from fake_useragent import UserAgent

service =
Service('/home/user/Downloads/chromedriver-linux64/chromedriver')
user_agent = UserAgent().random
options = Options().add_argument(f'--user-agent={user_agent}')
driver = webdriver.Chrome(options=options, service=service)

url = f'https://www.google.com/maps/dir/{src_lng},{src_lat}/{dest_lng},
                                   {dest_lat}/data=!4m2!4m1!3e0?hl=en'
driver.get(url)

if first_time == True:
      time.sleep(5)
      button = driver.find_element(By.TAG_NAME, 'button')
      button.click()
      first_time = False
```

Text 3. Selenium

3.2.2 Web page inspection

Considering that we set everything up about the Selenium suite, we had to find which classes, titles and elements to isolate using the HTML parser Beautiful Soup. This can be achieved by inspecting the part of the page that has the information we are looking for. For example, if we search in Google Maps the directions to go from one place to another and then navigate to the area of the page that has the information you need, then right click and click “Inspect”, we get something like the following image:



```
irections-trip-travel-mode-0 section-directions-trip-title-0" data-tri
id="section-directions-trip-0" role="link" jslog="14905; track:click;m
hVS0V3alRfcW1LcnZXQkF4V1VTUEVESFYzMUQ3Y1EtQ1FJQlNnQyJd">
  ::before
  <span role="img" class="Os0QJc google-symbols" id="section-direction
  mode-0" aria-label="Driving"></span>
 ▼<div class="MespJc">
   ▼<div>
     ▼<div class="XdKEzd"> == $0
         <div class="Fk3sm fontHeadlineSmall delay-heavy ">3 min</div>
       ▼<div class="ivN2le tUEI8e fontBodyMedium">
           <div>140 m</div>
         </div>
       </div>
     ▶<div>⋯</div>
     ▶<div class="ue5qRc tUEI8e fontBodyMedium">⋯</div>
     ▶<div>⋯</div>
     </div>
   </div>
   <div class="ieZ08"></div>
   ::after
  </div>
 </div>
 <div class="TFQHme AohhIf "></div>
▶<div>⋯</div>
▶<div class="dH9bXe">⋯</div>
</div>
</div>
</div>
<div class="hOJ2kd Hk4XGb"></div>
▶<div class="F5NvUc">⋯</div>
<div jstcache="29" style="display:none"></div>
</div>
</div>
▶<div jstcache="7" class="bJzME Hu9e2e" jsan="7.bJzME,7.Hu9e2e">⋯</div>
</div>
▶<div jstcache="8" style="left: 480px;" class="gYkzb" jsan="7.gYkzb,5.left,t-hNDcBOK
</div>
</div>
</div>
</div>
```

Figure 8. Page Inspection

Fortunately, the structure of the HTML document remains consistent across every iteration, allowing us to scrape the required data using Beautiful Soup's built-in functions. Subsequently, we can store this data to construct our time series database. The following code snippet represents our implementation.

```
from bs4 import BeautifulSoup

result = None

while result == None :
     soup = BeautifulSoup(driver.page_source, 'lxml')
     result = soup.find('div', class_='XdKEzd')

delay_div = result.find('div')
if len(delay_div['class']) > 2:
     delay = delay_div['class'][2]
else:
     delay = 'no-data'

travel_time = result.find('div').text.replace(' min', '')
distance  =  soup.find('div',
                     class_='ivN21etUEI8efontBodyMedium').text

speed = float(distance)/(int(travel_time)*60)
if delay == 'delay-heavy':
     speed /= 2
if delay == 'delay-medium':
     speed /= 1.5

if file_exist == False:
     edges.append([src_lng, src_lat, dest_lng, dest_lat, speed])
else:
     edges.append(speed)
```

Text 4. Beautiful Soup

As the web page is fully loaded by our driver, we parse the page’s source file using lxml’s HTML parser. Firstly, we search for the class named *“XdKEzd”* and then for the next div inside that class, which has our delay and travel time information. The class title of that next div contains the delay data, if there is such, and the class’s text is our travel time data. Then we search inside the class *“XdKEzd”* for the class named *“ivN21e tUEI8e fontBodyMedium”*, whose text is our distance data. Finally, we compute and store the recorded speed in our database.

3.2.3 Threading

Our objective was to devise a traffic data mining algorithm capable of running continuously for an extended duration, collecting data at regular intervals over a predetermined period. This aligns with the definition of a time series. To achieve this, we employed the **threading** library in Python, with each thread initiating a new automated session. By manually executing this script at the desired time, it autonomously initiates a new session after the specified interval has elapsed.

At this point, it's important to note that, on a single script execution for our file of edges, the optimal frequency we achieved was one measurement every 10 minutes. This was achieved with two different computers working in parallel for the same purpose. Consequently, each computer could initiate a new session every 20 minutes to avoid overloads and potential failures. In essence, the more edge IDs present, the greater the resources required to collect real-time data within short time intervals.

3.3 Data management

As stated in Section 3.1, the primary objective of the data mining algorithm we designed was to construct a dedicated database of time series for each edge ID. In practical terms, we calculate the speed, defined as the distance from the source to the destination node of the edge divided by its travel time. To enhance the differentiation between measurements, we opted to incorporate the factor of delay into the equation.

Prior to executing the script, it's necessary to predetermine its duration by adjusting certain parameters, such as the total number of sessions and the waiting time before each subsequent session begins. Once the execution is complete, a file is generated in the format: *"Day(yyyy-mm-dd).csv"*. However, it's important to note that data scraping might not occur throughout the entire day. The process often requires manual oversight for unexpected blocks that necessitate a manual refresh or for reCaptcha pop-ups that require verification.

Given this, if we have one file with data for a specific day and time span, and another file with data for the same day and month but a different time span, we need to merge these files to create a comprehensive time series database for that specific day of the month.

3.4 The time series database

Before we proceed to the description of our time series database, we have to define what we mean as time series and traffic of an edge. Assuming that Google Maps works as a sensor *s* for us on a particular road segment/edge, the time series of that sensor is a sequence of the speeds recorded during a specific time period *[t_s, t_e]* and are given by the formula:

$$TS_s = \{v_i, t_i\},\ t_s \leq t_i \leq t_e.$$

where v_i is the speed recorded by the sensor during the time period *[t_i, t_i+Δt)* and *Δt* is the transmission rate of the sensor.

An example of the time series of a sensor *s* with a transmission rate of 10 minutes would be: *TS_s = {(30, 9:00), (15, 9:10), (20, 9:20), ...}* and that is the representation of the traffic in the corresponding edge, recorded by its sensor. Therefore, the traffic of a road network is a set of time series that describe the traffic of its edges during a specific time period *[t_s, t_e]*:

$$TS = \{TS_s, s \in S\}$$

Having grasped these definitions, we successfully crafted our own time series database using Selenium and Beautiful Soup. The table below serves as an example of a dataset for a random day spanning from 9:00 to 17:30.

index	**src_lng**	**src_lat**	**dest_lng**	**dest_lat**	**9:00**	**9:10**	**...**	**17:30**
0	35.4795	23.9993	35.5042	24.0066	6.67	3.33	…	10
1	35.5030	24.0057	35.5026	24.0083	4.17	4.17	…	4.17
2	35.5063	24.0036	35.5042	24.0066	4.17	4.17	…	6.25
.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.
843	35.5182	24.0398	35.5167	24.0399	7.5	5	…	7.5

The *index* column contains the id of the hyper-edges of our road network. The *src_lng* and *src_lat* columns contain the longitude and latitude coordinates of each edge's starting/source point, respectively. Similarly, the *dest_lng* and *dest_lat* columns contain information about longitude and latitude coordinates of each edge's ending/destination point. The rest of the columns are named with the time each session started and contain the speed recorded, at this specific time, on every edge.

If we plot the time series of the edges, we can make useful observations, both for the specific day being plotted and for each individual edge itself, even before clustering them. For instance, when comparing the time series of e_0 and e_{57} for the same time span on different days of the same month, we obtain the following results:

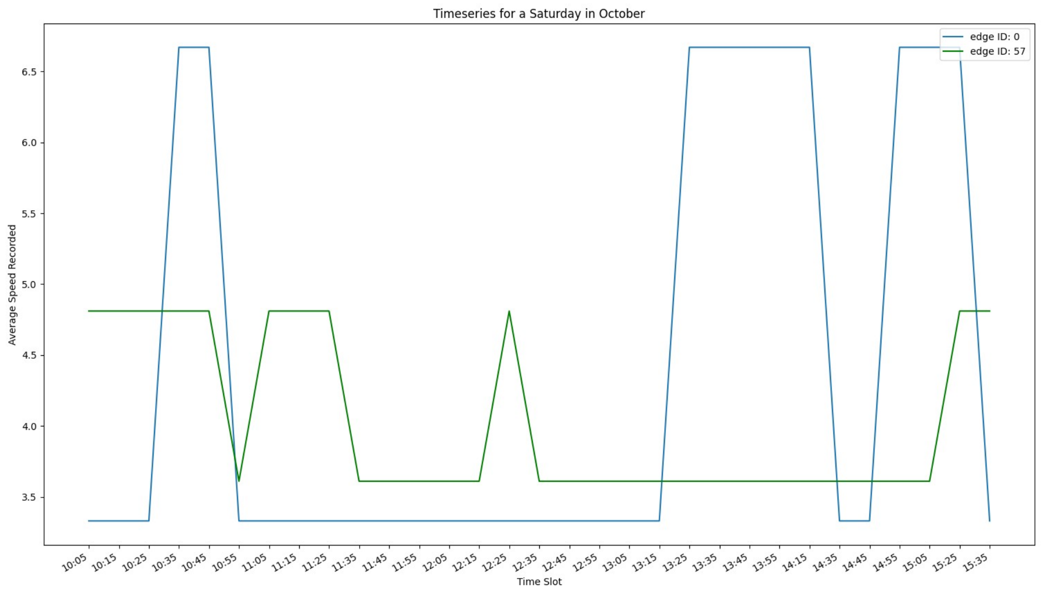


Figure 9. Time series' Plot(1)

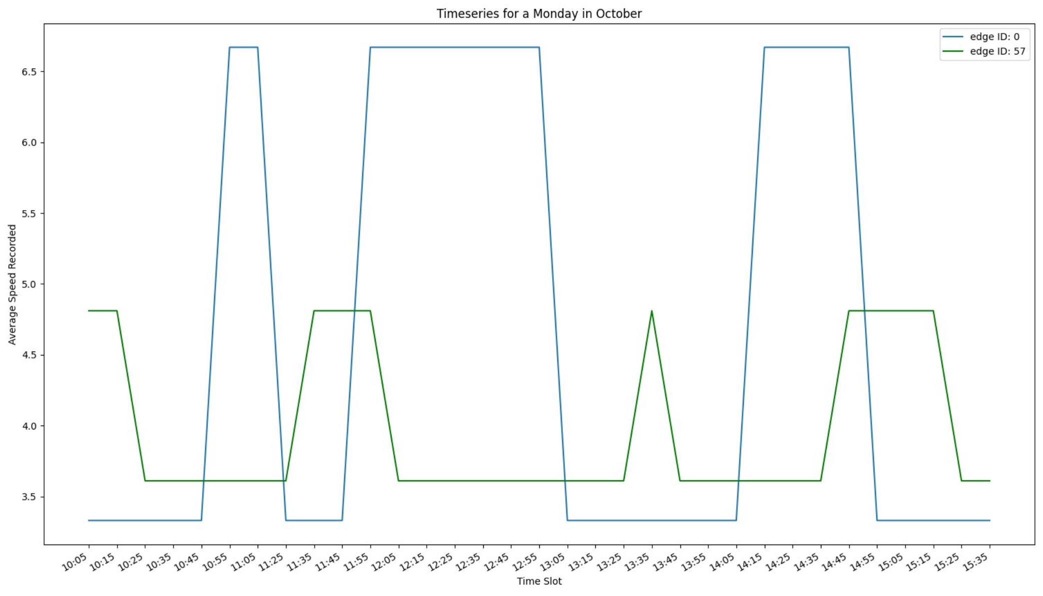


Figure 10. Time series' Plot(2)

3.5 Conclusions & Future work

Real-time data collection is undoubtedly essential for projects of this kind, given its influential role in shaping the final outcome. The combination of Selenium and Beautiful Soup proved to be ideal for us, and for several compelling reasons. Primarily, Selenium provides the capability to visualize precisely what you are attempting to scrape. This means you can verify the logic behind constructing edges and ensure the accurate extraction of data for your database through Beautiful Soup. Moreover, Selenium allows for effective handling of issues that may arise on the specific web application, such as the "cookie consent" page at the start of each session, webpage blocks, and verification pop-ups. Importantly, these challenges can be addressed without disrupting the automated process or losing previously collected data. The synergy of these factors, along with the excellent performance of Beautiful Soup's built-in functions, validates our choice for this approach.

Nevertheless, there are conceivable improvements that could enhance this script in the future. For instance, after every browser update, the need to download the new driver and replace it with the older one for proper functioning of Selenium’s WebDriver module can be a cumbersome task. Currently, the web driver manager tool provided for this purpose somehow does not work. Another inconvenience lies in the manual treatment of blocks and pop-ups, requiring developer supervision and making the process time-consuming. In conclusion, if these issues could be addressed, coupled with a high-speed internet connection and adequate resources, this algorithm has the potential to evolve into a powerful tool for constructing time series databases.

Chapter 4

Traffic Mining

In this chapter, we will elucidate our custom implementation of a divisive hierarchical clustering model. Each of the three steps of the model will be thoroughly analyzed, accompanied by an introduction to the Python tools that aided us in clustering the database of edges' time series created earlier. The outcomes of our algorithm will be discussed, and, in the final sections, we will demonstrate how to visualize the clustered data on an interactive map.

4.1 Clustering algorithms and tools

Python offers an extensive array of libraries and packages for data analysis and machine learning, facilitating the implementation of clustering algorithms tailored to specific needs. In our case, the combination of **DBSCAN**[15] and **OPTICS**[16] clustering algorithms emerged as the optimal choice. These algorithms utilize distance calculating functions that align with the requirements of constructing our three-level clustering model. In the following sections, we will elaborate on the workings of these algorithms and elucidate how we utilized them to achieve our objectives.

4.1.1 DBSCAN

Clustering analysis is fundamentally an unsupervised learning method that partitions data into specific groups. Density-based spatial clustering of applications with noise (DBSCAN)[15] is a data clustering algorithm commonly employed in machine learning to distinguish regions of high point density from those of low point density. It groups points that are close to each other based on a distance measurement function and designates points in low-density regions as noise.

The DBSCAN algorithm uses two parameters:

- **epsilon (eps)**: a distance measure that specifies how close the points have to be, so as to be considered neighbors and therefore part of a cluster.
- **min_samples**: the minimum number of points (neighbors) to form a dense region.

In order to choose these parameters, we need to have a basic knowledge about the dataset that will be used and then perform a parameter estimation, as follows:

- In general, if the value of epsilon is chosen too large, clusters may merge, and the majority of data points will be grouped into the same clusters. Conversely, if it is chosen too small, a significant portion of data points may be considered as outliers. Experimentation with this parameter, both increasing and decreasing it, is necessary to find the most suitable value for our data. However, smaller values of epsilon are typically preferred.
- The minimum number of min_samples can be determined from the number of dimensions (D) in the dataset, with the requirement min_samples >= D + 1. Setting min_samples = 1 wouldn't be meaningful, as it would result in every data point being considered a cluster on its own. Typically, the larger the dataset or the noise within it, the larger the value chosen for the min_samples parameter. Nevertheless, experimentation is crucial to identifying the most suitable value.

Given epsilon and min_samples, we can categorize the data points in the following three types:

- **Core Point**: a point that has at least min_samples within epsilon range.
- **Border/Non-core Point**: a point that has fewer than min_samples within epsilon range, but is in the neighborhood of a core point.
- **Noise/Outlier**: a point that is neither a core nor a border.

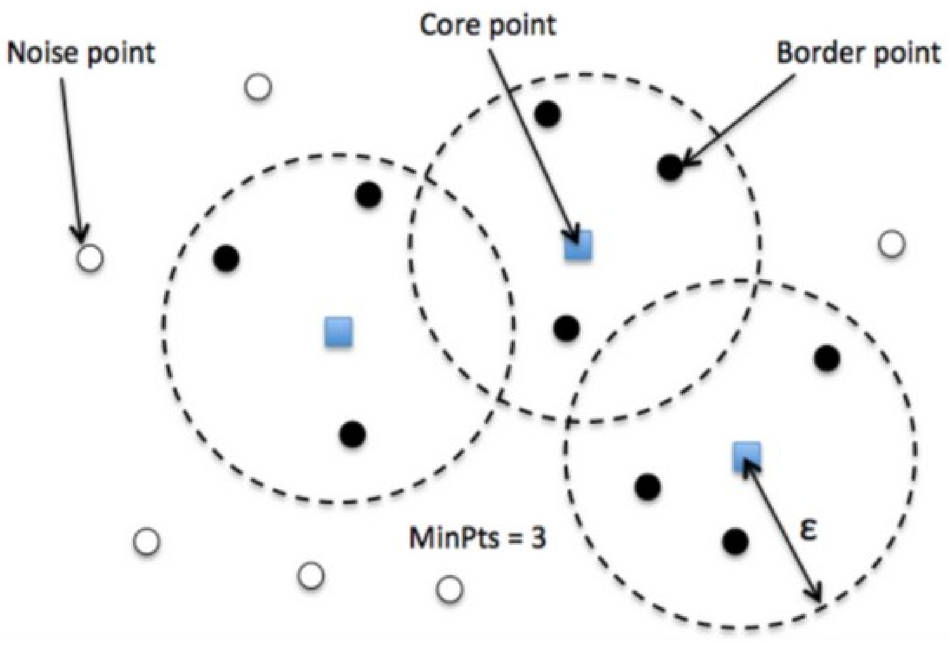


Figure 11. Core, Border and Noise Points[17]

To comprehend the entire process of the DBSCAN algorithm, it's essential to delve into the concepts of *Density Reachability* and *Density Connectivity*. By definition, a point *b* is directly density-reachable from a point *a* if *a* is a core point, and *b* is within its neighborhood. This implies that only core points have the capacity to reach non-core points in the context of density reachability. For example, on the diagram below for *min_samples=4*:

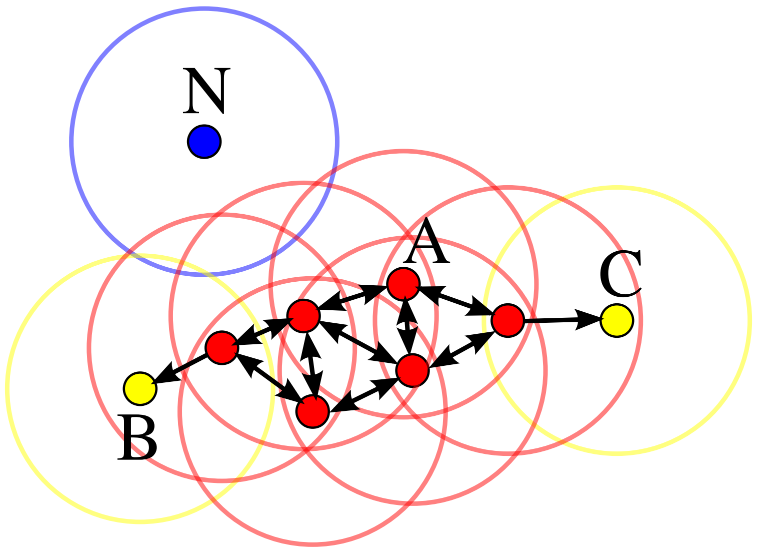


Figure 12. Density Reachability[18]

Point A and all the red points are core points as they contain at least 4 points, including itself, whereas points B and C are non-core points. Point A is directly density-reachable from the three red points that are connected to it with a "double arrow" and vice versa. Points B and C are directly *density-reachable from each red point that is closer to them, respectively, but* indirectly *density-reachable from point A. Also, point A and all the red ones are* not *density-reachable from points B and C, and lastly, point N is an outlier, so it is not density-reachable from any other point.*

Concerning Density Connectivity, two points *a* and *b* are considered density-connected if there exists a point *c* such that both *a* and *b* are density-reachable from *c*, forming a chaining process. Density Connectivity is a symmetric relation, in contrast to Reachability, which is asymmetric.

To sum up, the steps of the DBSCAN algorithm are the following:

1. For every point, find all the neighbor points within eps distance.
2. Identify the core points with at least min_samples as neighbors.
3. Create a new cluster for every core point.

4. Find all the density-connected points for each core point and assign them on the same cluster.
5. Assign each remaining non-core point to a nearby cluster, if possible, otherwise consider it as noise.

The advantages of DBSCAN are:

- It does not require specifying the number of clusters, unlike k-means which needs k as an input.
- It can handle clusters of different shapes and sizes.
- It can detect outliers and separate them from clusters.
- It requires only two parameters and is insensitive to the order of data.

On the other hand, the disadvantages of DBSCAN are:

- It cannot handle varying densities.
- It can struggle on border points that are reachable from more than one clusters, depending on the order the data are processed.
- It can struggle on high dimensional data, as it depends only on distance measures.
- It is hard to determine the correct set of parameters.

4.1.2 OPTICS

Ordering Points To Identify the Clustering Structure (OPTICS)[16] is a density-based algorithm that, unlike DBSCAN, can handle varying densities and shapes and can identify hierarchical structures. Unlike DBSCAN, where we need to pre-determine an optimal epsilon parameter, OPTICS processes multiple distance parameters simultaneously. It works with an infinite number of epsilons (epsi) smaller than a "generating distance" or max_eps, with the constraint *0 <= epsi <= max_eps*.

To ensure a consistent result, OPTICS stores the order in which the data are processed, giving priority to high-density clusters, specifically those with the lowest epsilon. This allows OPTICS to be effective in identifying hierarchical structures within the data. The information it provides consists of two values:

- **core distance**: is the minimum value of epsilon to classify a point as a core point. If it is not a core point, its core distance is *UNDEFINED*:

$$\text{core-dist}_{\varepsilon,MinPts}(p) = \begin{cases} \text{UNDEFINED} & \text{if } |N_\varepsilon(p)| < MinPts \\ MinPts\text{-th smallest distance in } N_\varepsilon(p) & \text{otherwise} \end{cases}$$

where p is a data point, ε is the distance value, $N_\varepsilon(p)$ is the neighborhood of the data point for the specific distance value, *MinPts* is the min_sampes value and *MinPts-distance(p)* is the data points’ distance from its MinPts-th neighbor.

- **reachability distance**: is the maximum of the core distance of point *p* and the Euclidean distance (or any other metric) between points *o* and *p*. If *p* is not a core point, its core distance is *UNDEFINED*:

$$\text{reachability-dist}_{\varepsilon,MinPts}(o,p) = \begin{cases} \text{UNDEFINED} & \text{if } |N_\varepsilon(p)| < MinPts \\ \max(\text{core-dist}_{\varepsilon,MinPts}(p), \text{dist}(p,o)) & \text{otherwise} \end{cases}$$

where *o* and *p* are data points, ε is the distance value, $N_\varepsilon(p)$ is the neighborhood of *p* data point for the specific distance value and *MinPts* is the min_sampes value.

After calculating the reachability distance for every data point, the OPTICS algorithm constructs an ordered list of points, known as the clustering structure of the dataset. To facilitate the visualization of this clustering structure, OPTICS generates a *reachability plot*. This plot illustrates the reachability distance values for each data point in the order in which they appear in the cluster ordering, offering insights into the clustering patterns within the dataset.

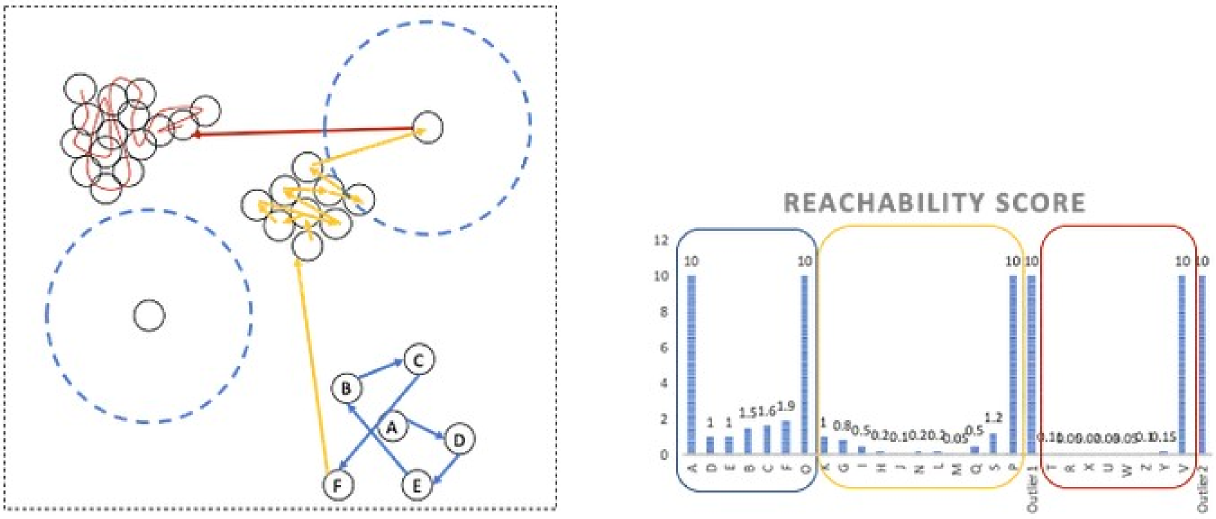


Figure 13. Reachability Distances and Reachability Plot[19]

On the left side we have a visual representation of reachability distances and on the right side the reachability plot, which corresponds to the three clusters on the left.

In the ordered list depicted on the reachability plot, each point is associated with a reachability distance, signifying the ease with which that specific data point can be reached from other points in the dataset. Notably, clusters with *higher* density are reflected as *deeper* "valleys" on the plot, whereas clusters with *lower* density appear as *shallower* "valleys." This observation suggests that points with similar reachability distances are more likely to belong to the same cluster, providing a visual indication of the clustering structure within the data.

Finally, the advantages of the OPTICS algorithm are:

- It can handle varying densities.
- It does not need to determine the perfect epsilon, as it only needs it to reduce the process time.
- It can reveal clusters that would not be apparent with a constant epsilon.

On the other hand, the disadvantages of the OPTICS algorithm are:

- It has higher computational complexity than DBSCAN.
- It requires more memory than DBSCAN.

4.2 Divisive hierarchical clustering

Hierarchical clustering is a clustering technique that splits or merges clusters depending on their similarities or differences. There are two types of hierarchical clustering:

- **Agglomerative**: Initially, each data point is regarded as a single cluster. At each subsequent step, similar clusters merge until one or N clusters are formed.
- **Divisive**: Initially, every data point is considered to be part of the same large cluster. At each step, clusters split into sub-clusters based on certain criteria, continuing until each data point becomes an individual cluster or N clusters are formed.

For the purpose of this thesis, we designed a divisive hierarchical clustering model. Divisive hierarchical clustering, in contrast to agglomerative hierarchical clustering, is commonly employed in statistical analysis. It operates as a top-down technique, necessitating either raw data or a distance matrix for execution. When using raw data, it automatically computes the distance matrix in the background, employing a chosen distance metric, such as the Euclidean distance.

As we were unable to find Python tools that met our requirements for this project segment, we developed a custom divisive hierarchical clustering algorithm with three distinct levels. The primary distinctions among these levels lie in the choice of distance metric and the type of data utilized. Our database incorporates both geographical data for the edges and their associated speed measurements. To simplify this, consider the logic behind the algorithm as follows:

1. Consider all data points as a single cluster.
2. Choose the distance metric that will be applied on each step.
3. Split every cluster into sub-clusters using the OPTICS or DBSCAN algorithm, until the process is completed.

The following diagram shows an example of our divisive hierarchical clustering implementation.

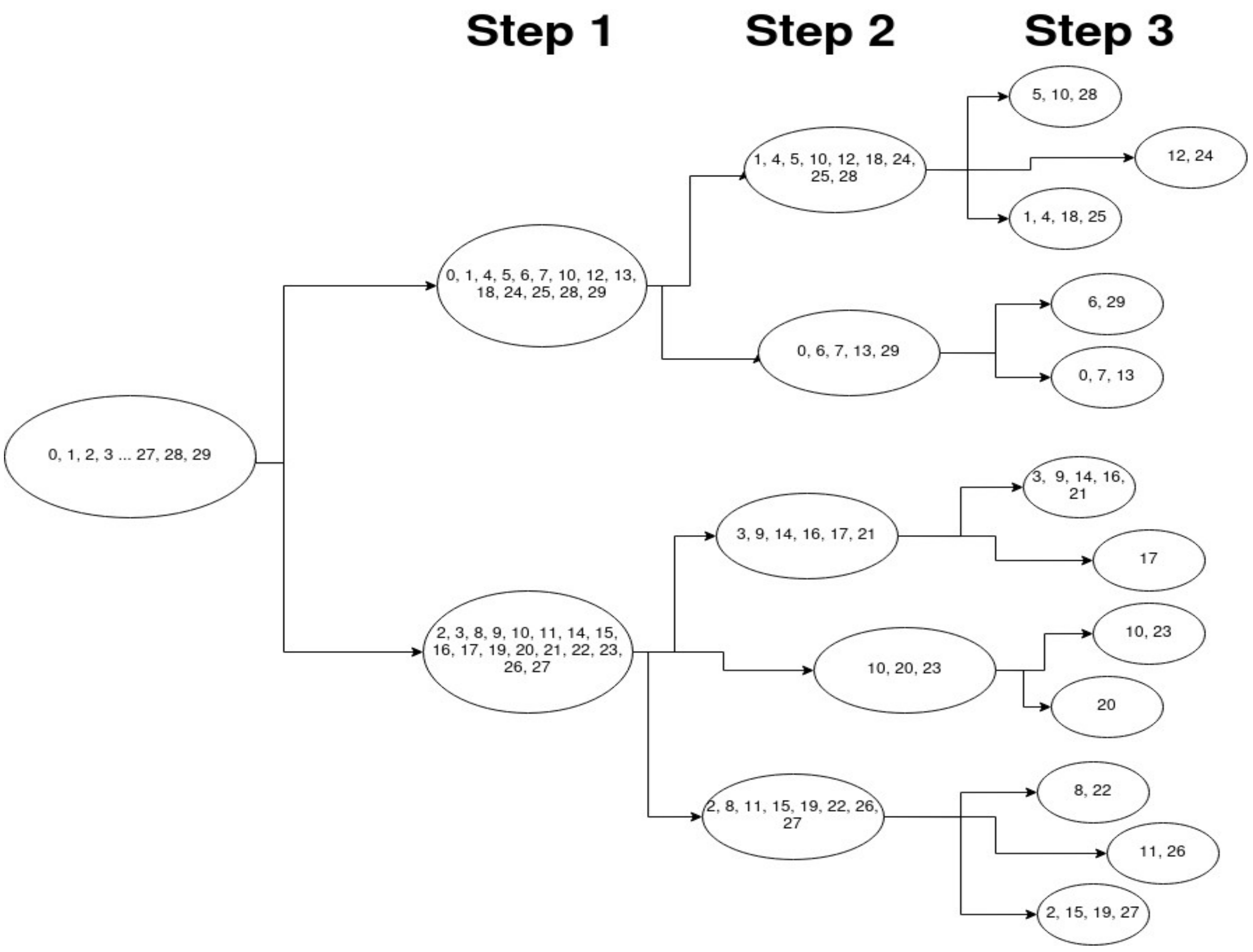


Figure 14. Divisive Hierarchical Clustering Diagram

Lets say that in the first big cluster we have the data points of our time series database. At step 1, we cluster these points and we get two smaller clusters. We follow the same logic using the proper distance metric and data on each step, until we complete them. After step 3, we get the final clusters that will determine the traffic relationships among the edges of our road network.

4.3 The 3-step clustering model

At this point, we will elucidate each step of the 3-step hierarchical clustering model we devised, employing DBSCAN, OPTICS, and other tools from the **scikit-learn** library in Python. We will delve into both the theoretical and practical aspects of each step, presenting our thoughts and work. The approach adopted is rooted in [5].

4.3.1 Step 1

In step 1, where all edges are initially members of the same cluster, our goal was to identify those edges whose traffic exhibits both increasing and decreasing patterns at the same rate. We accomplished this by calculating the **shape-based** distance of the edges, determined by the *Euclidean* distance of their respective normalized time series. To normalize the time series, we utilized the *normalize()* function from the 'sklearn.preprocessing' tool, which employs the Euclidean norm formula. So if $x=(x_1, x_2, \ldots, x_n)$ is a row of our time series database, after normalization it becomes $(x_1/||x||_2, x_2/||x||_2, \ldots, x_n/||x||_2)$, where:

$$\|\boldsymbol{x}\|_2 := \sqrt{x_1^2 + \cdots + x_n^2}.$$

Every row in our database has the data we collected for each edge id which means, by definition, that every row contains the edge id's time series. Now that we have their normalized values, we can calculate their Euclidean distance, as follows:

$$dis_{shape}(e_1, e_2) = dis_{shape}(TS_1, TS_2) = \sqrt{\sum (v'_1[t_i] - v'_2[t_i])^2}, t_s \leq t_i \leq t_e$$

where we assume that we want to calculate the Euclidean distance of edges e_1 and e_2. Then, TS_1 and TS_2 are their corresponding time series for a specific period of time *[t_s, t_e]* and *v_1'[t_i], v_2'[t_i]* are the normalized values at *t_i* for *TS_1* and *TS_2* respectively.

We implemented step 1 using the OPTICS algorithm. Initially, we created an array comprising values selected from our time series database, representing the rows and columns, and subsequently normalized them. The OPTICS algorithm was then applied with specific parameters, including max_eps=0.2, min_samples=10, and the Euclidean distance metric. We determined these parameters through experimentation, assessing their silhouette score, which ranges from -1 (worst) to 1 (best). Scores near 1 signify that data points are distant from other clusters, while scores near 0 suggest overlapping clusters, and negative values indicate incorrect cluster assignments. Considering our normalized data range from 0 to 1, we observed that an epsilon greater than 1 would be impractical. Similarly, a significantly smaller value for min_samples would result in small clusters unlikely to further split in subsequent steps. We present a code snippet of our implementation right below.

```
import numpy as np
from sklearn.cluster import OPTICS
from sklearn.metrics import silhouette_score
from sklearn.preprocessing import normalize

X_train = np.array(time_series.values)
X_normalized = normalize(X_train)

optics_model = OPTICS(min_samples=10, max_eps=0.2,
                         metric='euclidean').fit(X_normalized)

print(f'Score at Step 1 is:
          {silhouette_score(X_normalized,optics_model.labels_)}')
```

Text 5. Step 1 Implementation

4.3.2 Step 2

In step 2, our objective was to identify edges, within the sub-clusters created in step 1, that are topologically close to each other in the network graph. To achieve this, we computed the **structure-based** distance between the data points, determined by the *haversine* distance of their respective source edge points' coordinates. The haversine formula calculates the great-circle distance between two points on a sphere based on their longitudes and latitudes.

The central angle θ between two points on a sphere is:

$$\theta = \frac{d}{r}$$

where d is the distance between the two points (epsilon value) and r is the radius of the sphere (radius of Earth = 6371km).

Assuming that we need to calculate the haversine distance between two edges e_1 and e_2, the haversine of θ would be as follows:

$$\operatorname{hav}(\theta) = \operatorname{hav}(\varphi_2 - \varphi_1) + \cos(\varphi_1)\cos(\varphi_2)\operatorname{hav}(\lambda_2 - \lambda_1)$$

where φ_1, φ_2 are the latitudes of the source edge points of e_1 and e_2, and λ_1, λ_2 are the longitudes of the source edge points of e_1 and e_2, respectively.

Finally, the haversine function, that computes half a versine of the angle θ, applied to both the θ and the latitude, longitude differences is:

$$\operatorname{hav}(\theta) = \sin^2\left(\frac{\theta}{2}\right) = \frac{1 - \cos(\theta)}{2}$$

We implemented step 2 using the DBSCAN algorithm and its haversine distance metric. Initially, we generated an array that included the latitude and longitude values of the source points for the edges within each cluster formed in step 1. Subsequently, we selected a set of min_samples and epsilon parameters, with epsilon being divided by the Earth's radius in kilometers, as the haversine metric requires it in radians. The choice of these parameters was made through experimentation, adjusting them incrementally and decrementally while monitoring the silhouette score. We present a code snippet of our implementation and a plot of its results right below.

```
import numpy as np
from sklearn.cluster import DBSCAN
from sklearn.metrics import silhouette_score

kms_per_radian = 6371
epsilon = 5/kms_per_radian
min_samples = 3

coords = temp[['source_lat', 'source_lng']].values
X_train = np.array(coords)

dbscan_cluster_model = DBSCAN(eps=epsilon, min_samples=min_samples,
             algorithm='ball_tree',metric='haversine').fit(X_train)

print(f'Score at Step 2 (for cluster {i}) is:
     {silhouette_score(X_train, dbscan_cluster_model.labels_)}')
```

Text 6. Step 2 Implementation

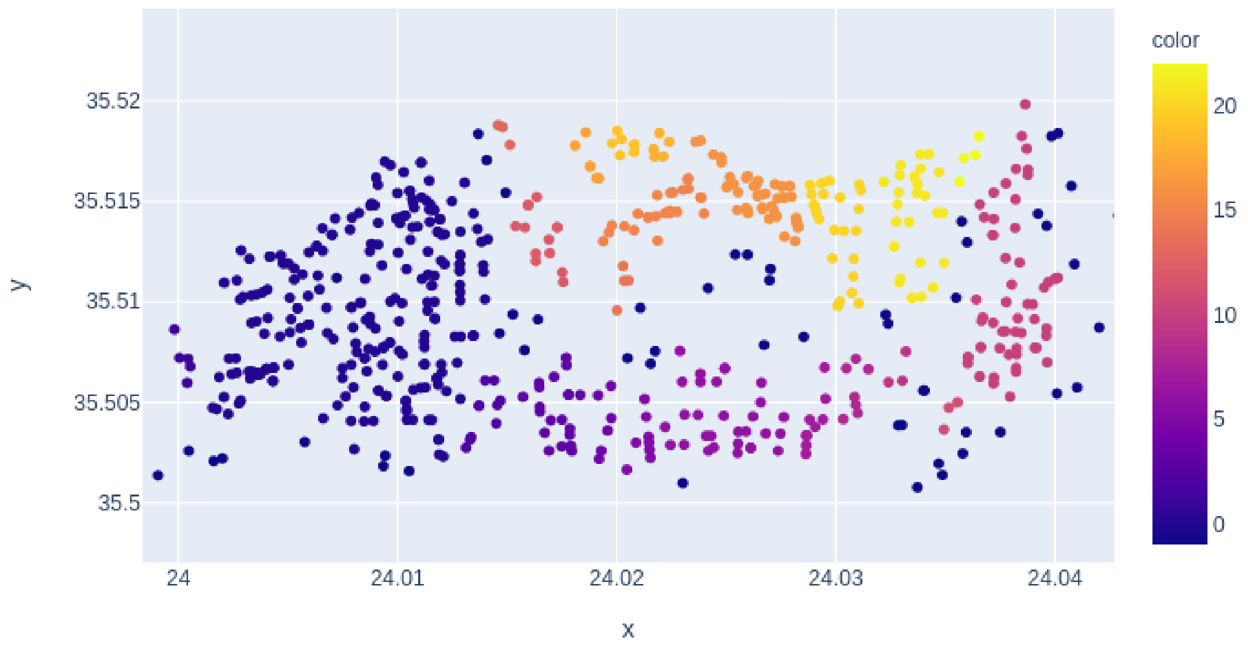


Figure 15. DBSCAN Haversine Distance Results

On x axis we have the latitudes and on y axis the longitudes for the edges' source points of a specific cluster generated after step 1. On the right side of the plot we have the number and the color of the cluster that each edge will be assigned after step 2.

4.3.3 Step 3

In the concluding step, step 3, our aim was to identify edges whose time series exhibit similar values, forming the final clusters of our model. To accomplish this, we computed the **value-based** distance, which is determined by the *Euclidean* distance of their respective time series.

As we mentioned at Section 4.3.1, every row of our database contains the edge id's time series information. Assuming that we want to calculate the Euclidean distance between two edges e_1 and e_2, our formula would be as follows:

$$dis_{value}(e_1, e_2) = dis_{value}(TS_1, TS_2) = \sqrt{\sum (v_1[t_i] - v_2[t_i])^2}, t_s \leq t_i \leq t_e$$

where TS_1 and TS_2 are their corresponding time series for a specific period of time $[t_s, t_e]$ and $v_1[ti]$, $v_2[ti]$ are the corresponding values at t_i for TS_1 and TS_2 respectively.

We implemented step 3 using the OPTICS algorithm and its Euclidean distance metric. Initially, we formed an array comprising the data of the edges within each cluster generated in step 2. Subsequently, we selected a set of max_eps and min_samples parameters based on their silhouette score. For min_samples, we chose the minimum number that makes sense, considering it's the final step in our divisive hierarchical clustering model. Regarding the max_eps parameter, we conducted experiments by adjusting it incrementally and decrementally to optimize the silhouette score. We present a code snippet of our implementation right below.

```
import numpy as np
from sklearn.cluster import OPTICS
from sklearn.metrics import silhouette_score

X_train = np.array(time_series.values)

optics_model = OPTICS(min_samples=2, max_eps=2,
                              metric='euclidean').fit(X_train)

print(f'Score at Step 3 (for cluster {i}, {j}) is:
          {silhouette_score(X_train,optics_model.labels_)}')
```

Text 7. Step 3 Implementation

4.4 Separation of traffic relationships

We enhanced our clustering algorithm by introducing an additional feature that allows users to extract specific traffic relationships from the street network. Users now have the flexibility to choose between extracting all traffic relationships, only the propagates, or only the splits/merges. This customization empowers users to tailor the information according to their needs, facilitating its use for various research purposes or gaining a better understanding of traffic flow dynamics.

To implement this feature, we utilized the ‘geopy.distance’ module in Python, which can calculate geodesic distances between two points. Geodesic distances represent the shortest path between two points on a curved surface, making it particularly suitable for our Earth-based context. Following we have a code snippet of our implementation and its explanation.

```
import geopy.distance as dist

cur_cluster = final_df.loc[final_df['final_clusters'] == cluster]

for i in cur_cluster.index:
      coords_source1 = (cur_cluster.iloc[i]['source_lng'],
                              cur_cluster.iloc[i]['source_lat'])
      coords_dest1 = (cur_cluster.iloc[i]['destination_lng'],
                              cur_cluster.iloc[i]['destination_lat'])
      lng_source1 = cur_cluster.iloc[i]['source_lng']
      lng_dest1 = cur_cluster.iloc[i]['destination_lng']

      if dist.geodesic(coords_source1, coords_dest1).km >= 0.15:
            propagates.append(cur_cluster.iloc[i])

      for j in range(i, len(cur_cluster)-1):
            lng_source2 = cur_cluster.iloc[j+1]['source_lng']
            lng_dest2 = cur_cluster.iloc[j+1]['destination_lng']

            if ((lng_source1 == lng_dest2) or (lng_dest1 ==
                                                  lng_source2)):
                  propagates.append(cur_cluster.iloc[i])
                  propagates.append(cur_cluster.iloc[j+1])
```

Text 8. Separation of Traffic Relationships

We observed that within a group of actual edges, those indicating propagation should belong to the same cluster and be consecutive. While our approach may seem slightly unconventional due to working with hypothetical edges, it is constructed as follows:

- For each edge within every cluster generated after step 3, we initially examine the distance between its starting and ending points. Given that our edges may comprise two or more actual edges, if the distance between the corresponding points of an edge is equal to or greater than 150 meters, we categorize that hypothetical edge as indicating traffic propagation.
- Subsequently, we assess whether the identified edge is connected to any other edge within the same cluster. This is achieved by comparing the longitudes of their source and destination points, with the aim of identifying consecutive edges. By definition, the presence of consecutive edges indicates traffic propagation in our context.
- Upon completing this process for each cluster and its associated edges, any edges that remain are considered to represent splits/merges in the context of our clustering algorithm.

4.5 Experimental results

In our initial attempts to implement the 3-step clustering model, we presumed that the DBSCAN algorithm would be suitable for each step since it provides both Euclidean and haversine distance metrics essential for clustering our time series database. However, as we progressed, we encountered challenges in determining a notable set of min_samples and eps input parameters, particularly in steps 1 and 3 where we dealt with data points of varying densities and shapes. This led us to explore alternative algorithms, ultimately choosing OPTICS for these steps. As previously explained, OPTICS eliminates the need for a perfect predetermined epsilon but introduces a max_eps parameter serving as a threshold for the process duration.

For step 2, we persisted in using the DBSCAN algorithm since OPTICS lacked the haversine distance metric crucial for computing the structure-based distance, resulting in a hybrid model to meet our specific requirements.

To validate our approach, we strategically placed markers in the city center, where we had a good understanding of the traffic flow. We recorded the traffic relationships immediately before and after each marker. Subsequently, we conducted experiments on our clustering algorithm using DBSCAN exclusively for every step, OPTICS for every step, and finally, our HYBRID model. Throughout these experiments, we assessed the accuracy of each model in identifying the traffic relationships among the markers.

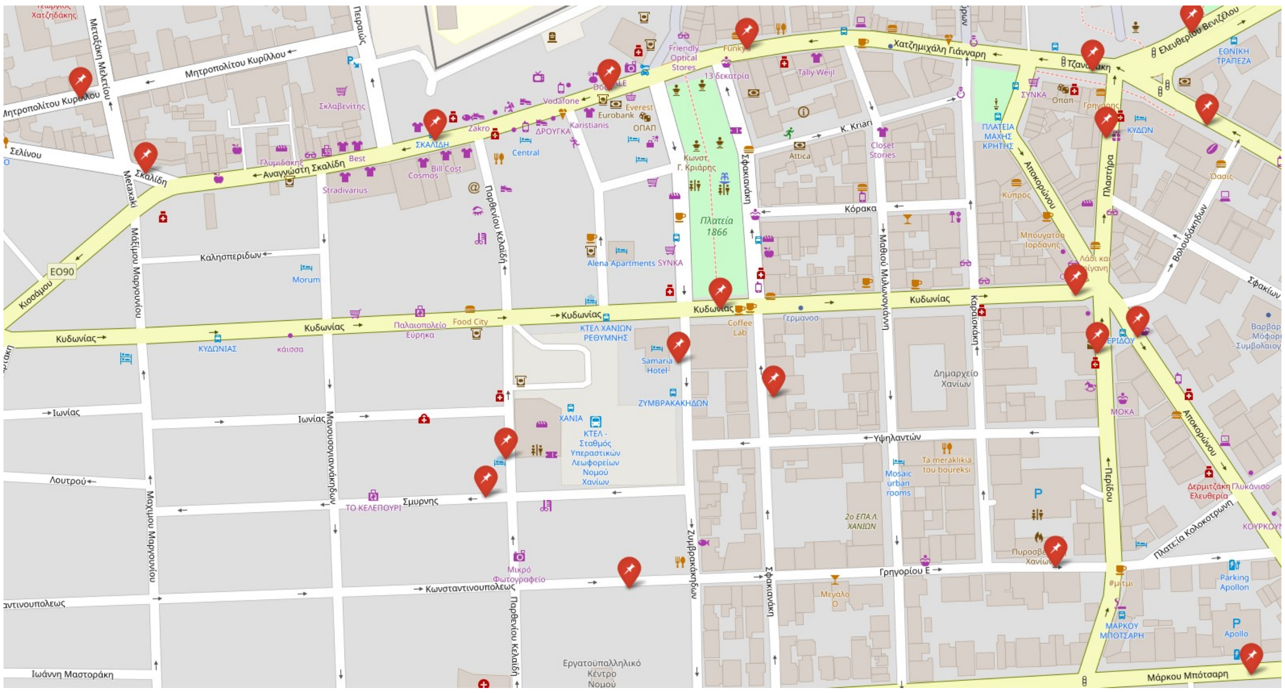


Figure 16. Markers' Position

Next, we present two charts illustrating the results of this experiment, conducted for the same day, month, time span, and time interval. The horizontal axis represents traffic relationships, while the vertical axis depicts the percentage accuracy for each implementation.

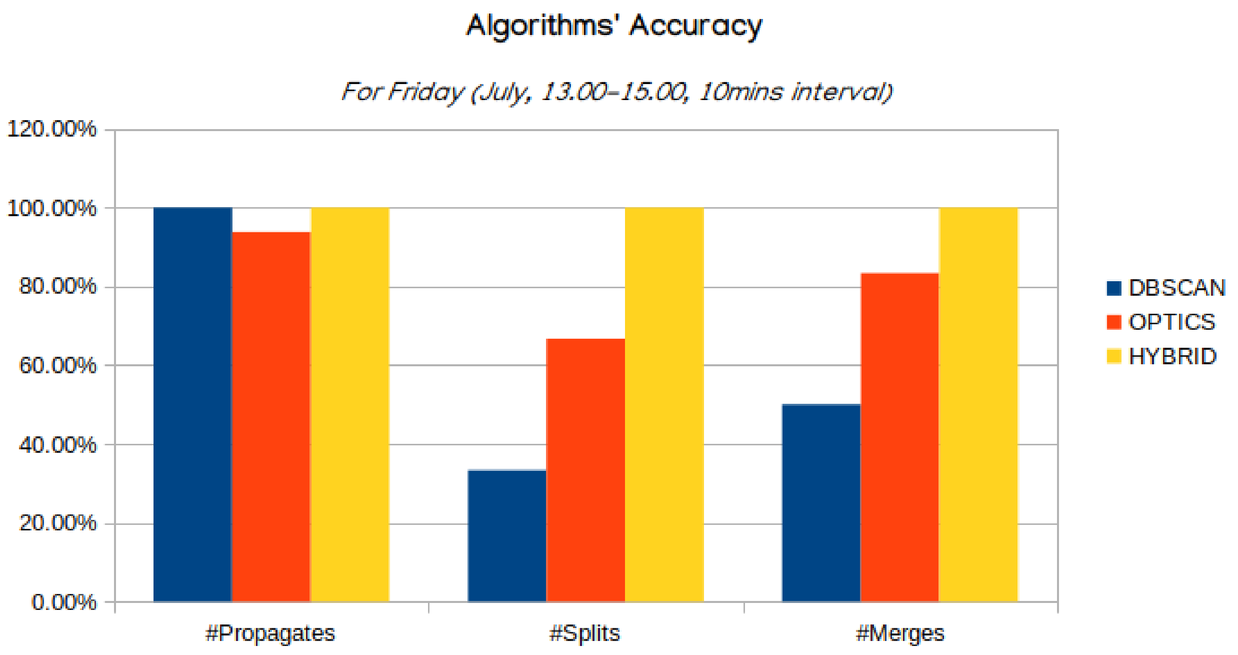


Figure 17. Algorithms' Accuracy Graph (1)

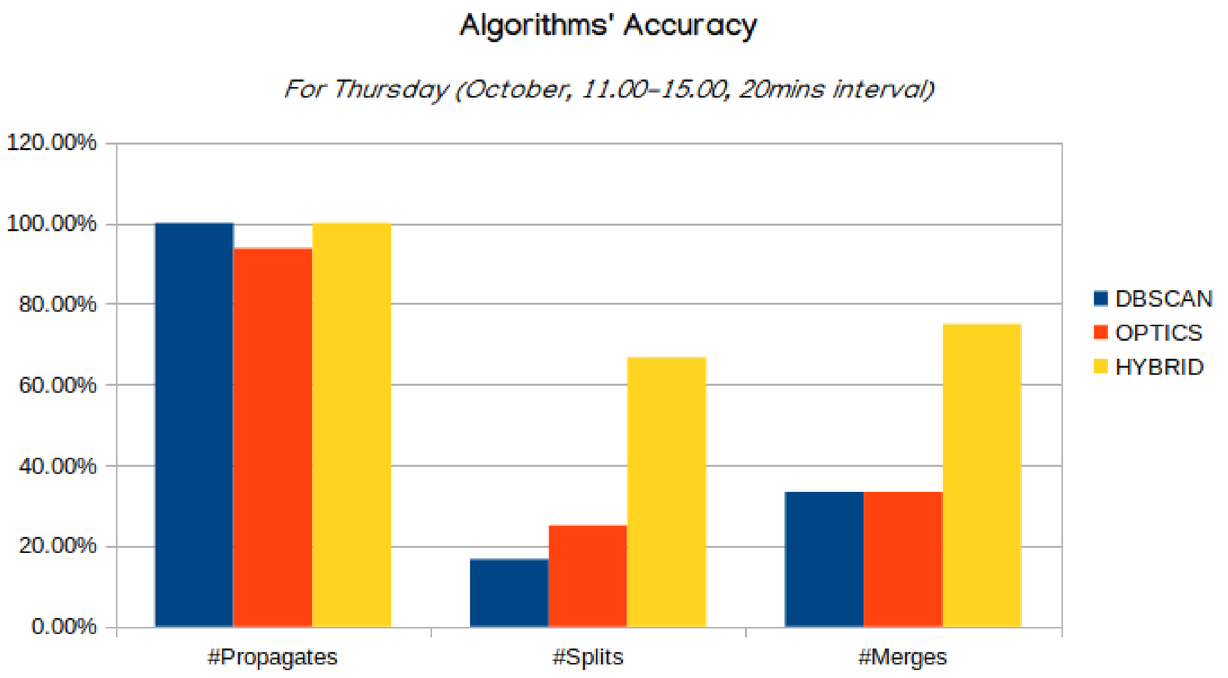


Figure 18. Algorithms' Accuracy Graph (2)

4.6 Display of clustered edges

For visualizing the results of our divisive hierarchical clustering model alongside the verification markers, we employed the **osmnx**[20] and **folium**[21] libraries in Python. We will delve into the capabilities of these tools and their utility in projects that demand the visual representation of road networks.

4.6.1 OSMnx

OSMnx is an open-source Python package built on top of *NetworkX*, *Matplotlib*, and *GeoPandas*, offering powerful capabilities for real-world road network analysis and visualization. It facilitates the download, modeling, analysis, and visualization of street networks and other geospatial features from OpenStreetMap.

To model a road network using OSMnx, one can acquire the necessary GIS data by finding and downloading the appropriate shapefiles online. OSMnx simplifies this process by allowing users to download OpenStreetMap (OSM) data and construct topologically accurate, one-way directional road networks with straightforward queries, such as a place name or a bounding box. It performs pre-processing on the raw OSM data, converting them into a *NetworkX MultiDiGraph*.

For network analysis and spatial network statistics calculations, OSMnx can handle tasks such as finding and plotting the shortest-path routes between two or more points. Additionally, the OSMnx package can convert a NetworkX graph into a *GeoPandas GeoDataFrame*, which represents the tabular format of the network. This makes it easy to customize resulting maps using GeoPandas mapping tools. Below is a code snippet showcasing our implementation, providing insight into the functionality of the aforementioned statements:

```
import osmnx as ox

graph = ox.graph_from_place('Municipality of Chania',
                                      network_type='drive')

origin = ox.distance.nearest_nodes(graph, Xsrc,  Ysrc)
destination = ox.distance.nearest_nodes(graph, Xdst, Ydst)
route = ox.distance.shortest_path(graph, origin, destination,
                                                  weight='length')

gdf_route = ox.utils_graph.route_to_gdf(graph, route, weight='length')
route_map = gdf_route.explore(tiles='OpenStreetMap', tooltip=False,
                   color=color, style_kwds=dict(opacity=0.6, weight=7))
```

Text 9. OSMnx

We begin by creating a NetworkX MultiDiGraph using the "Municipality of Chania" query to obtain the place boundary polygon and specifying the "drive" type for the street network. Subsequently, we utilize the *nearest_nodes()* function to identify the nearest nodes to the source and destination points of an edge. Here, *Xsrc, Ysrc, Xdst,* and *Ydst* represent their respective coordinates in l*ongitude-latitude* format.

With the nodes identified, we compute the shortest path between them and generate a GeoDataFrame of the route. Finally, to visualize the results interactively, we leverage the GeoDataFrame's built-in *explore()* function. We select the OpenStreetMaps tileset for the map, define a style for the plotted edge, and assign a color representing its cluster. This creates an interactive *Leaflet* map for a comprehensive view of the road network analysis.

4.6.2 Folium

Completing the process outlined in Section 4.6.1 for every edge in our database resulted in an interactive Leaflet map that displayed the road network edges, color-coded according to their respective clusters. To incorporate markers for verifying our experiments on clustering methods, we opted for the folium library in Python. Folium leverages the data manipulation capabilities of the Python ecosystem and the mapping strengths of the Leaflet.js library.

To achieve this, we first constructed a file containing the coordinates, in longitude-latitude format, of each marker. We then added these markers to the appropriate locations on the map already created with osmnx. For visualization, we utilized pop-up messages to distinguish the markers and enhance the interactive leaflet map. Following we have our implementation of this process.

```
import folium
import pandas as pd

markers_df = pd.read_csv('input/markers.csv',
                              dtype={'marker_location':str})

for ind in markers_df.index:
     message = '<b>Marker'+str(ind)+'</b>'
     cur_location = markers_df.loc[ind,
'marker_location'].split(' ')
     folium.Marker(location=[float(cur_location[0]),
                         float(cur_location[1])], popup=message,
                         icon=folium.Icon(color='red',
                         icon='pushpin')).add_to(route_map)
```

Text 10. Folium

4.6.3 The interactive map

Upon completing the addition of colored edges and markers on the 'route_map', we save it as an HTML file and open it using our web browser. The visualization of our clustering results provides a more accessible means for verifying our modeling logic and identifying the traffic relationships that emerge among the edges of the road network. This visual representation enhances the clarity of our clustering outcomes. Knowing that every cluster has its own unique color, we could define the traffic relationships as follows:

- **Propagate**: the color remains the same on the consecutive edges that propagate their traffic.
- **Split**: the color of an edge splits into two different colors of the edges that it splits the traffic.
- **Merge**: the different colors of two edges merge into one third different color of the edge that merges their traffic.

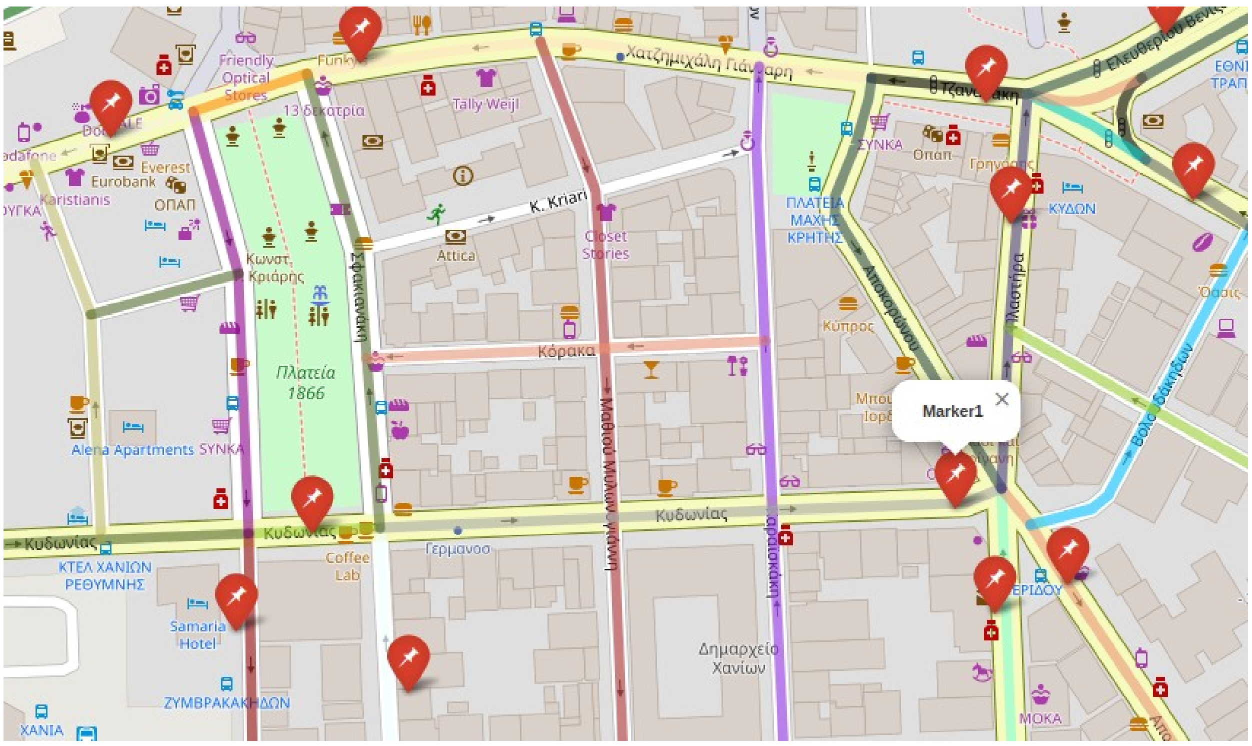


Figure 19. The Interactive Map

Before Marker1 we have traffic propagation (gray) and after the marker we have traffic split (purple, orange).

4.7 Conclusions & Future work

Divisive hierarchical clustering is infrequently employed by developers, leading to a scarcity of dedicated libraries in Python for such implementations. By decomposing our clustering algorithm into three distinct steps and addressing each one independently, we successfully amalgamated them to create our own hierarchical clustering model. The pivotal role played by the powerful DBSCAN and OPTICS algorithms cannot be overstated; they furnished us with indispensable tools and information crucial for navigating through each step. These algorithms not only facilitated the execution of each phase but also enabled us to scrutinize and refine our results. Their adeptness in handling the diverse densities and shapes inherent in our dissimilar hypothetical edges' time series and spatial data proved instrumental in resolving one of our primary challenges.

Moreover, the visual representation of our data through the use of the osmnx tool significantly facilitated the examination of the final results from our custom divisive hierarchical clustering model. It transformed our raw data, which were initially challenging to interpret, into a format that is easily understood by humans. The tool's seamless integration with OpenStreetMaps allowed us to effortlessly display our edges' data without the need for additional modifications in our code. In essence, osmnx emerges as an optimal choice for visualizing street networks reliant on OSM data, offering a diverse array of built-in functions capable of addressing a wide range of tasks.

In summary, the hybrid model we adopted yielded satisfactory results. However, we acknowledge that if, in the future, the OPTICS algorithm incorporates the haversine distance metric, it would be preferable to utilize this algorithm for step 2 as well.

Chapter 5

The application

In this chapter, we will illustrate our transformation process of transitioning from a *command-line interface* (CLI) script for clustering, to a *graphical user interface* (GUI) desktop application. This shift aims to enhance user interaction by leveraging graphical elements. We will showcase the tools employed for designing the interface and provide an explanation of the functionality pertaining to its widgets.

5.1 CustomTkinter

CustomTkinter[22] is an extension of the widely used Tkinter module in Python, representing a modernized version with added UI elements that offer extensive customization options. Tkinter itself is a lightweight, cross-platform GUI framework that is renowned for its simplicity and is commonly employed for developing desktop applications. It provides numerous built-in widgets for designing interfaces to suit various requirements.

In alignment with this philosophy, CustomTkinter enhances the capabilities of Tkinter by introducing additional widgets such as:

- **CtkFrame**: a “container” widget that groups the other widgets together.
- **CtkLabel**: a widget to display text.
- **CtkTextBox**: a widget to display multi-line, scrollable text or to take input from the user.
- **CtkSwitch**: a widget used for toggle options.

- **CtkOptionMenu**: a widget to display a list of options/values.
- **CtkRadioButton**: a widget that allows the user to choose only one of a predefined set of options.
- **CtkButton**: a widget to display a button that can be clicked to perform an action.
- **CtkTabview**: a widget that creates tabs of CtkFrames.

In order to be able to organize the geometry of these widgets in the application frame, CustomTkinter offers three methods:

- **the pack() method**: organizes the positioning of widgets in relation to each other.
- **the grid() method**: organizes the positioning of widgets in a two dimensional grid of rows and columns.
- **the place() method**: organizes the positioning of widgets either with x, y coordinates or relative to another widget.

It is important to note that we should not combine these methods in the same master window, but we should choose one and stick with it instead. Therefore, to sum up, the steps of creating a desktop GUI application using the CustomTkinter module are the following:

1. Import the CustomTkinter module.
2. Create the main window of the application.
3. Add a CtkFrame as a container for the widgets.
4. Add widgets to the frame and apply their functionality.
5. Use the mainloop() function to run the application.

5.2 Design & Functionality

Our aim was to create a straightforward application with user-friendly options, ensuring ease of use without the need for an instruction manual. To achieve this, we implemented a vertical positioning of the widgets using the *pack()* method. We fine-tuned the padx and pady variables to optimize the aesthetic appeal of the interface.

For the majority of our options, we employed CustomTkinter's ‘CtkOptionMenu’ widget, and for time's am/pm selection, we utilized the ‘CtkRadioButton’ widget. The title of each option menu and the error message were designed using the ‘CtkLabel’ widget. Finally, for the button responsible for initiating the clustering based on our preferences, we used the ‘CtkButton’ widget. Below is an example of our setup:

```
import customtkinter

app = customtkinter.CTk()
app.geometry("450x850")

frame = customtkinter.CTkFrame(master=app)
frame.pack(pady=30, padx=50, fill="both", expand=True)

label = customtkinter.CTkLabel(master=frame, text="Menu Title")
label.pack(pady=(30,0), padx=10)

optionmenu = customtkinter.CTkOptionMenu(frame, width=200,
                          values=[option_values],
command=message_update)
optionmenu.pack(pady=0, padx=10)

button = customtkinter.CTkButton(master=frame, text="Button",
                fg_color="green", border_width=1, border_color="white",
                width=80, command=button_callback)
button.pack(pady=(30,0), padx=10)

app.mainloop()
```

Text 11. CustomTkinter Interface

The CustomTkinter module offers a wealth of customizations for its widgets, providing developers with various styling options. The most noteworthy aspect of the widgets' variables is the command parameter, which signifies their functionality. For instance, the *message_update()* function is responsible for updating the error message label whenever an error occurs. On the other hand, the *button_callback()* function performs preprocessing on the user's selections before providing them as input to our clustering routine. This modular approach enhances the flexibility and functionality of the GUI application.

```
from clustering import clustering_routine

def button_callback():
      am_to_pm1 = int(optionmenu_from.get().split(":")[0])
      am_to_pm2 = int(optionmenu_to.get().split(":")[0])

      if radiobutton_var_from.get() == 2 and optionmenu_from.get() != "12:00":
            am_to_pm1 += 12
            start_time = str(am_to_pm1) + ":00"
      elif radiobutton_var_from.get() == 1 and optionmenu_from.get() == "12:00":
            am_to_pm1 -= 12
            start_time = str(am_to_pm1) + "0:00"
      elif radiobutton_var_from.get() == 1 and am_to_pm1 < 10:
            start_time = "0" + str(am_to_pm1) + ":00"
      else:
            start_time = optionmenu_from.get()

      if radiobutton_var_to.get() == 2 and optionmenu_to.get() != "12:00":
            am_to_pm2 += 12
            end_time = str(am_to_pm2) + ":00"
      elif radiobutton_var_to.get() == 1 and optionmenu_to.get() == "12:00":
            am_to_pm2 -= 12
            end_time = str(am_to_pm2) + "0:00"
      elif radiobutton_var_to.get() == 1 and am_to_pm2 < 10:
            end_time = "0" + str(am_to_pm2) + ":00"
      else:
            end_time = optionmenu_to.get()
```

Text 12. Button Command (part1)

Firstly, we import our divisive hierarchical clustering routine. Next, we reconstruct the starting (‘optionmenu_from’) and ending (‘optionmenu_to’) times of the measurements that we are going to cluster, based on the user's am/pm preferences. In the first case, we search for the pm version of times ranging from 1:00 to 11:00, and we add 12 to their first two digits to reconstruct them in a format ranging from 13:00 to 23:00. In the second case, we look for the am version of 12:00 and reconstruct it to 00:00. In the third case, we add a leading zero to the am version of times ranging from 1:00 to 9:00, so as to reconstruct them in a format ranging from 01:00 to 09:00. In the last case, we simply retrieve the user's selection of time as is, following the same logic for both ‘start_time’ and ‘end_time’.

```
if optionmenu_interval.get().split(" ")[1] != "hour":
            parameter = int(int(optionmenu_interval.get().split(" ")[0]) / 10)
      else:
            parameter = 6

input_file = "data/"+optionmenu_month.get()+"/"+optionmenu_day.get()+".csv"
relationships = optionmenu_relationships.get()

try:
      clustering_routine(input_file, start_time, end_time, parameter,
                                                                relationships)
except:
      label_message.configure(text="Invalid options.\nTry again!",
                                                          text_color="red")
```

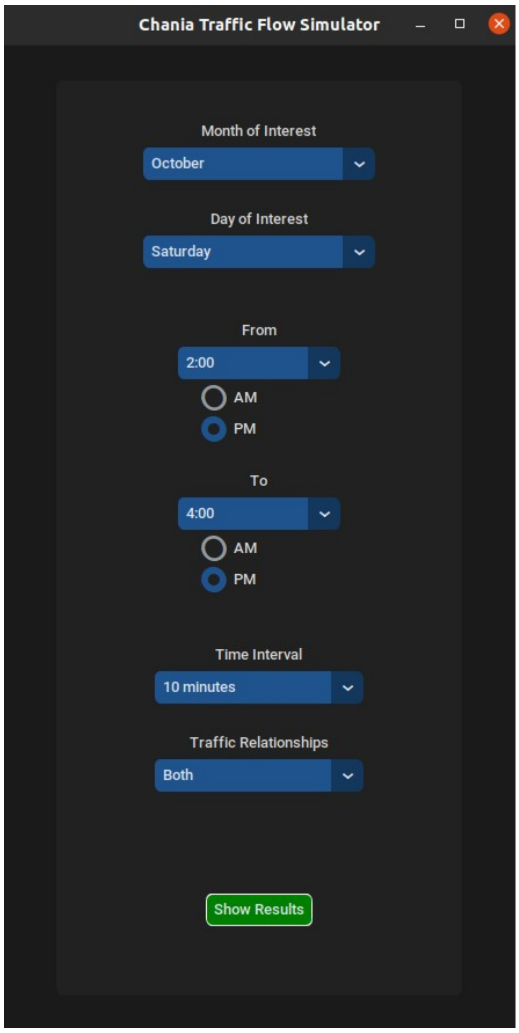
Text 13. Button Command (part 2)

After reconstructing the time values, we extract the parameter for the time interval selected by the user, dividing it by 10, as the data in our time series database are scraped with a transmission rate of 10 minutes. Using the chosen day and month, we construct the path for the input file. Finally, we use all the above as input variables for our divisive hierarchical clustering routine.

5.3 Usage

The options available in our application's interface are as follows:

- **'Month'** and **'Day'**: Users can select the month and day for which traffic data will be clustered.
- **'From'** and **'To'**: Users can specify the start and end times for the measurements collected on the chosen day and month.
- **'Time Interval'**: Users have the flexibility to choose the time interval between samples in the time series database.
- **'Traffic Relationships'**: Users can indicate which of the detected traffic relationships they want to display.


Chania Traffic Flow Simulator

Month of Interest: October

Day of Interest: Saturday

From: 2:00 AM [✓ PM]

To: 4:00 AM [✓ PM]

Time Interval: 10 minutes

Traffic Relationships: Both

Figure 20. Application's Interface

If we click the 'Show Results' button with the specified options in Figure 17, we would cluster the traffic data collected on a Saturday in October. To be more precise, we would select the average speeds measured between 14:00 and 16:00 with a time interval of 10 minutes. After applying our 3-step divisive hierarchical clustering model to these data, an interactive map would be generated. This map would display the edges of our road network, with each edge colored according to the cluster to which it belongs.

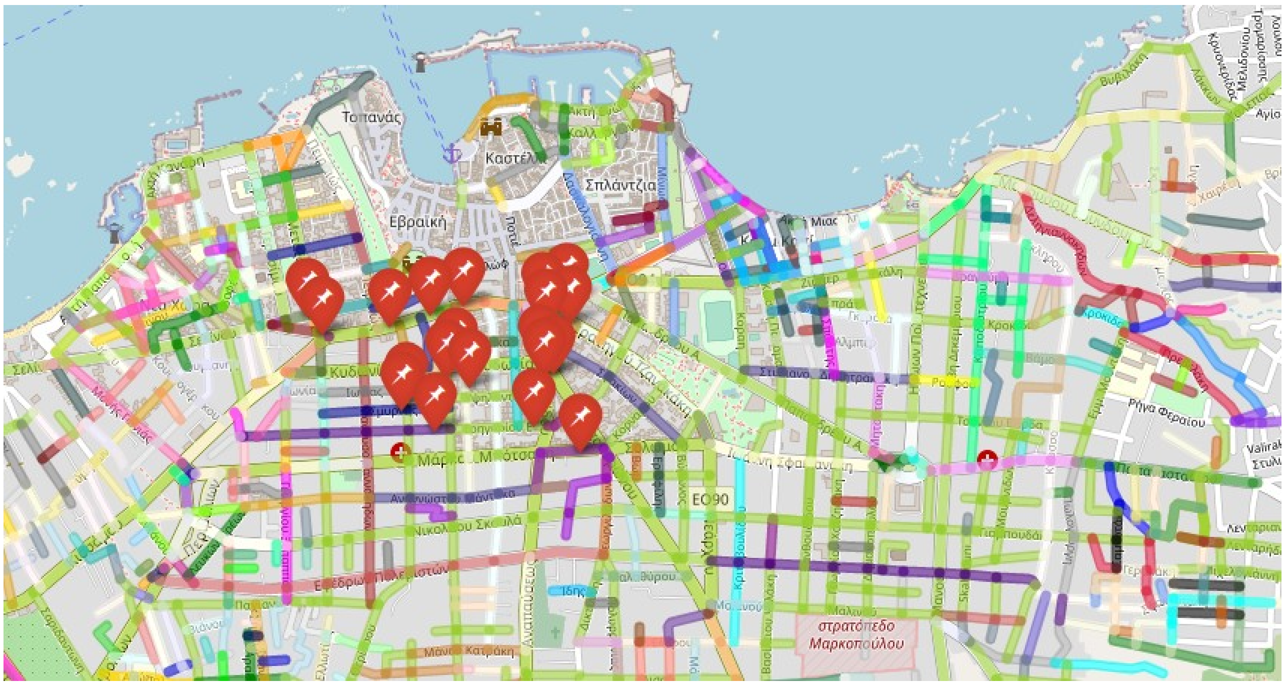


Figure 21. Application's Results

Visualizing our clustering results allows us to observe the traffic relationships among the edges, providing insights into how traffic flows within the road network. By zooming in on the map depicted in Figure 18 and applying the definitions outlined in Section 4.6.3, we can pinpoint traffic propagation, split, and merge. This visual representation enhances our understanding of the intricate dynamics of traffic patterns within the network.

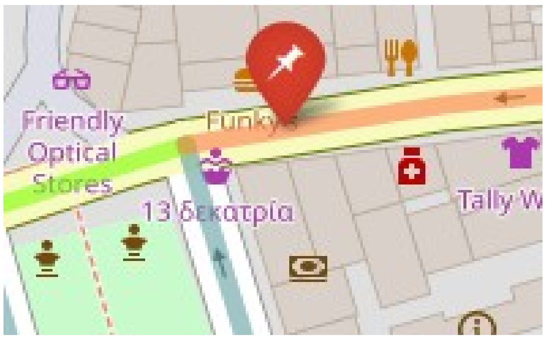


Figure 22. Traffic Merge

The 'orange' and 'light blue' edges merge their traffic into the 'light green' edge.

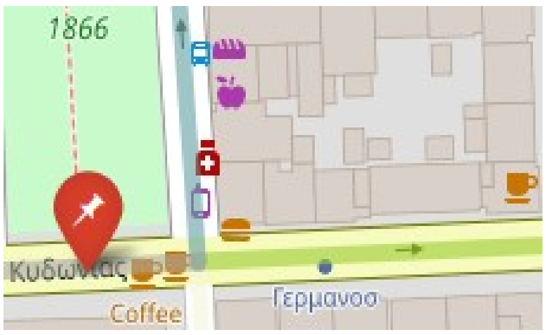


Figure 23. Traffic Split

The 'gray' edge splits its traffic between the 'light blue' and 'light green' edges.

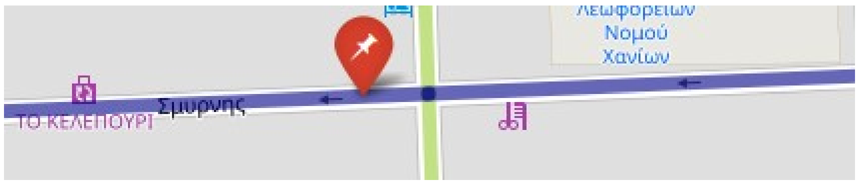


Figure 24. Traffic Propagation

The 'blue' edge, preceding the marker, propagates its traffic to the 'blue' edge after the marker.

5.4 Conclusions & Future Work

Utilizing the CustomTkinter module, we successfully crafted an aesthetically pleasing and user-friendly GUI desktop application that encompasses the necessary functionality for executing our clustering routine. As a suggestion for future work, we propose the development of a web-based version to make it accessible to users through the internet without requiring installation. This would enhance the convenience and reach of the clustering tool.

Chapter 6

Epilogue

In this thesis, we addressed the challenge of traffic management by creating an application that extracts traffic relationships between the edges of a road network, building upon the approach outlined in [5]. We successfully implemented these techniques in practice, utilizing real-world data and providing a visual representation of the algorithms' results.

The availability of geospatial data through OpenStreetMap played a crucial role in our ability to design and extract information about the road network, aligning seamlessly with data obtained from Google Maps. This alignment facilitated the scraping of traffic data necessary for constructing our time series database, a task that would have been challenging without access to such comprehensive geospatial information.

Consequently, we developed an automated scraping routine utilizing Selenium and Beautiful Soup. This routine enabled the collection of real-time traffic data over extended periods, serving as the foundation for constructing our comprehensive database.

We subjected the collected data to analysis through a 3-step divisive hierarchical clustering model. This model calculates the shape-based, structure-based, and value-based distances using the DBSCAN and OPTICS algorithms. Starting from a single, large cluster encompassing all edges, the model progressively segments the data into smaller sub-clusters, revealing traffic relationships between edges, when displayed on the map.

In the concluding phase, we integrated the clustering process into a simple GUI desktop application, crafted with CustomTkinter, so that it ensures accessibility for a wider audience.

References

[1] Google Maps APIs, https://developers.google.com/maps

[2] Mapbox web services APIs, https://docs.mapbox.com/api/overview/

[3] Mapbox | Maps, Navigation, Search, and Data, https://www.mapbox.com/

[4] OpenStreetMap, https://www.openstreetmap.org

[5] Irene Ntoutsi, Nikos Mitsou, Gerasimos Marketos: Traffic mining in a road-network: How does the traffic flow? Int. J. Bus. Intell. Data Min. 3(1): 82-98 (2008)

[6] PostgreSQL: The world's most advanced open source database, https://www.postgresql.org/

[7] Home - osm2pgsql, https://osm2pgsql.org/

[8] pgAdmin - PostgreSQL Tools, https://www.pgadmin.org/

[9] PostGIS, https://postgis.net/

[10] the QGIS project, https://www.qgis.org/

[11] pandas - Python Data Analysis Library, https://pandas.pydata.org/

[12] GoogleMapPlotter, https://github.com/gmplot/gmplot/wiki/GoogleMapPlotter

[13] Selenium, https://www.selenium.dev/

[14] BeautifulSoup4, https://pypi.org/project/beautifulsoup4/

[15] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. (1996). A density-based algorithm for discovering clusters in large spatial databases with noise. In Proceedings of the Second International Conference on Knowledge Discovery and Data Mining (KDD'96). AAAI Press, 226–231

[16] Mihael Ankerst, Markus M. Breunig, Hans-Peter Kriegel, Jörg Sander. (1999). OPTICS: Ordering Points To Identify the Clustering Structure. ACM SIGMOD international conference on Management of data. ACM Press. pp. 49–60

[17] Nagesh Singh Chauhan,

DBSCAN Clustering Algorithm in Machine Learning, Kdnuggets,

https://www.kdnuggets.com/2020/04/dbscan-clustering-algorithm-machine-learning.html, 2022

[18] DBSCAN, Wikipedia, By Chire - Own work, CC BY-SA 3.0,

https://commons.wikimedia.org/w/index.php?curid=17045963

[19] Yufeng, Understanding OPTICS and Implementation with Python, Towards Data Science,

https://towardsdatascience.com/understanding-optics-and-implementation-with-python-143572abdfb6, 2022

[20] OSMnx 1.7.1 documentation, https://osmnx.readthedocs.io/en/stable/

[21] Folium, https://pypi.org/project/folium/

[22] Tom Schimansky, CustomTkinter, https://customtkinter.tomschimansky.com/

[23] Myra Spiliopoulou, Irene Ntoutsi, Yannis Theodoridis, René Schult: MONIC: modeling and monitoring cluster transitions. KDD 2006: 706-71

[24] Chania Traffic Flow Simulator, https://github.com/st4mk/CTFS.git