ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

GRADO EN INGENIERÍA INFORMÁTICA - INGENIERÍA DEL SOFTWARE (GRUPO EN INGLÉS)

MANAGING OPENSTREETMAP MAPS WITH NEO4J DATABASE FOR TRAFFIC APPLICATIONS

Realizado por
MARIO AROCA PÁEZ

Dirigido por

DANIEL CAGIGAS MUÑIZ

Departamento

DEPARTAMENTO DE ARQUITECTURA Y TECNOLOGÍA DE COMPUTADORES

Sevilla, marzo de 2024

Acknowledgments

First and foremost, I wish to express my deepest gratitude to my tutor for this project, Daniel Cagigas Muñiz, for giving me the opportunity to develop such an innovative and interesting project. His guidance, support, and proximity have been fundamental at every stage of the process.

Additionally, I am profoundly grateful to my parents and my two brothers for their endless love, encouragement, and belief in my aspirations. Their constant support and sacrifices have been the cornerstone of my success, providing me with strength and motivation throughout my four years of university life. I also extend my sincere thanks to my close friends and to Mari, whose companionship, understanding, and unwavering faith in me have been invaluable. Their contributions to my life extend beyond mere words of support; they have been a source of joy, resilience, and inspiration, playing an instrumental role in shaping my journey.

Abstract

In the exploration of integrating OpenStreetMap (OSM) data with Neo4j graph database for advanced traffic applications, an innovative approach has been developed to address the challenges of urban traffic management. This project has successfully transformed extensive geographical data from OSM into a structured graph database using Neo4j, laying the groundwork for sophisticated traffic analysis and simulation capabilities.

The project encompasses the installation and configuration of Neo4j on a Linux system, the creation of a comprehensive toolkit for interacting with the database through both Cypher and Python APIs, and the exploration of various visualization techniques to effectively interpret the transformed data. Through these methodologies, a dynamic and interconnected graph representation of urban areas has been achieved, enabling the analysis of traffic patterns and contributing to the development of efficient traffic management solutions.

By achieving its primary objectives, this project not only demonstrates the potential of merging open-source geographic datasets with graph database technology but also sets a solid foundation for future innovations in traffic management and urban planning. It highlights the significant advantages of using Neo4j for processing complex geographic information. It also paves the way for the development of more advanced traffic simulations and analyses that could incorporate real-time data and various constraints.

This individual effort has enriched both the academic and practical aspects of software engineering and data analysis within the context of intelligent traffic management systems, indicating a promising direction for further research and application in this field.

Table of Contents

Introduction

In an age where the volume of data is growing exponentially, the ability to visualize, understand, and navigate complex networks is more crucial than ever. The realms of geographic information and graph theory have converged to present innovative solutions to intricate problems in urban planning, navigation, and beyond. This project delves into this convergence, focusing on the transformation of OpenStreetMap (OSM) data into a graph database using Neo4j, a leading graph database management system, to facilitate advanced traffic applications.

Presentation of the problem

Urban areas worldwide face the challenge of managing traffic efficiently. The complexity of urban road networks and the dynamic nature of traffic flow necessitate a robust framework for traffic management and analysis. Traditional mapping solutions, while useful, often fall short when it comes to offering the flexibility and depth required for in-depth traffic analysis and simulation. This project addresses this gap by leveraging the rich, community-driven data from OpenStreetMap and the powerful graph database capabilities of Neo4j.

Justification for the choice of the topic

The choice of this topic is strategically aligned with the pressing need for advanced traffic management solutions and fills a significant educational and practical gap within the curriculum of the "Grado en Ingeniería Informática-Ingeniería del Software." Traditional coursework and projects within this program, while comprehensive, rarely delve into the intricate convergence of geographic information systems, graph theory, and data processing at the scale addressed in this project.

This project stands out because it not only involves the integration and analysis of large-scale geographic data but also requires the application of sophisticated software engineering principles and graph database technologies. These areas are increasingly pivotal in the field of software engineering. Yet underrepresented in the conventional curriculum.

Undertaking this project will result in gaining hands-on experience with real-world data and scenarios, transcending the typical theoretical focus of the degree program. This project offers a unique opportunity to apply software engineering principles to solve complex, real-world challenges in traffic management. Acquiring and honing skills in cutting-edge technologies such as Neo4j, a leading graph database management system, and OpenStreetMap data handling, positioning themselves at the forefront of modern software engineering and data analysis practices. And finally working on innovative thinking and problem-solving by tackling a project that demands a deep understanding of complex systems, algorithmic thinking for traffic simulation, and optimization, and the development of a robust and scalable software solution.

By addressing a complex, real-world problem through the lens of advanced computational and software engineering techniques, this project not only enriches the academic experience but also enhances the practical skill set of students, preparing them to meet the demands of the modern tech landscape and to contribute meaningfully to the field of intelligent traffic management systems.

Objectives of the work

The primary objective of this project centers on the innovative conversion of OpenStreetMap (OSM) data into a Neo4j graph database format. This transformation aims to leverage the intricate and comprehensive geographical data available through OSM, and by utilizing the advanced capabilities of Neo4j, enable a more dynamic and interconnected representation of geographic information. This process not only facilitates the sophisticated analysis of traffic patterns but also enhances the management and simulation capabilities within urban planning contexts.

A pivotal aspect of this undertaking involves the installation and configuration of the Neo4j database on a Linux-based system. This local deployment is crucial for ensuring a stable and controlled environment for database manipulation and analysis, providing a foundation for the development and testing of database queries, algorithms, and data manipulation strategies.

Furthermore, an essential goal of this work is to explore and demonstrate the versatility and power of interacting with the Neo4j database through various means. This includes the utilization of Cypher, Neo4j's query language, for complex data querying and manipulation, as well as the integration of Python scripting. Python serves as a bridge to extend the functionality of Neo4j, enabling automated data processing, the execution of sophisticated queries, and the development of custom algorithms for data analysis. This dual approach ensures a comprehensive understanding and manipulation of the data, catering to both direct database interaction and programmatic control.

Lastly, the project aims to showcase the capabilities of Neo4j's visualization tools through local deployment. By effectively utilizing Neo4j's built-in viewer, the project demonstrates how geographic data, transformed and stored within the graph database, can be visually represented. This visualization not only aids in the intuitive understanding of complex datasets but also highlights the relationships and patterns within the urban and traffic infrastructure. This provides invaluable insights into traffic management and urban planning strategies.

In summary, this project endeavors to bridge the gap between vast, open-source geographic datasets and the advanced analytical capabilities of graph databases. By achieving these objectives, it lays the groundwork for innovative applications in traffic management, urban planning, and beyond, harnessing the power of Neo4j to transform raw data into actionable intelligence.

Technological Fundamentals:

Before delving into the specifics of OpenStreetMap (OSM) and Neo4j, it's important to understand the technological backbone of this project. In this section, the two main technologies that have been instrumental in the development of the work will be explored: OpenStreetMap, a comprehensive and open-source map database, and Neo4j, a highly efficient graph database management system. These tools not only provide the foundation for our project but also exemplify the innovative approaches we have taken to tackle the challenges of urban traffic analysis and simulation.

OpenStreetMap

OpenStreetMap (OSM) is a mapping project that empowers users to create, edit, and utilize geographical data. Unlike traditional mapping services like Google Maps or Apple Maps OSM is not managed by a single entity but relies on a global community of contributors who actively update and improve the maps. One of the key features of OSM is its commitment to open data. The maps and associated data, typically available in XML, JSON format, or in the official OSM website are freely accessible, allowing users the flexibility to use, modify, and distribute the information freely.

OSM data encompasses a wide range of geographical elements, including nodes (points), ways (lines), and relations (complex structures), making it rich in detail and versatility. This openness and richness of data make OSM an ideal foundation for a variety of applications, from customized map creation to intricate geographic analyses, aligning perfectly with the diverse requirements of projects like ours, which seeks to leverage this data for traffic applications.

Graph databases and Neo4j

Graph databases [24] represent a specialized category of databases optimized for storing and managing data as graphs. This structure is fundamentally different from traditional relational databases that are organized into tables. In graph databases, data is structured as nodes (entities) and edges (relationships), closely mirroring real-world relationships and enabling effective traversal and querying of complex connections. This model excels in scenarios where relationships are as crucial as the data itself, allowing for the representation of intricate network interactions and dependencies.

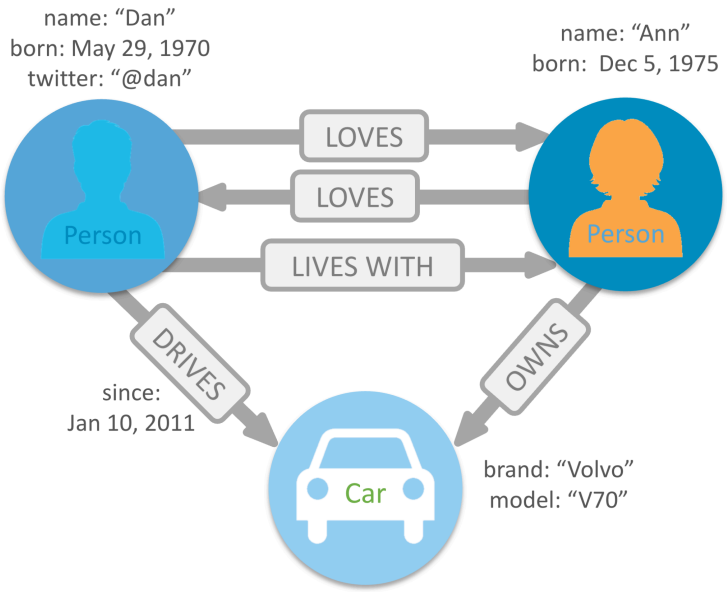


Fig. 1 Graph relationships schema

A prominent strength of graph databases is their inherent ability to model and query highly interconnected data intuitively. They facilitate the detailed representation of relationships with properties, enabling the storage of additional metadata about the connections between nodes, which proves invaluable in contexts where understanding the nuances of network interactions is paramount.

Among graph database systems, Neo4j is renowned for its leadership and widespread adoption. It offers a robust and efficient solution for managing intricate, highly interconnected data structures. Neo4j's native graph processing capabilities, augmented by its powerful query language, Cypher, make it an indispensable tool in applications where interpreting and leveraging the underlying relationships within data is key. For instance, a simple Cypher query like:

```
MATCH (a:Location)-[r:ROAD]->(b:Location) RETURN a, b, r
```

Exemplifies the intuitiveness and expressiveness of Neo4j in handling complex relationship queries, an attribute that makes it especially beneficial for the traffic simulation and analysis project. This query is designed to efficiently identify and retrieve pairs of geographical locations that are directly connected by a road, showcasing the simplicity and expressiveness of Cypher. It exemplifies the process of transforming OSM data into a structured, queryable graph format with Neo4j, which forms the foundation of traffic analysis and simulation efforts within urban areas such as Sevilla.

In the context of our project, the ability of Neo4j to handle complex queries efficiently and its powerful graph processing capabilities make it an ideal choice for transforming OSM data into a meaningful, queryable graph structure. This structure will form the backbone of our traffic simulations, enabling us to model and analyze intricate traffic flow patterns and interactions within the urban landscape of Sevilla.

State of Art

Existing solutions

In the domain of integrating geographic data with graph databases, the neo4j-contrib/osm (https://github.com/neo4j-contrib/osm) [21] open repository has served as a pioneering effort. It aimed to connect OSM data with a Neo4j graph database, presenting a novel approach to managing and analyzing geographic information within a graph database context. Despite its innovative intent, an in-depth analysis reveals several shortcomings that significantly limit its applicability and effectiveness in current traffic management and urban planning scenarios.

Firstly, the repository has not been updated since November 2021, a stagnation that raises concerns about its compatibility with the latest versions of Neo4j. The fast evolution of Neo4j's features and capabilities means that maintaining up-to-date compatibility is crucial for leveraging the full potential of the graph database technology. The lack of recent updates suggests that users may encounter compatibility issues, which could obstruct the integration of new Neo4j features and optimizations into their projects.

Moreover, the neo4j-contrib/osm project suffers from a notable lack of comprehensive documentation. Adequate documentation is crucial for any open-source project, as it facilitates understanding, adoption, and contribution by a broader community of developers and researchers. The absence of detailed documentation and usage guidelines restricts the accessibility of the project to new users and constrains its potential for community-driven enhancements and expansions.

Recognizing these limitations, this project proposes an updated and comprehensive approach to integrating OSM data with Neo4j. The proposed new implementation focuses on ensuring compatibility with the latest Neo4j versions, thereby harnessing new features and improvements that enhance performance and scalability. It is also a priority the creation of documentation, covering everything from setup and configuration to advanced usage scenarios, facilitating easier adoption.

In conclusion, while neo4j-contrib/osm laid the foundational stones for OSM and Neo4j integration. This project evolves this concept to meet the contemporary demands of traffic management and geographic information analysis.

Tools and Technologies used

In the context of this project, all scripting has been done using the programming language Python, due to the ease and versatility it provides in order to create useful and understandable scripts easily. Within the programming language different libraries have been used for the various purposes of the project such as: “neo4j” for the access to the database, “osmnx” for the retrieval and conversion into graph of the data, “matplotlib” for several graphical representations, “python-dotenv” to store securely the Neo4j Aura [20] credentials, “py2neo” to interact with the database without using Cypher, “math” for performing mathematical operations and “tkinter” for creating an interface for some script. Also some visualization tools have been built using HTML and JavaScript. And all this code has been done using the Visual Studio Code IDE.

In the process of the project two versions of Neo4j have been used. Firstly the first connections and interactions with the database were made using an instance of Neo4j Aura and after securing that connection and all the process regarding it a local version of Neo4j installed locally on Linux was used.

During the development of the project two operating systems have been used. For the first part of the project doing the connection with Neo4j Aura a machine running Windows 11 was used. However to perform all the local deployment of the database the same machine was used but running Ubuntu 22.04 LTS due to the limitations that Windows presents. Like file permission management and access control, which is more restrictive on Windows, complicating the setup and efficient management of a local database.

All the code generated for this project is stored in GitHub in two different repositories, MarioArocaPaez /OSM-to-Neo4j [10] for the Windows and Neo4j Aura

connection and MarioArocaPaez/neo4jLinuxConnection for the Linux local connection. Not only the code is stored there but has all the information regarding updates, insertion and deletion of code through the time that the project has been developed, all information regarding commits can be publicly accessed.

Finally this project report has been written using Google Docs and all communication between tutor and student has been made through in person meetings or by email using Microsoft Outlook.

Methodology

1. Extracting data from OpenStreetMap

Extracting a map from OpenStreetMap is not the easiest task on the internet. Firstly if a user wants to do so in the official webpage he will encounter the message: You requested too many nodes (limit is 50000). Either request a smaller area, or use planet.osm [1]. The main webpage has a very small limitation for the size of the maps that can be downloaded which is significantly limiting for the objective of this project due to the fact that with these restrictions in the webpage it is not even possible to extract a map of downtown Sevilla.

The first suggested option is to download it in planet.osm which is a web page containing copies of the official OSM database. However the main issue with the maps found in this site is that most of them are maps containing information of the whole world with weights up to 135 GB. Meaning it would be an inconvenience for the project having to store such large files and with so much irrelevant data, if the aim of the project is centered around the city of Sevilla there is no point in having data from the whole word.

After some investigation the most viable option to extract a map from OpenStreetMap is overpass turbo [2] which is a web interface to interact with the OSM database through the Overpass API which is an API designed to interact with the OSM database. Nevertheless, using this tool would mean manually downloading a .osm file and having to apply the transformations to convert it into a graph in order for it to be able to transform into a neo4j database. This is where OSMnx [3] comes, being a Python package to easily download, model, analyze, and visualize street networks and other geospatial features from OSM. This package allows the easy retrieval and manipulation of the maps without having to manually download the maps and insert them into the project. Another property of this package that suits this project is the function *.graph_from_place()* which is ideal for converting any map into a graph in order to transform it into neo4j more easily.

2. Converting the data into Graph

In order to extract the data using the OSMnx package it is only necessary to call the already mentioned method with the desired location, in this case the city of Sevilla. It is important to include in this method the *netwok_type="drive"* statement, meaning that the graph will consist of the roads that can be driven with a car, there are several options that can be used as walking paths, bike paths or all of them at the same time. So as this project is centered around traffic, only car driven roads will be included in the final graph.

```
# Search OpenStreetMap and create an OSMNx graph
G = ox.graph_from_place("Sevilla, Andalucía, España", network_type="drive")
```

The only way to check the quality of this graph is to actually see it and compare it with the original. It is easy to actually see these graphs using the famous matplotlib [4] python package, combining it with the OSMnx function plot_graph(). It returns a tuple with the figure of the graph and its axes making it easier to plot with the matplotlib method *show()*. The first test was done with Martín de la Jara, a small village inside the community of Sevilla.

Below it is visible the plotted graph retrieved when this operation is performed with Martín de la Jara.

```
# Search OpenStreetMap and create an OSMNx graph
G = ox.graph_from_place("Martín de la Jara, Sevilla", network_type="drive")

# Plot the first graph
# ox.plot_graph(G) returns a tuple of two objects: the figure and the axes
fig, ax = ox.plot_graph(G)
plt.show()
```

Graph:

Fig. 2 OSM Graph from Martín de la Jara with Matplotlib

Original OpenStreetMap:

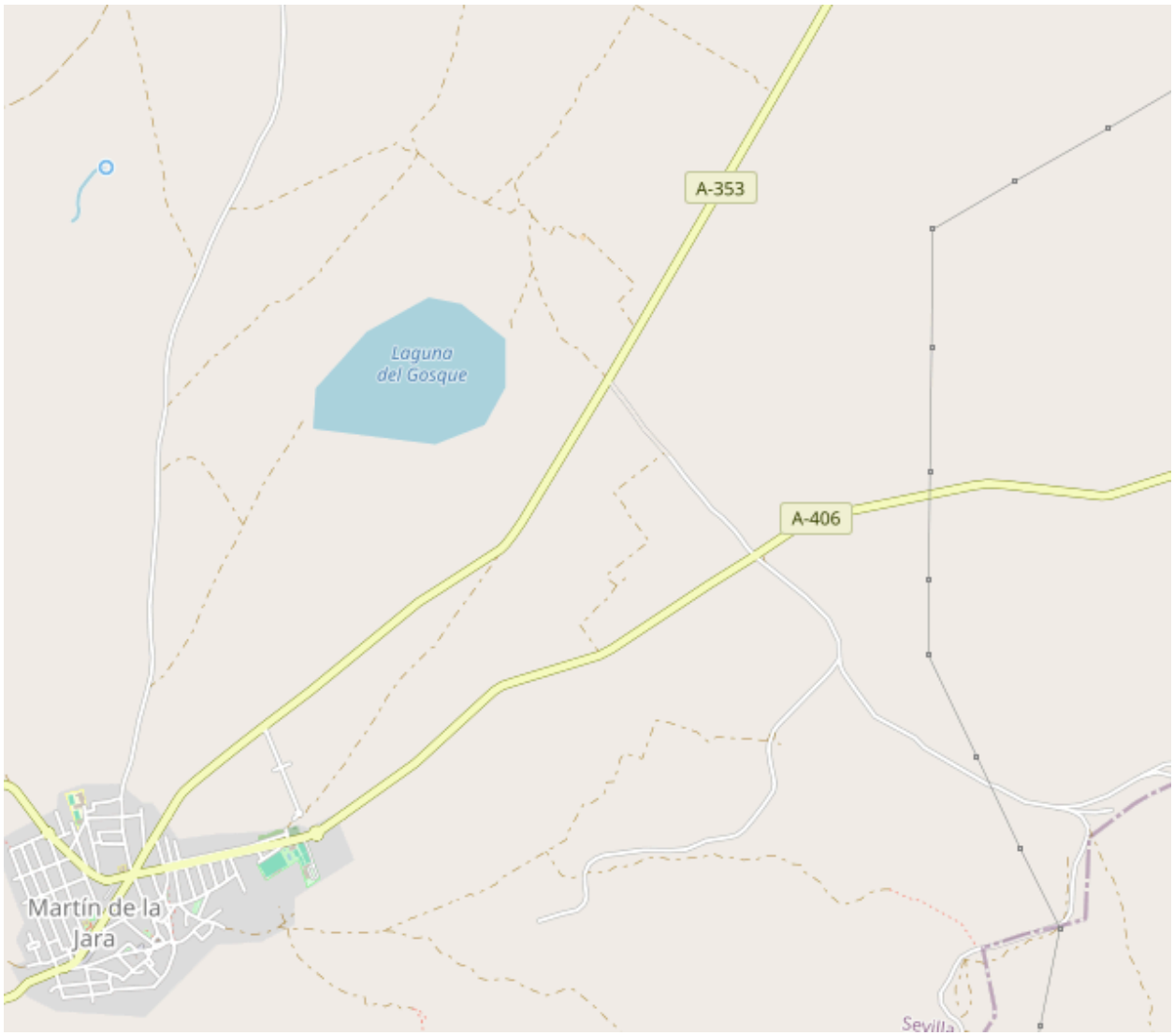


Fig. 3 Martín de la Jara in OSM

And the second one with Estepa, a slightly bigger town. It was important at this point to use relatively small locations in order to detect with plain sight that the graph was correct, comparing it with the original map.

Graph:

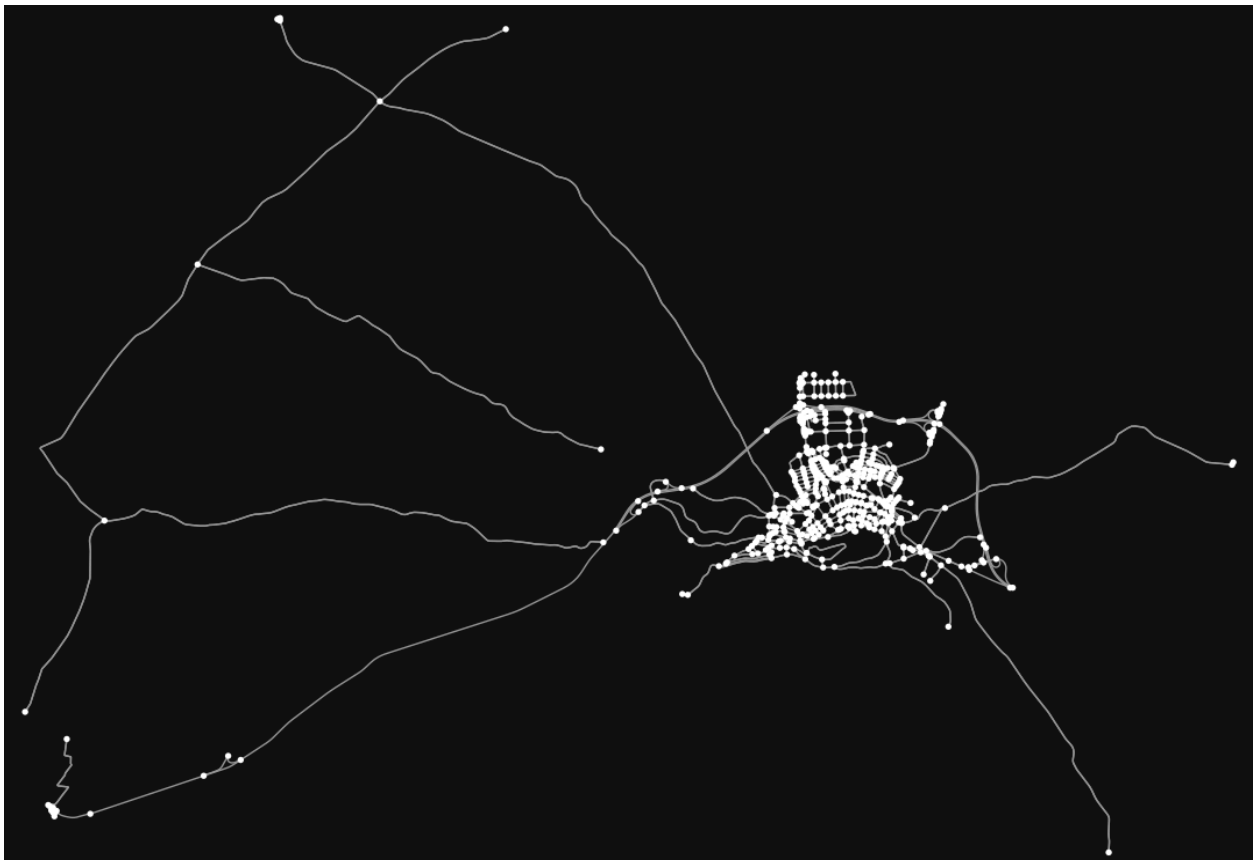


Fig. 4 OSM Graph from Estepa with Matplotlib

Original OpenStreetMap:

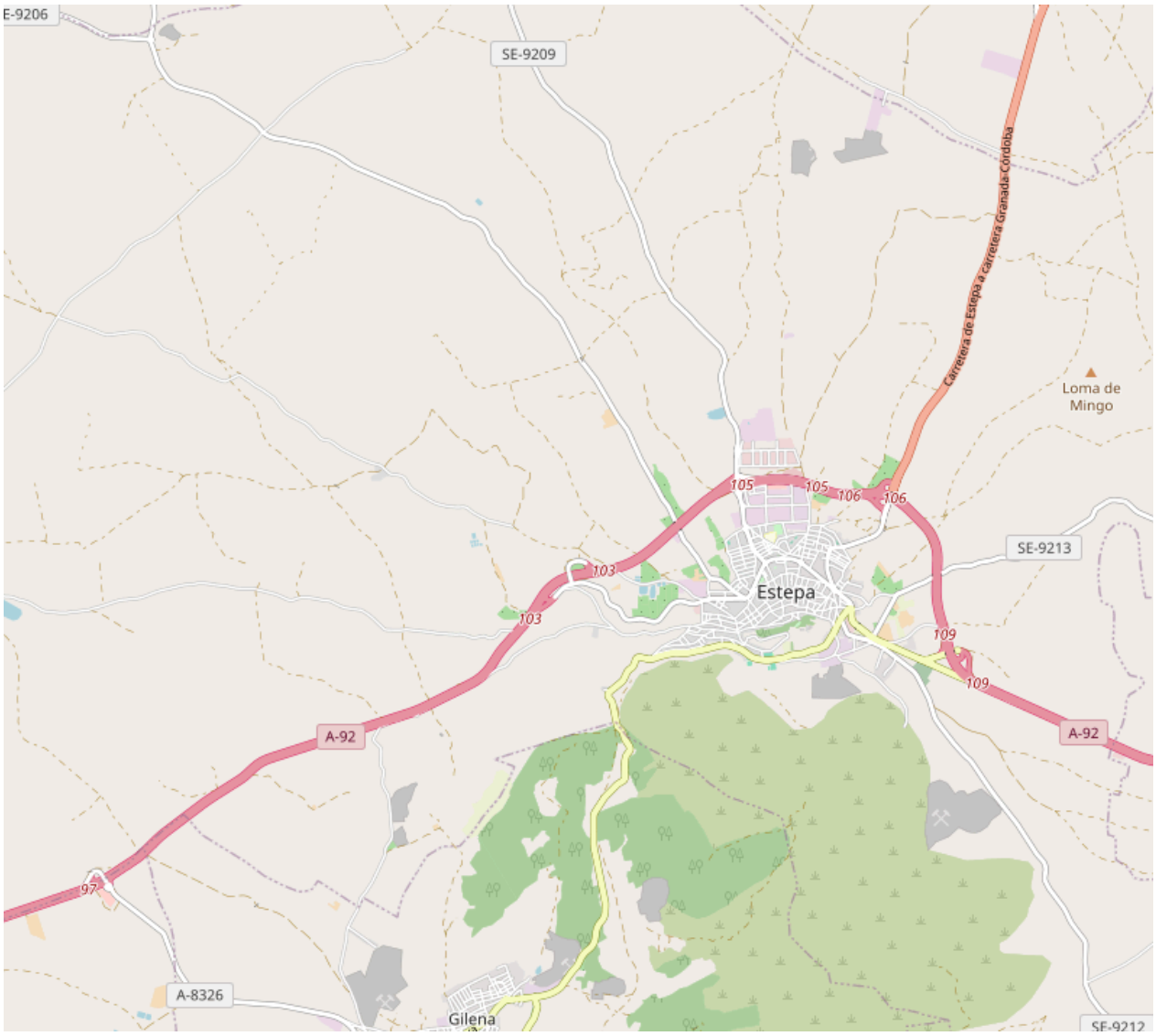


Fig. 5 Estepa in OSM

It is clear the quality of the graph compared to the original map, so the tests with the city of Sevilla started. Firstly as it is a significantly bigger graph it was not easy to see if everything matched but zooming to different neighborhoods and zones the quality of the graph was clear.

Sevilla Graph:

Fig. 6 OSM Graph from Sevilla with Matplotlib

Sevilla OpenStreetMap:

Fig. 7 Sevilla in OSM

Downtown Sevilla Graph:

Fig. 8 OSM Graph from Downtown Sevilla with Matplotlib

Downtown Sevilla OpenStreetMap:

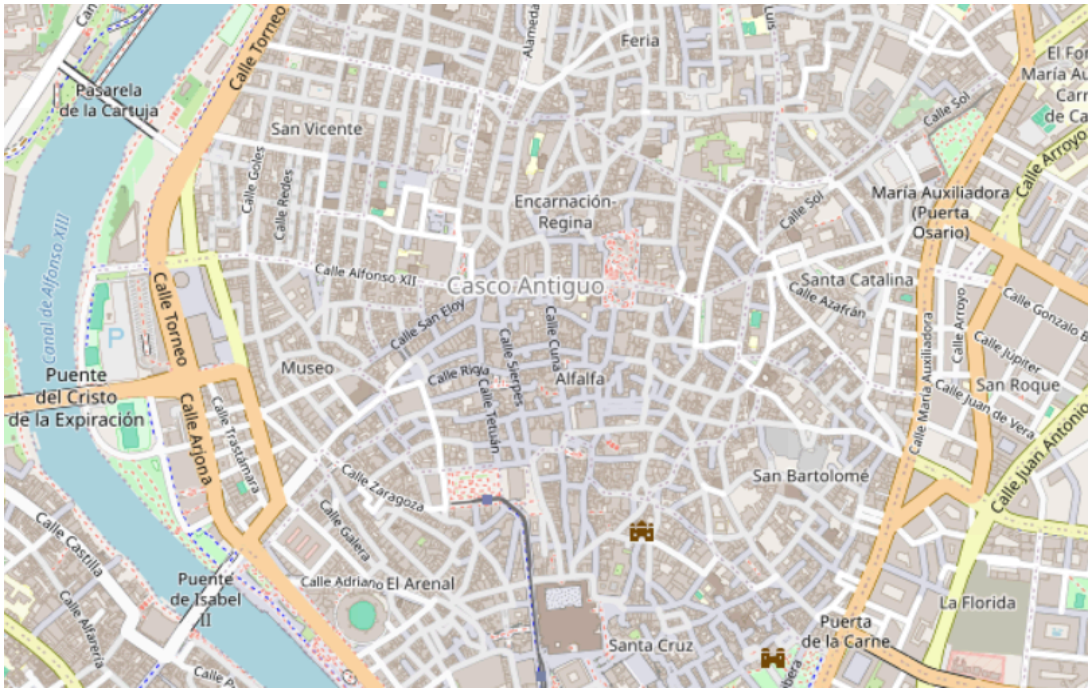


Fig. 9 Downton Sevilla in OSM

Triana Graph & OpenStreetMap:

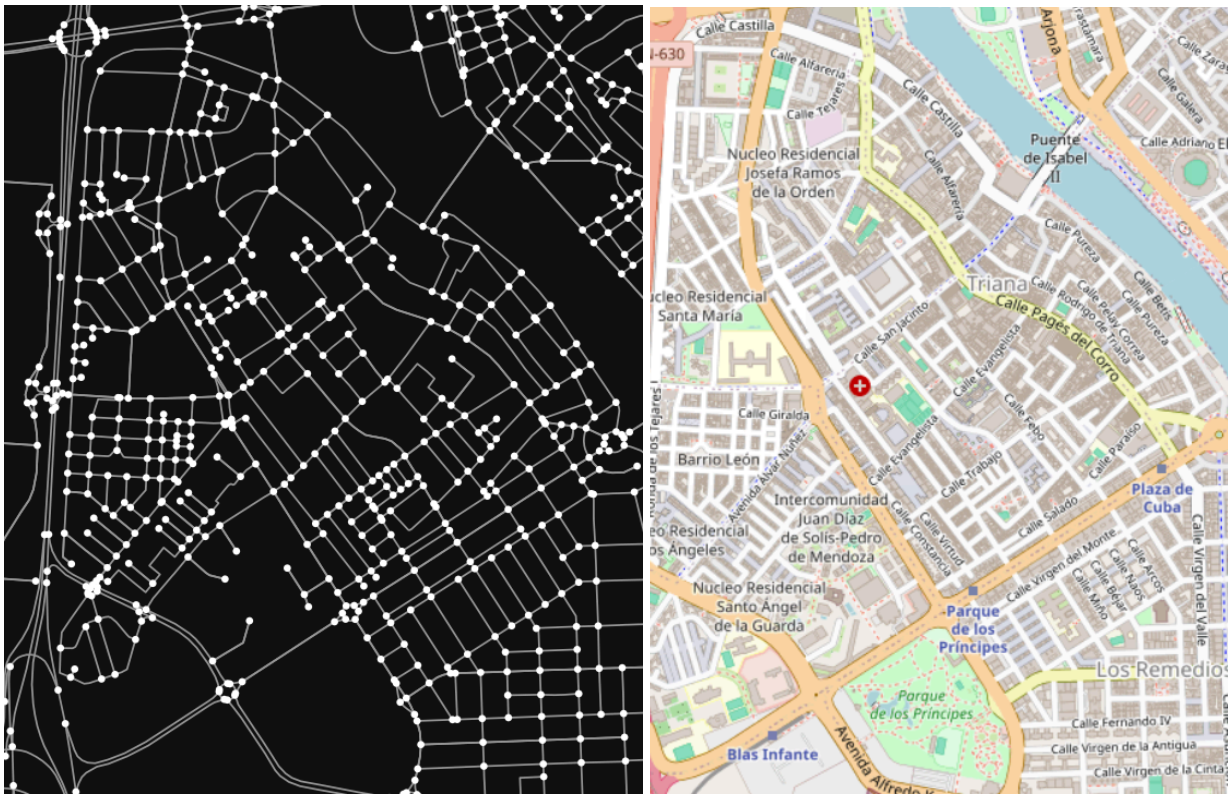


Fig. 10 OSM Graph from Triana with Matplotlib VS Triana in OSM

Consequently this methodology for transforming the maps into graph format provides great quality at the result and it is a suitable way to proceed for this project.

3. Structure of the graph

Having already built the graph it is necessary to explain how they are structured. Typically a graph consists of edges, vertices and relationships. A clear example is the famous Seven Bridges of Königsberg by Euler. Königsberg was a city in Prussia divided by 2 pieces of land and 2 islands. These 4 lands were connected together by 7 bridges. The original problem consisted in trying to walk through all the bridges without crossing more than 1 time though any of them. The problem itself is irrelevant for this project, but what is interesting is how this problem is transformed into a graph.

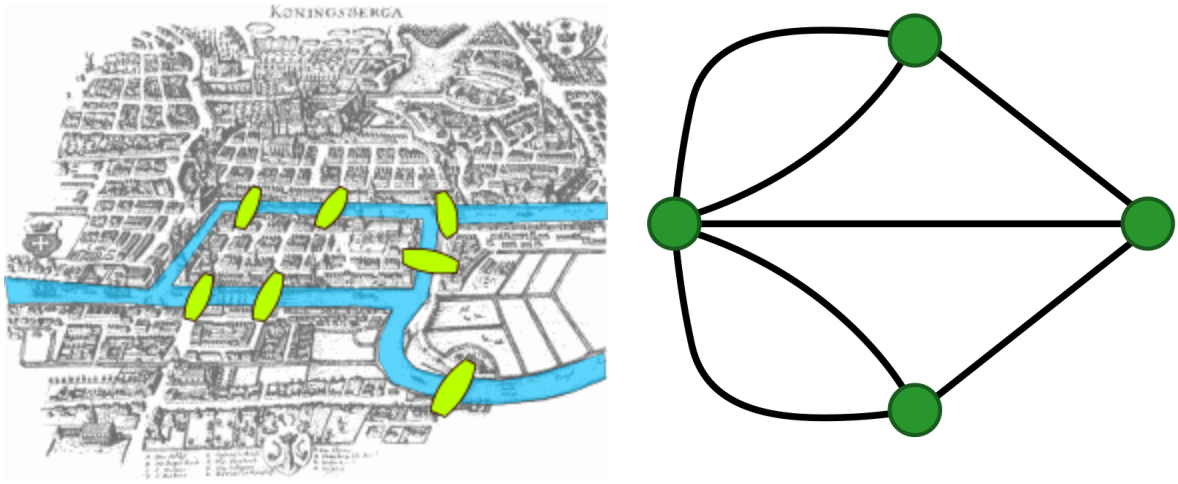


Fig. 11 Bridges of Königsberg

Converting the pieces of land into the vertices, the bridges into the edges and the connection between them into relationships resulting in a perfect representation of a graph. A similar approach is taken by OSMnx to turn the maps into graphs.

Firstly it is important to note that as explained in the above section this process is only done with the streets that can be driven by car, so the map that is going to be converted only has roads, then what it does is to convert the different streets into edges and the intersections between them into the edges. As an example here 3 streets can be seen there is Avenida Pablo Iglesias and Calle San Juan Bosc and Calle Urquiza which are intersections into Avenida Pablo Iglesias.

Fig. 12 Intersection in OSM

This is simply transformed as 3 vertices and 2 edges. The 3 vertices being the mentioned streets and the edges the intersections between the streets.

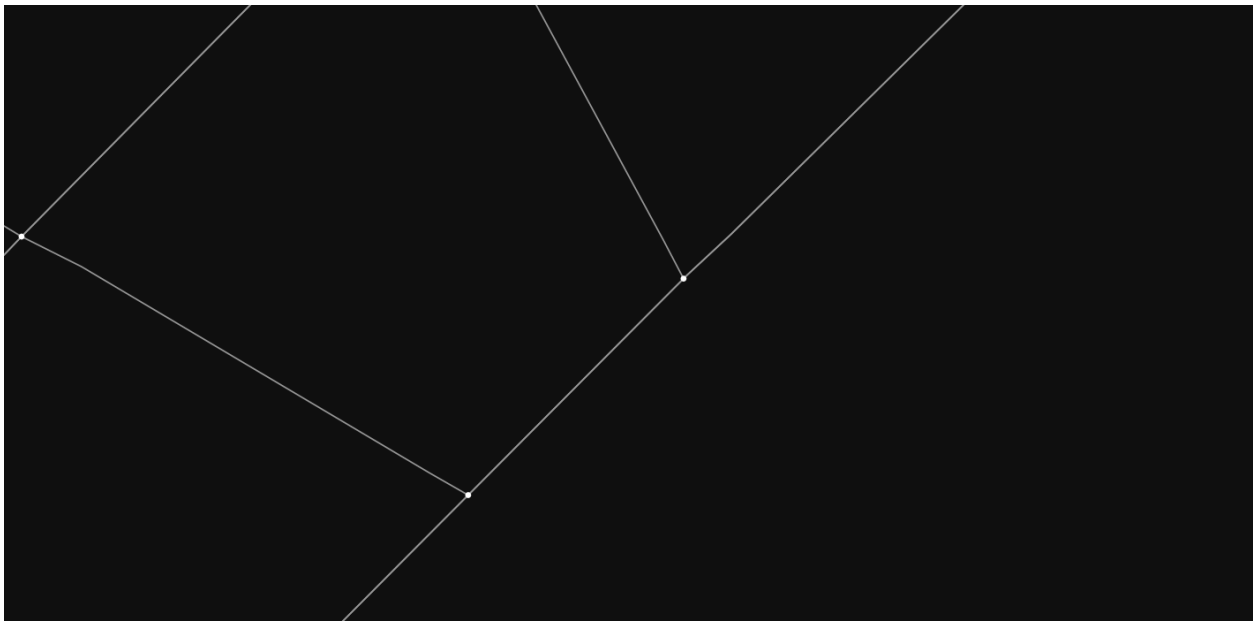


Fig. 13 Intersection in OSM Graph with Matplotlib

Another important point is how roundabouts are handled given that they have several intersections in a small portion of terrain. Here there is an example with the roundabouts at the entrance of Triana:

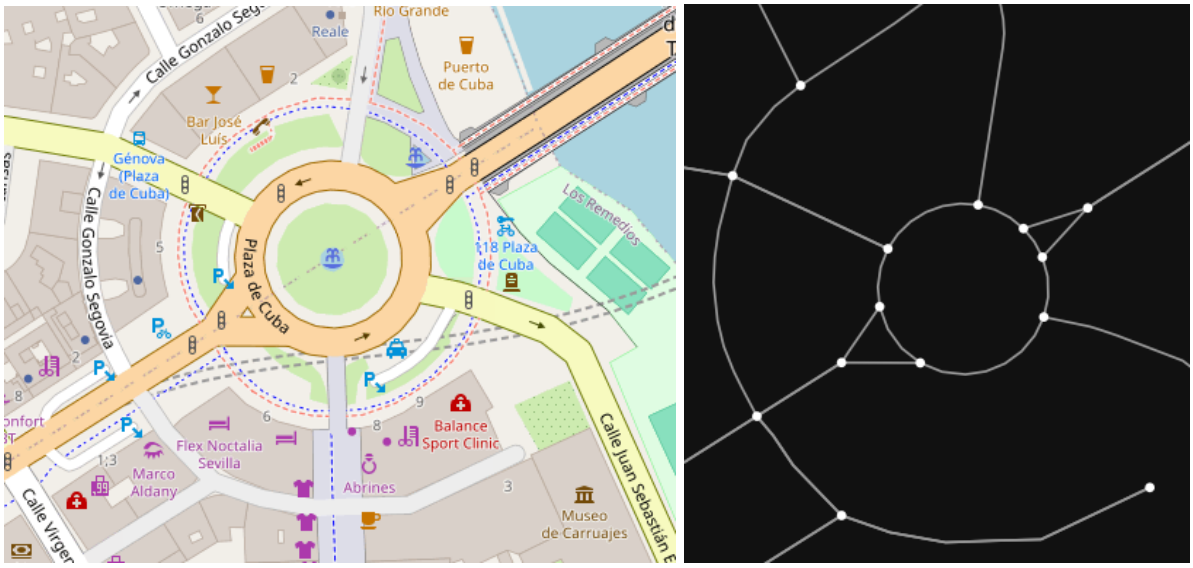


Fig. 14 Fig Triana roundabout in OSM VS OSM Graph with Matplotlib

In the accompanying image, the structural transformation of a roundabout into a graph is depicted. The roundabout, a complex intersection with multiple entry and exit points, is simplified into a series of vertices and edges. Each entry and exit acts as a vertex, while the connections between them are represented as edges. This simplification into a graph enables us to apply various algorithms for network analysis, traffic flow optimization, and route planning.

To conclude, the conversion of physical urban layouts into graph structures allows for a more abstract yet powerful representation of transportation networks. By utilizing the capabilities of OSMnx, the detailed and often cumbersome map data is distilled into a graph that faithfully represents the connectivity and structure of the area's road network. Through this process, one can analyze and understand complex urban environments more effectively. The graph becomes a tool not just for visualization, but also for the application of mathematical models that can address real-world problems related to urban planning and traffic management. In essence, the graph is not just a representation; it is a foundational element for deeper insights into the dynamics of urban spaces.

4. Neo4j Importation

To start with, the initial neo4j connection was established using neo4j Aura DB [15] which is a fully-managed cloud graph database service offered by Neo4j. It's designed to provide users with a zero-administration, always-on database service in the cloud. In order to use it it is necessary to sign in with an email, and create an instance for a database. For the free available version it is only possible to have 1 instance, that is why this is not the definitive Neo4j database that will be used. Once this is done the user will have a uri corresponding to the database, a user and a password. This is what will be used to interact with the database using Python.

Firstly in order to connect with Neo4j using Python it is necessary to install the Neo4j python library, then the connection is done using a driver that requires the uri and the credentials.

```
# Create a Neo4j driver
driver = neo4j.GraphDatabase.driver(NEO4J_URI, auth=(NEO4J_USER, NEO4J_PASSWORD))
```

Once this is done, the steps to be followed are the same as for the representation, the graph is fetched and the nodes and relationships are stored. It is important to reset the indexes of both nodes and relationships. This operation renumbers the index from 0 to the length of the DataFrame minus one, ensuring that the index is a continuous range of numbers. This can be helpful if the original data had a different indexing scheme that might not be sequential or if some rows were removed, leading to gaps in the index.

```
# Search OpenStreetMap and create an OSMNx graph
G = ox.graph_from_place("Sevilla, Andalucía, España", network_type="drive")

gdf_nodes, gdf_relationships = ox.graph_to_gdfs(G)
gdf_nodes.reset_index(inplace=True)
gdf_relationships.reset_index(inplace=True)
```

It is important to understand exactly which data has been passed to these 2 variables by performing this operation. Firstly, gdf_nodes contains information about the nodes in the graph, where each node typically represents an intersection or endpoint in the road network. The columns in gdf_nodes include:

- **osmid**: The OpenStreetMap ID of the node.
- **y**: The latitude of the node.
- **x**: The longitude of the node.
- **geometry**: A POINT geometry representing the location of the node.
- **highway**: If the node is part of a highway, this attribute indicates its type or significance.
- **ref**: Reference numbers or codes associated with the node, if any.
- **street_count**: The number of streets (edges) that connect to this node, indicating its connectivity within the network.

And then gdf_relationships contains information about the edges in the graph, where each edge represents a segment of the road. The columns in gdf_relationships include:

- **u, v**: The OpenStreetMap IDs of the start and end nodes of the edge, respectively.
- **key**: A key that differentiates multiple edges between the same pair of nodes (if such exist).
- **osmid**: The OpenStreetMap ID(s) of the edge. If an edge is composed of multiple OSM ways, this can be a list.
- **name**: The name of the roadway or path, if available.
- **highway**: The type or classification of the road or path.
- **oneway**: A boolean indicating whether the road segment is one-way or not.
- **length**: The length of the road segment in meters.
- **geometry**: A LINESTRING or MULTILINESTRING geometry representing the shape of the road segment.
- **lanes**: The number of lanes in the road segment.
- **maxspeed**: The maximum speed limit of the road segment.
- **ref**: Reference numbers or codes associated with the road segment, if any.

It is important to note that thanks to the u and v, the roads will include their direction even if they are only one direction or both directions.

That would be everything for the retrieval of the data. After that comes the moment to create the database, it is important for the creation of the database to have some constraints and indexes to ensure a stable, well-structured, and performant database. In this case, three constraints/indexes are created. Also nodes are defined as “Intersection” and edges as “ROAD_SEGMENT” for Neo4j. The first constraint ensures that there are no repeated nodes by their osmid (OpenStreetMap Identifier), which must be unique. This avoids any duplication of nodes in the database and ensures the integrity of the data. The second index is created on the osmids property of each ROAD_SEGMENT. It's not a traditional constraint but an index that, if it does not exist, is created to expedite Neo4j operations that refer to that ID. This index is especially useful for quickly accessing and managing relationships, making operations like matching and merging road segments based on their OSM IDs more efficient. The third constraint/index, the point index, is a specialized index for spatial data. It's created on the location property of Intersection nodes. This point index is essential for performing efficient spatial queries. It optimizes the database for operations that involve geographical data, such as calculating distances between points or finding the nearest intersection to a given location. By indexing the location property of intersections, this point index allows Neo4j to quickly execute spatial operations, leveraging the geospatial capabilities of the database to their fullest. This is particularly beneficial for traffic and urban planning applications, where understanding the spatial relationship between various elements is crucial.

```
# Define Cypher queries to create constraints and indexes
constraint_query = "CREATE CONSTRAINT IF NOT EXISTS FOR (i:Intersection) REQUIRE i.osmid IS UNIQUE"
rel_index_query = "CREATE INDEX IF NOT EXISTS FOR ()-[r:ROAD_SEGMENT]-() ON r.osmids"
point_index_query = "CREATE POINT INDEX IF NOT EXISTS FOR (i:Intersection) ON i.location"
```

Once everything is settled it is the moment to import the data. This will be done by using 2 Cypher queries. The first for the nodes will iterate over the list of nodes and all of its data ``gdf_nodes”. It takes all rows with a valid osmid and creates a node containing.

- A location that is created by neo4j with the point() function, which constructs a geographic point from the given latitude and longitude.

- The reference number associated with the node.
- The type of path.
- The number of streets (edges) connecting into the node.

```
node_query = '''
    UNWIND $rows AS row
    WITH row WHERE row.osmid IS NOT NULL
    MERGE (i:Intersection {osmid: row.osmid})
        ON CREATE SET i.location = point({latitude: row.y, longitude: row.x }),
            i.location = point({latitude: row.y, longitude: row.x }),
            i.ref = row.ref,
            i.highway = row.highway,
            i.street_count = toInteger(row.street_count)
    RETURN COUNT(*) as total
'''
```

And a similar query will be used in order to import the edges from "gdf_relationships". With the difference that as the edges are created they will be linked to their source and target node, that is why the nodes must be imported first. By executing this query the edges will contain:

- Source (u) and destination (v).
- One way or not.
- Number of lanes.
- Name of the street.
- Type of road (field named highway).
- Maximum speed in Kilometers per hour.
- Length of the edge, which in most cases is not the total length of the street, is the length between one intersection and the next one.

```
rels_query = '''
    UNWIND $rows AS road
    MATCH (u:Intersection {osmid: road.u})
    MATCH (v:Intersection {osmid: road.v})
    MERGE (u)-[r:ROAD_SEGMENT {osmid: road.osmid}]->(v)
        SET r.oneway = road.oneway,
            r.lanes = road.lanes,
            r.ref = road.ref,
            r.name = road.name,
            r.highway = road.highway,
            r.max_speed = road.maxspeed,
            r.length = toFloat(road.length)
    RETURN COUNT(*) AS total
'''
```

In order to insert all this data efficiently, a batching technique is used, which involves dividing the entire dataset into smaller, manageable groups or "batches." This method significantly enhances the performance and reliability of data importation by avoiding the common pitfalls associated with handling large volumes of data at once, such as system overload or transaction timeouts.

The function takes the entire dataset and splits it into smaller subsets, each containing a specified number of rows (for example, 10,000 rows per batch). This makes it easier to manage memory usage and keeps the import process running smoothly. For each subset of data (or batch), the function executes the Cypher query to insert that batch into the Neo4j database. By doing this in steps, it is ensured that the database is not overwhelmed with too much data at once.

The function keeps track of how many batches have been processed and provides feedback on the success of each batch. This is helpful for monitoring the import process and quickly identifying any issues that may arise. It repeats this process, moving through the dataset batch by batch, until all data has been successfully imported into the database.

```
def insert_data(tx, query, rows, batch_size=10000):
    total = 0
    batch = 0

    while batch * batch_size < len(rows):
        results = tx.run(query, parameters={'rows': rows[batch*batch_size:(batch+1)*batch_size].to_dict('records')}).data()
        print(results)
        total += results[0]['total']
        batch += 1
```

Finally the Cypher queries will be run using the neo4j drives as follows. When importing the nodes and edges the geometry field has been dropped because it has no relevance with this project.

```
# Run our constraints queries and nodes GeoDataFrame import
with driver.session() as session:
    session.execute_write(create_constraints)
    session.execute_write(insert_data, node_query, gdf_nodes.drop(columns=['geometry']))

# Run our relationships GeoDataFrame import
with driver.session() as session:
    session.execute_write(insert_data, rels_query, gdf_relationships.drop(columns=['geometry']))

driver.close()
```

Once all the processes are executed, the database can be visited. In this case everything in this section will be shown with Neo4j Aura, since it is the one mentioned at the beginning.

When in the database, it recognises 10.234 nodes and 20.266 edges, which shows the success of the importation process. Some queries can be done to see the content of the database, or the different data that it contains. For example one query can be performed to see the content of the whole database (although Neo4j has a limitation of 1000 nodes for visualization for free users).

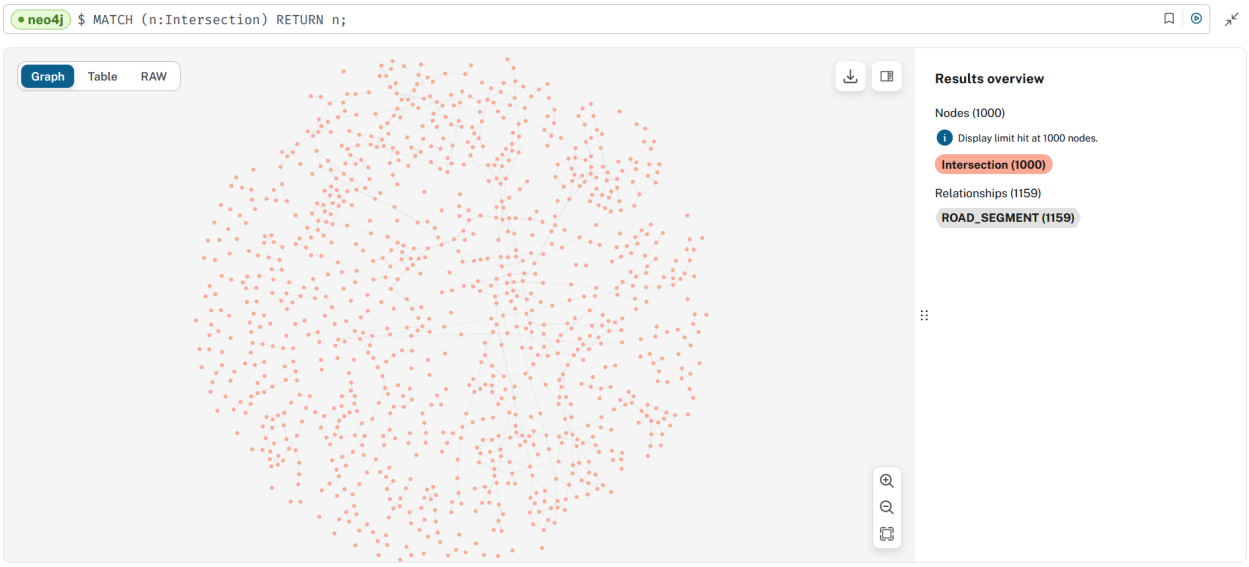


Fig. 15 Sevilla Graph viewed from Neo4j Aura app

When selecting a node like in figure 16 the different imported data about it from Sevilla can be seen:

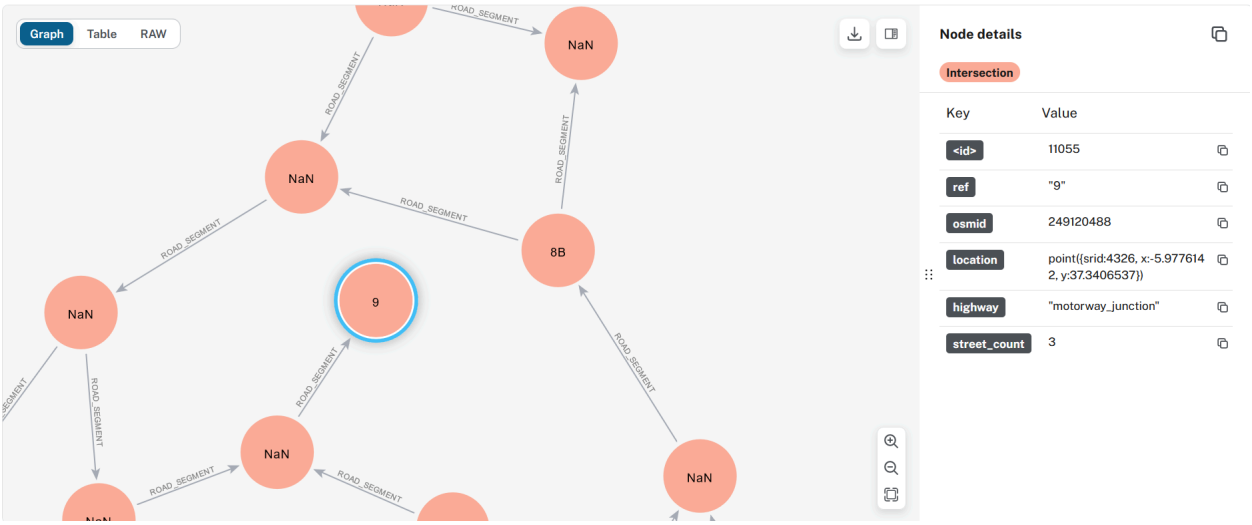


Fig. 16 Node details in Neo4j Aura

And the same thing with edges:

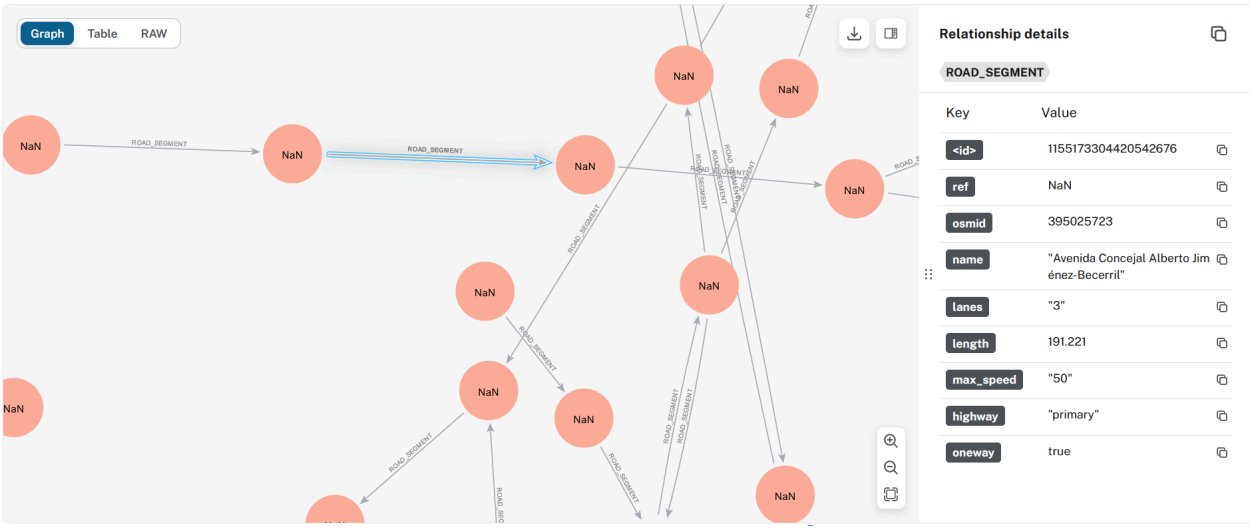


Fig. 17 Edge details in Neo4j Aura

Neo4j Aura also provides an Explore view with more options for visualization for the graph, one of these is the Coordinate layout that thanks to the coordinates that each node contains can depict a graph with the same shape of the city. In theory this view has a visualization limit of 10000 nodes but by selecting all nodes and left clicking to expand, the whole graph can be seen. Unfortunately the rendering speed inside the application will not be very good and it will work slowly.

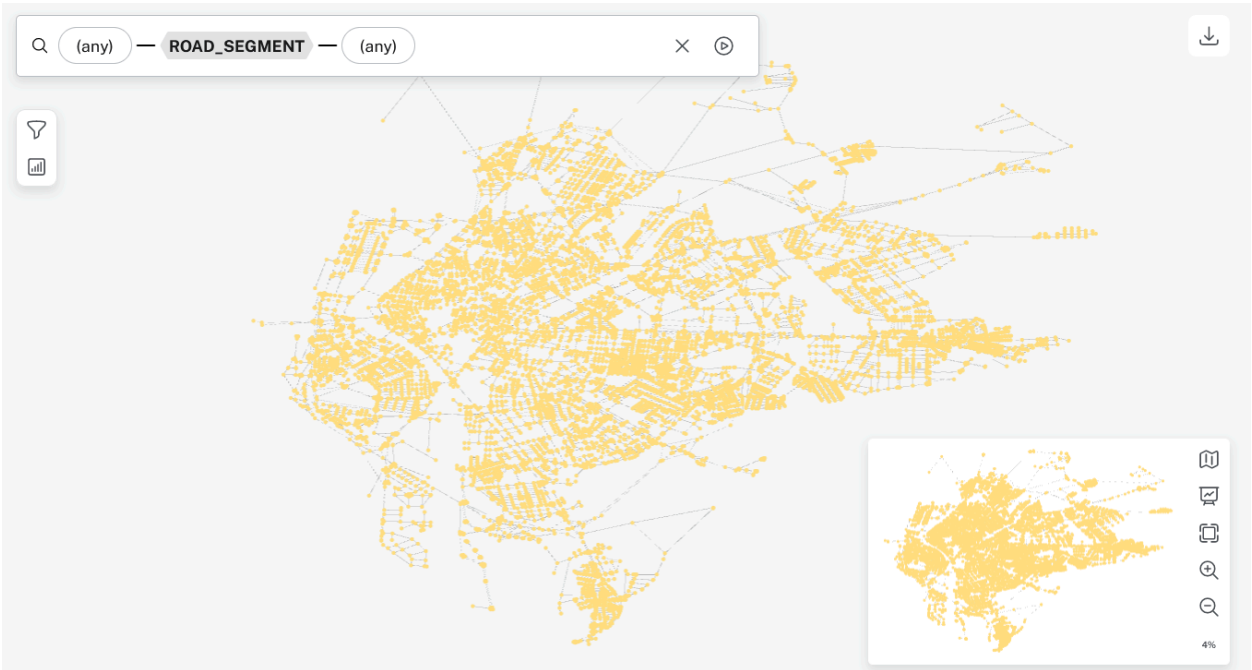


Fig. 18 Sevilla Graph viewed from Neo4j Aura Explore view

5. Performing Cypher Operations

As already mentioned Cypher is the default language used for interacting with Neo4j databases. The main objective of this project aligns with not using it but for the first tests it is really useful. The most commonly used method is shortestPath() to compute the shortest path between nodes from the database. For instance the following Cypher query computes the shortest route between two nodes and also returns the names of the streets that it goes through:

```
MATCH path=shortestPath((startNode)-[edges:ROAD_SEGMENT*]->(endNode))

WHERE ID(startNode) = 653 AND ID(endNode) = 692

RETURN path, [rel IN relationships(path) | rel.name] AS edgeNames
```

If that is executed in the Neo4j console it returns the path, but it is not really visible and cannot really give too much visual information about the path:

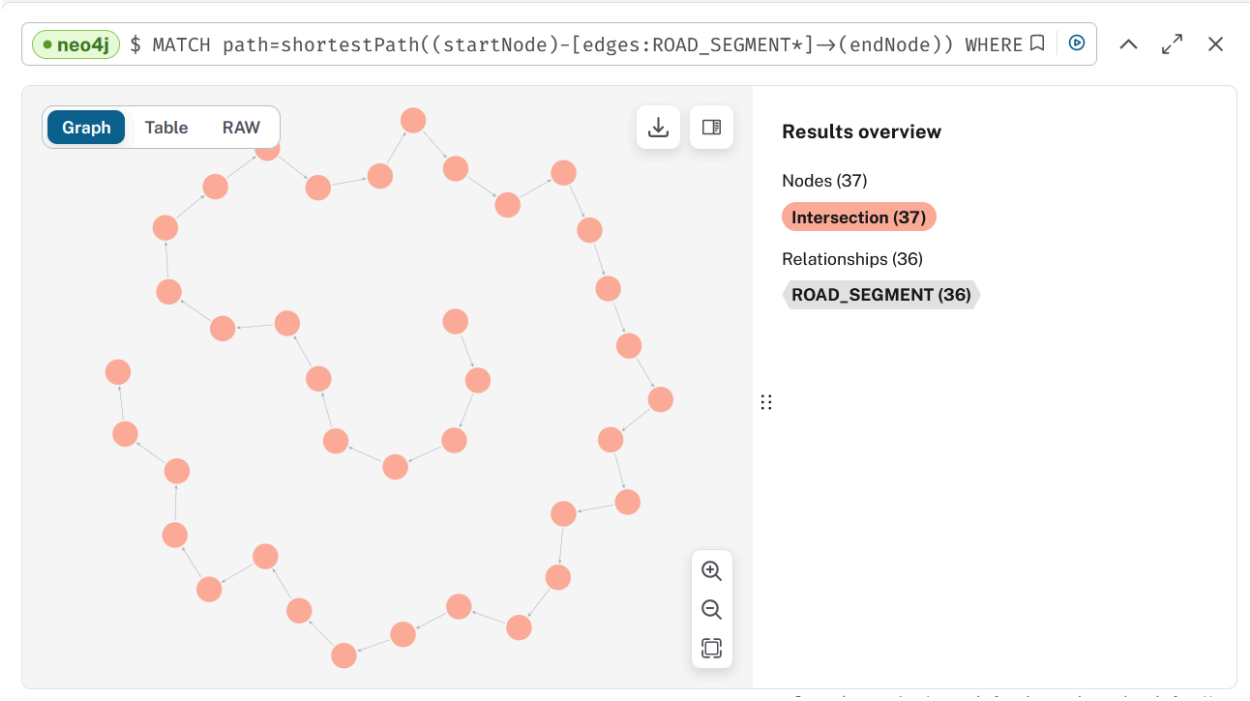


Fig. 19 Route from Cypher query in Neo4j Aura console

That is why after successfully performing this operation a new script was created. This script was used for calculating this route through Neo4j but visualizing the route using matplotlib with the graph of the city similarly to section 2 but coloring the route. The steps to execute the Cypher query are the same as in the last section and then it is possible to color the map with that route, having that the query performed before can result into the following route:

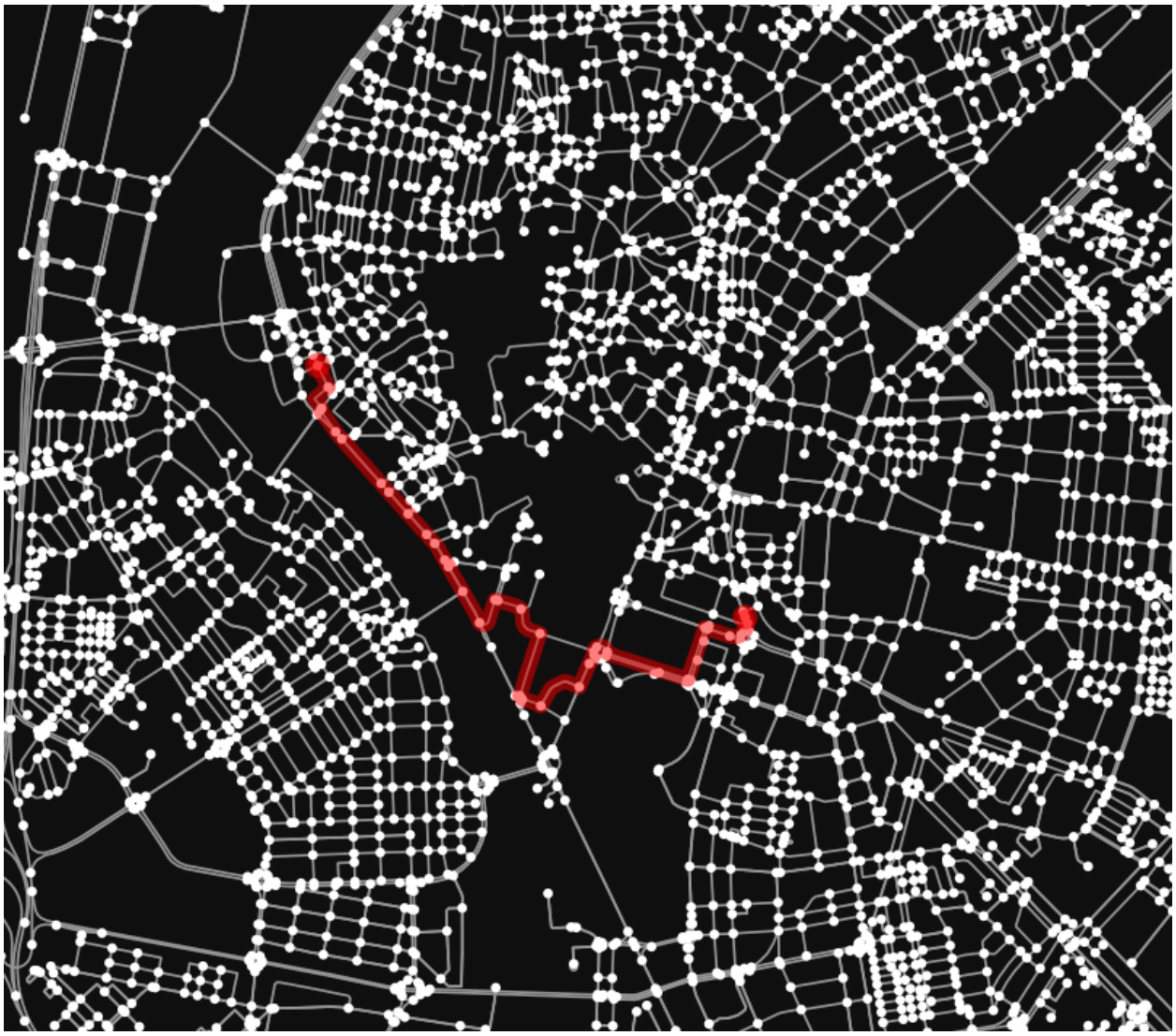


Fig. 20 Route represented in Matplotlib

Which is much more convincing than the result in the Neo4j console and proves that the operations that are being performed inside the city do have some sense.

However there is one problem with the query used before and it is that it does not have any weight configured, so if the weight is not specified Neo4j just tries to compute the route with the least number of nodes, which can not be the best route. In this case as we are working with roads the ideal weight is the length of the roads in meters so the shortest path is actually the shortest one. This can be done in Cypher by using a query like this one:

```
MATCH path=shortestPath((startNode)-[:ROAD_SEGMENT*]->(endNode))

WHERE ID(startNode) = 1716 AND ID(endNode) = 5260

WITH path, [rel IN relationships(path) | rel.name] AS edgeNames, REDUCE(s = 0, x IN
relationships(path) | s + x.length) AS totalDistance

RETURN path, edgeNames, totalDistance
```

Which gives a more optimized route for finding the shortest path (Take into account that this example route is a different one from the previous showed)

Fig. 21 Weighted route in Matplotlib

6. Running Neo4j Locally on Linux

Having already the database settled and working with everything that is needed for this project is a great step on the objective of the work, however it cannot be considered finished since it is using directly the services from Neo4j meaning that any update or change made by Neo4j can change all the behavior of the project and break it. The main idea is to use the technology from Neo4j but using the least possible services from them so this project stays the least dependent on how Neo4j evolves. That is why it is surely important to adapt this solution to a locally running Neo4j database in Linux.

Installing Neo4j on linux is not the easiest task for newcomer users to Linux. At least in Debian based LInux distributions package repositories do not contain a copy of the Neo4j database engine so it has to be added manually. Everything related to the installation, configuration and usage of Neo4j in LInux will be explained in detail in the Deployment section.

Once everything is installed and working Neo4j will be listening using the bolt [12] protocol at *bolt://localhost:7687* and the user will be able to interact with the database directly at *http://localhost:7474/browser/*

For the first connection a test database was used with simple nodes labeled as Greeting and a hello world message. The connection is similar to the one done in Neo4j Aura but as there is not authentication required (This is not by default it has to be done manually and it is explained in the Deployment section) when setting the driver for establishing the connection auth is set to None

```
self.driver = GraphDatabase.driver(uri, auth=None)
```

and instead of using the credentials that were required in Aura the connection is just done by calling using the bolt protocol to the port 7687 (by default it uses port 7687 for the

```
greeter = HelloWorldExample("bolt://localhost:7687")
```

connection and port 7474 for the graphical interface to interact with, but it can easily be changed if ports are occupied).

The test database just contains nodes called Greeting and a message that can be manually written, in this case it is hello, world

```
# and return the message along with the node's id.
result = tx.run("CREATE (a:Greeting) "
                "SET a.message = $message "
                "RETURN a.message + ', from node ' + id(a)", message=message)
```

```
greeter.print_greeting("hello, world")
# Close the database connection
```

If the script is run there should not be any problem and the database should be created. It is indicated in the consolo but can also be seen in port 7474.

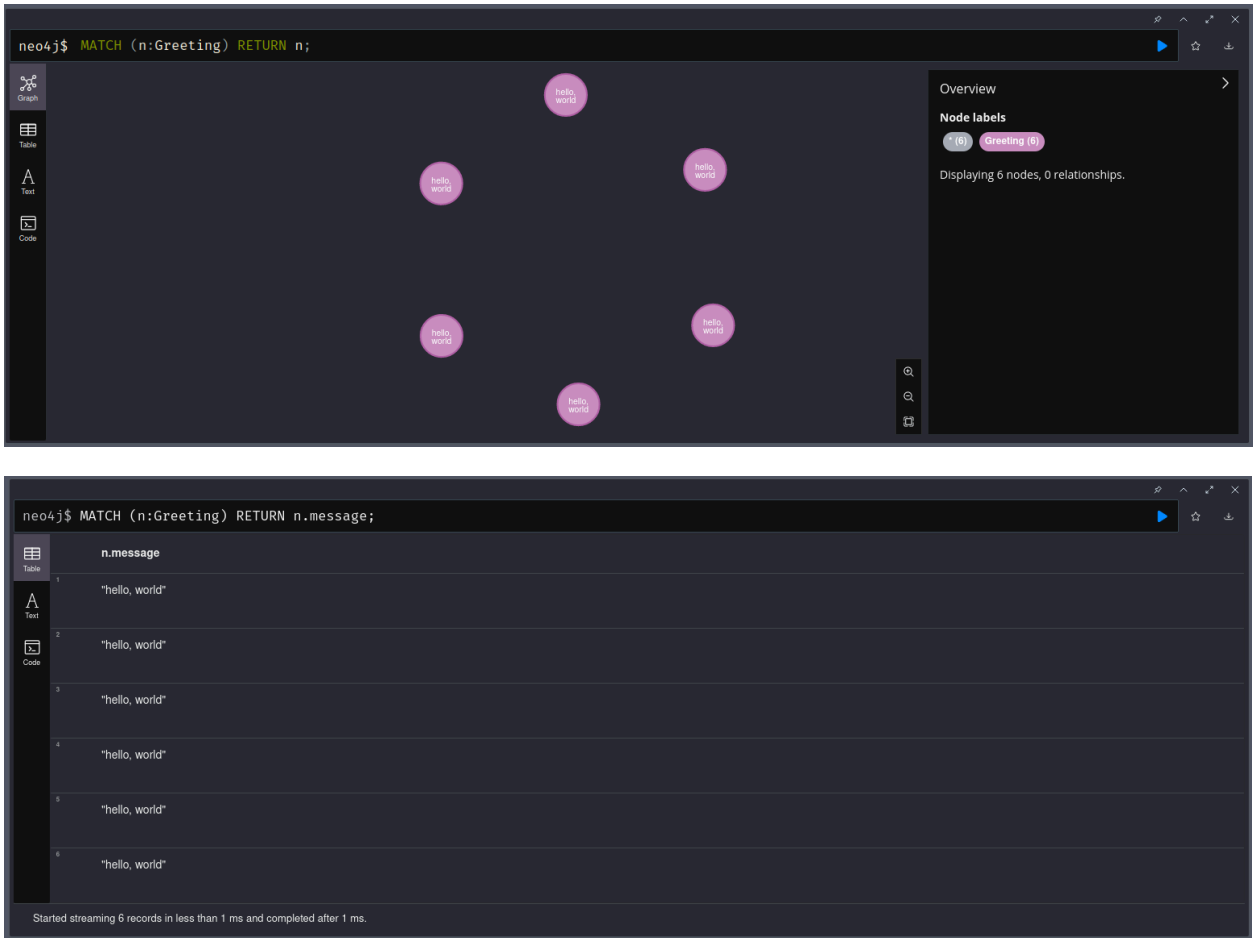


Fig. 22 Basic graph in Local Neo4j console

After checking the success of this operation the original script to fetch the map from OSM, transform it into a graph and insert it into Neo4j was adapted to work in this environment. As mentioned before, changing the connection uri and adapting it to non authentication required.

```
# Local bolt connection
NEO4J_URI = "bolt://localhost:7687"

# Neo4j driver with no auth
driver = neo4j.GraphDatabase.driver(NEO4J_URI, auth=None)
```

When executed there should be no problem and the database should be running. Some Cypher queries can be perform in order to test the well functioning of it:

Seeing the whole graph (Visualization limit of 300 nodes):

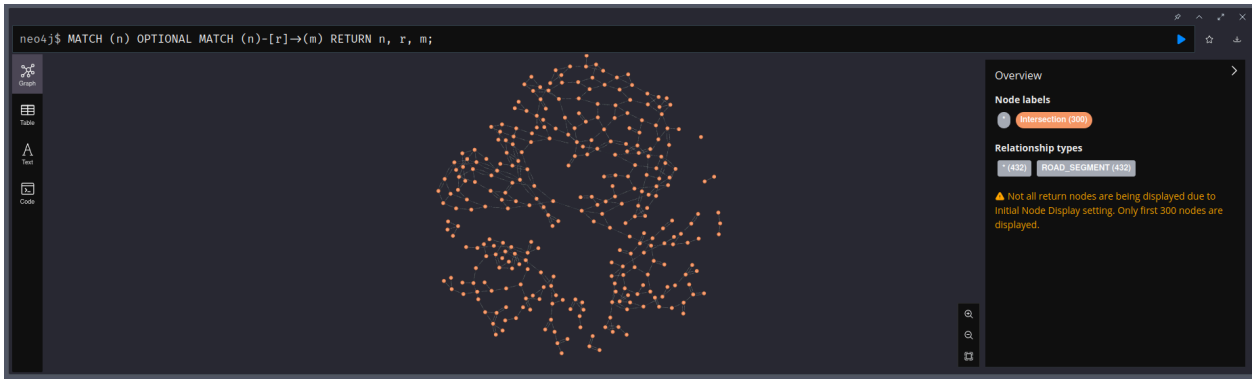


Fig. 23 Sevilla graph from Local Neo4j console

Node details:

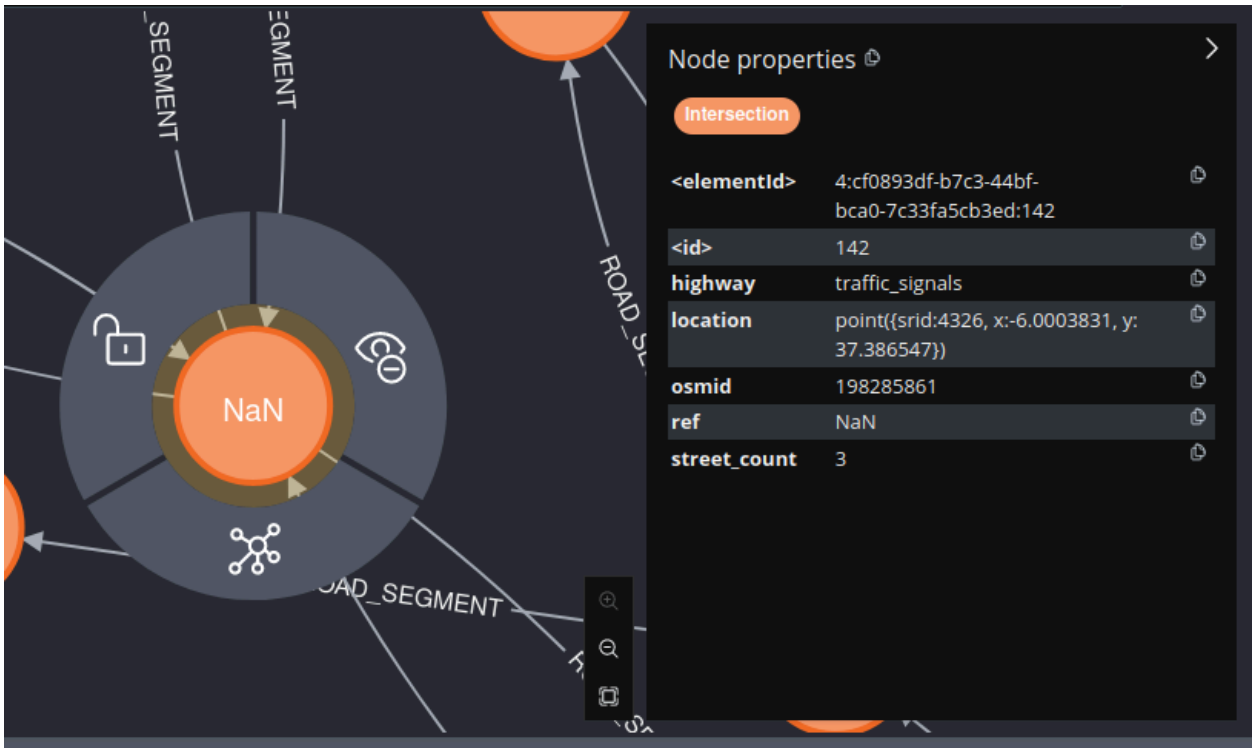


Fig. 24 Node details from Local Neo4j console

Edge details, Example in Figure 26 is Paseo de Cristóbal Colón a big avenue with multiple directions next to the river and “La Torre del Oro”:

Fig. 25 Edge details from Local Neo4j console

Street names:

Fig. 26 Edge property (Street Name)

Calculating a route:

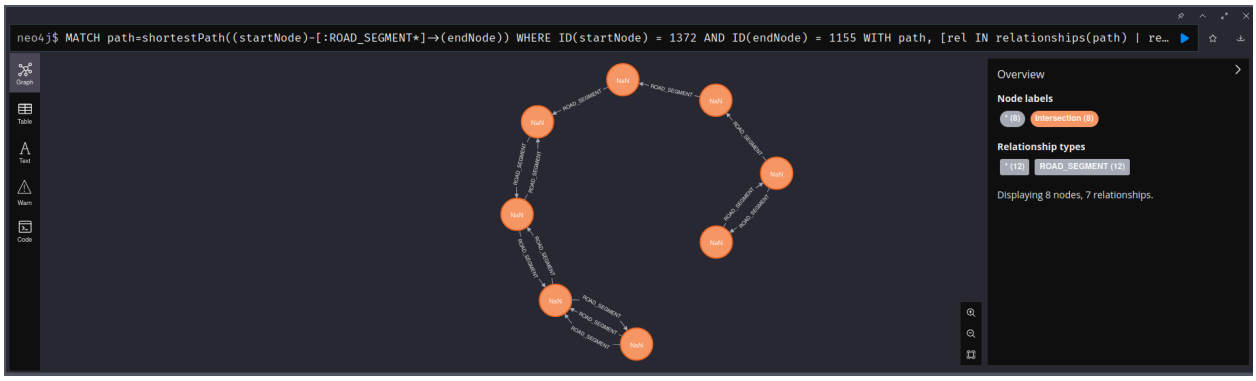


Fig. 27 Calculus of a route from Local Neo4j console

Despite the good news of having this database working locally with no issue it is important to note the problem of visualizing the graph, due to the fact that now when the whole graph is tried to be shown it has a display limit of 300 nodes which is even less than the 1000 nodes that could be seen using Aura. In addition this solution does not have the explore view that Aura possessed meaning that the native visualization options in Neo4j are much more limited in this scenario. New open source solutions must be explored and analyzed.

7. Visualizing the graph

As already seen, the visualization of the graph is a big concern for this project because being able to see the graph is crucial for understanding what is going on and the different operations and analysis that can be performed. Nonetheless it is something hard for this project as the graph that is being worked for is as big as more than 10000 nodes.

The first option [14] Neo4j provides for visualization is the Neo4j browser. That is the basic console that can be used for interacting with the database. For small graphs it is ideal since it is easy to get started with, provides a direct view of the graph and it is great for rapid query development. Despite that only 300 nodes can be visualized at the same time which is pretty limiting for this project. Neo4j browser can be really helpful for computing routes like in section 5 but not good to observe the whole graph.

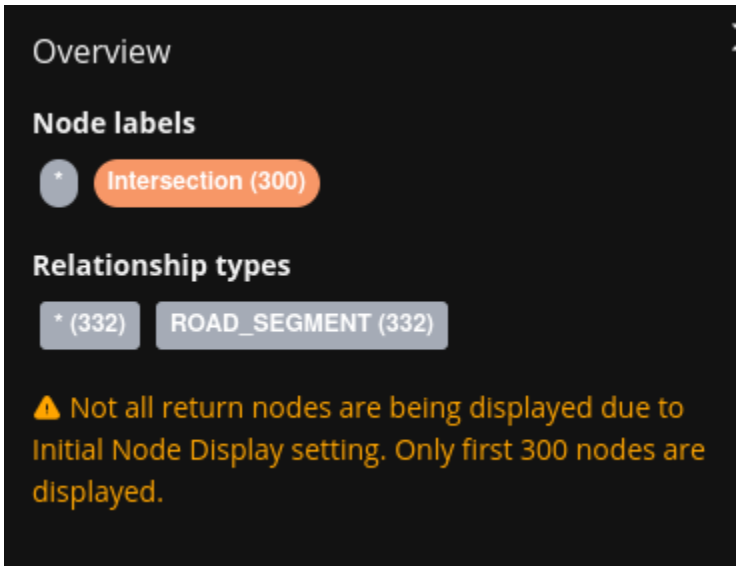


Fig. 28 Visualization properties in local Neo4j

The other option Neo4j offers is Neo4j Bloom [9], a beautiful and expressive data visualization tool to quickly explore and freely interact with Neo4j's graph data platform with no coding required. However as the decision for this project was to try to use the least possible services from Neo4j and more open source solutions this option was not explored

for the development of the project. Nevertheless here it can be seen the difference between the free and the paid version of Neo4j Bloom [11].

Bloom Features	Basic *free*	Enterprise *with paid plan*
Near-Natural language search	✓ Yes	✓ Yes
Exploration	✓ Yes	✓ Yes
Use with Local Database	✓ Yes	✓ Yes
Use with Remote Databases	✗ No	✓ Yes
Scene Saving	✗ No	✓ Yes
Perspective Storage	Local to client	In database
Sharing and Authorization	✗ No	✓ Yes

Fig. 29 Free VS Paid Neo4j comparison

The first tried solution for the visualization of the graph was like before using matplotlib with the difference that in this case the graph from neo4j is being plotted and not the graph from OSM. This is a quick and easy solution to implement from a coding point of view, however the graph takes a long time to be plotted and can be barely seen. As there are so many nodes and relationships matplotlib shows its limitations on plotting it and results in something more similar to a doodle than a graph.

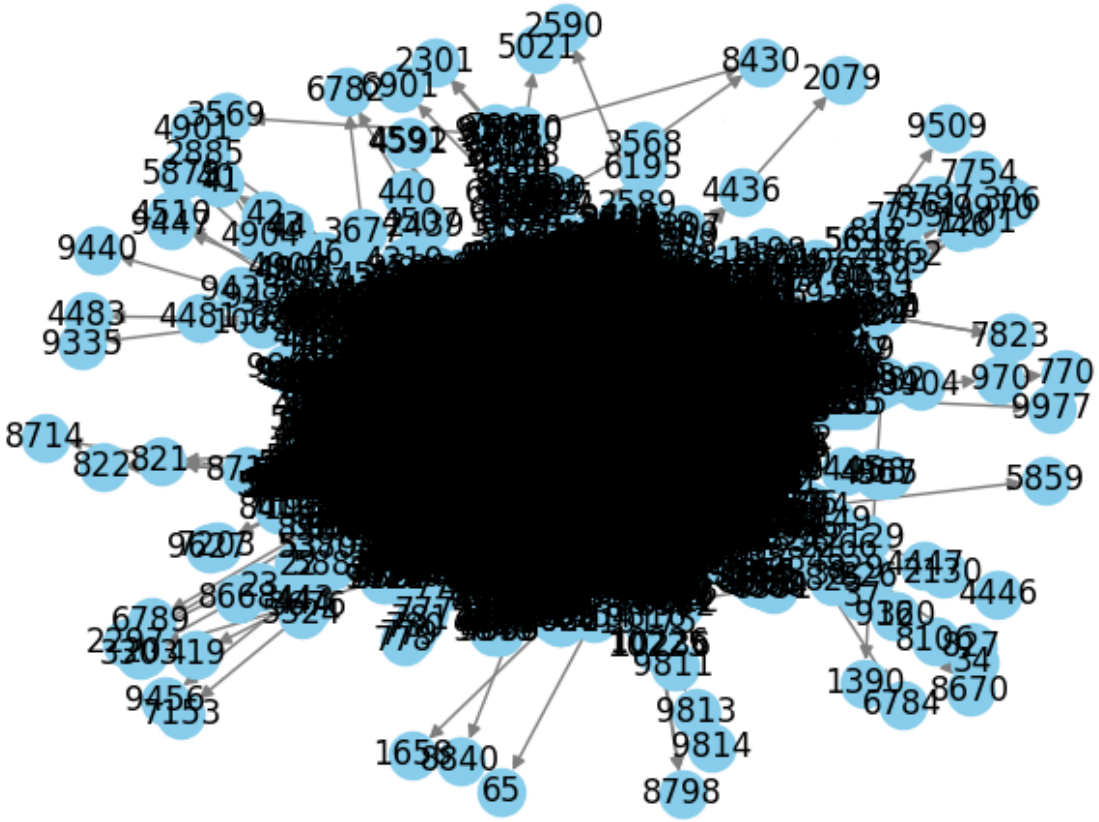


Fig. 30 Neo4j Graph plotted with matplotlib

Given this result more complex solutions were considered. There are several Java Script libraries intended for the visualization of graphs, not many are prepared for data from Neo4j but there are some options. Firstly Cytoscape [8], which is an open source software platform for visualizing molecular interaction networks and biological pathways and integrating these networks with annotations, gene expression profiles and other state data. Although Cytoscape was originally designed for biological research, now it is a general platform for complex network analysis and visualization so in theory it is ideal for representing the graph for this project.

Visualizing data from Neo4j using Cytoscape.js involves a structured approach that seamlessly integrates database querying with graphical representation. The process initiates with setting up the HTML and CSS framework to host the visualization.

```html
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Visualization with Cytoscape.js</title>
    <script src="https://unpkg.com/cytoscape/dist/cytoscape.min.js"></script>
    <script src="https://cdn.jsdelivr.net/npm/neo4j-driver"></script>
    <style>
        #cy {
            width: 100%;
            height: 100vh;
            border: 1px solid lightgray;
        }
    </style>
</head>
```

The core of the operation is orchestrated through JavaScript, beginning with establishing a connection to the Neo4j database using its native driver. Once connected, an asynchronous function, fetchData, is defined to execute a Cypher query, retrieving a predefined limit of nodes and relationships from the graph database.

```javascript
const uri = "bolt://localhost:7687";
const user = "";
const password = "";

const driver = neo4j.driver(uri, neo4j.auth.basic(user, password));
const session = driver.session();

const fetchData = async () => {
    try {
        const result = await session.run(
            "MATCH (n)-[r]->(m) RETURN n, r, m LIMIT 1000"
        );

        const nodes = new Set();
        const edges = [];

        result.records.forEach(record => {
            const n = record.get('n');
            const m = record.get('m');
            const r = record.get('r');

            nodes.add({ id: n.identity.low, label: n.labels[0], ...n.properties });
            nodes.add({ id: m.identity.low, label: m.labels[0], ...m.properties });
            edges.push({ id: r.identity.low, source: n.identity.low, target: m.identity.low, ...r.properties });
        });

        return {
            nodes: Array.from(nodes),
            edges: edges
        };
    } catch (error) {
        console.error('Something went wrong: ', error);
    } finally {
        await session.close();
        driver.close();
    }
};
```

The fetched data undergoes transformation within the fetchData function, where nodes and edges are extracted from the query result and formatted into structures compatible with Cytoscape.js. This involves iterating over the result set, populating nodes and edges sets with their respective properties such as IDs, labels, and other attributes. The aim is to construct a dataset that represents the graph's entities and their connections accurately.

Upon successful data retrieval and transformation, the visualization is constructed. Cytoscape.js then renders the graph based on this data, applying predefined styles to nodes and edges to enhance readability and aesthetics. The visualization style is customizable, allowing for adjustments in content display, color schemes, and layout properties to suit different data characteristics or user preferences.

When executed, the graph will look somehow like the image below. It is important to note that in the html file for the creation a limit of 1000 nodes is set because as others this solution presents performance problems when trying to represent too many nodes.

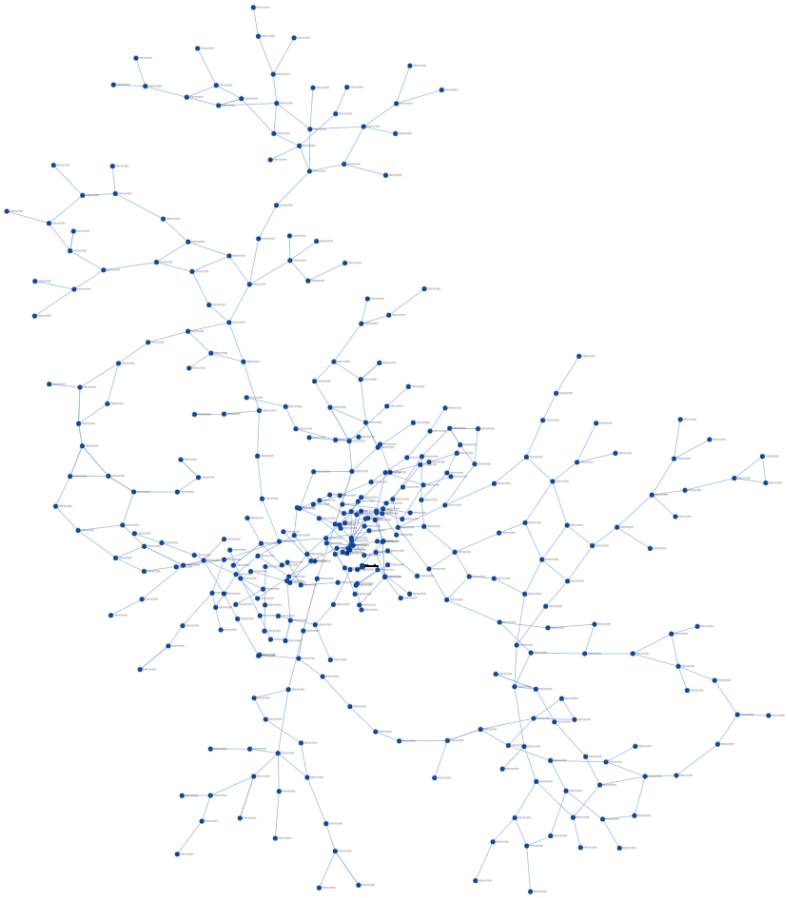


Fig. 31 Neo4j Graph plotted with Cytoscape.js

Another possible solution for the visualization is Neovis.js [22], a JavaScript library to help developers build graph visualizations from Neo4j data. Providing a bridge between Cypher and a customizable graph visualization in the browser. The approach to visualize the graph involves embedding necessary scripts for Neovis.js and configuring a visual container (#viz) styled to fill the viewport.

```
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Visualization with Neovis.js</title>
    <script src="https://cdn.jsdelivr.net/npm/vis-network@9.0.2/dist/vis-network.min.js"></script>
    <script src="https://cdn.jsdelivr.net/npm/neovis.js@1.5.0/dist/neovis.js"></script>
    <style>
        #viz {
            width: 100%;
            height: 100vh;
            border: 1px solid lightgray;
        }
    </style>
</head>
```

The core of this visualization technique is the configuration object specified within the script. This object contains settings for connecting to the Neo4j database, including the server URL and credentials. It also defines how nodes and relationships from the graph are visualized, specifying properties like captions, sizes, and fonts for nodes, as well as thickness and captions for relationships.

```
script type="text/javascript">
   var config = {
       container_id: "viz",
       server_url: "bolt://localhost:7687",
       server_user: "",
       server_password: "",
       labels: {
           "Intersection": {
               caption: "osmid",
               size: "length",
               font: {
                   size: 14
               }
           }
       },
       relationships: {
           "ROAD_SEGMENT": {
               thickness: "length",
               caption: false
           }
       },
       initial_cypher: "MATCH (n)-[r]->(m) RETURN n, r, m LIMIT 5000"
   };

   var viz = new NeoVis.default(config);
   viz.render();
```

A Cypher query, detailed in the configuration, directs what portion of the graph to visualize. Upon initializing a Neovis.js instance with this configuration, the viz.render() method is called, which automatically fetches the graph data, processes it, and renders an interactive graph in the specified container.

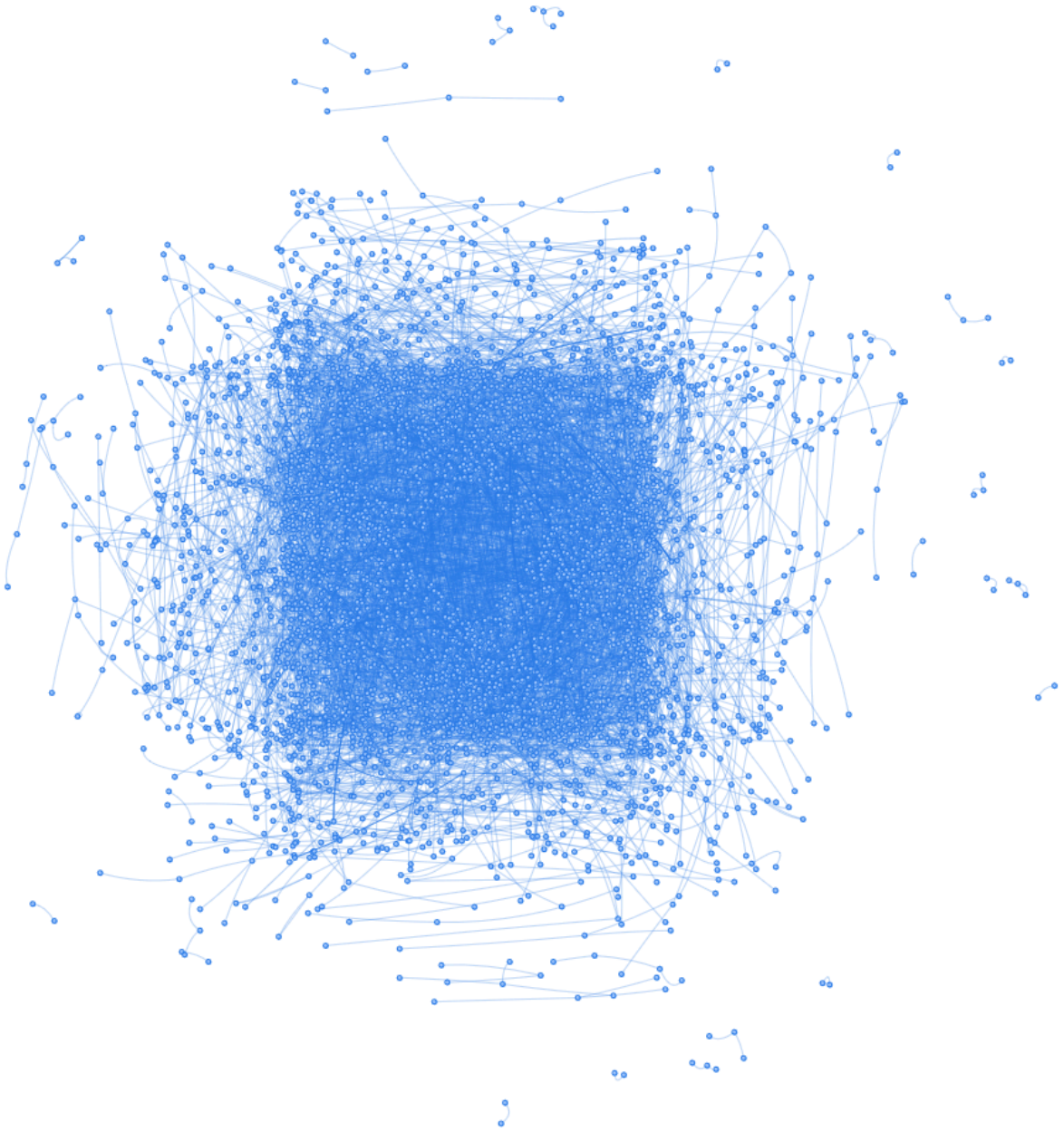


Fig. 32 Neo4j Graph plotted with Neovis.js

Nonetheless the created graph has the same performance problems that Cytoscape presents, although a bit better since for example in the image there are 5000 nodes. Another good improvement from Cytoscape is the fact that it detects and is able to display the properties from the nodes and edges, which is really good for understanding the graph.

Fig. 33 Edge and Node properties in Cytoscape.js

In conclusion, the exploration of graph visualization within this project underscores the importance of effective visual representation for the analysis and understanding of urban and traffic networks. While tools like Cytoscape.js and Neovis.js extend our capabilities beyond the basic offerings of the Neo4j Browser, the search for scalable, high-performance visualization solutions continues. Future work in this area will be vital for advancing our ability to comprehend and optimize urban infrastructure and traffic systems, leveraging the full potential of graph databases like Neo4j to transform raw data into actionable insights. However for the scope of this project it is still possible to use any of the mentioned tools for visualizing at least partly the graph and then Neo4j browser or matplotlib can perfectly be used to represent the different routes calculated.

8. Interacting with the database without Cypher using Py2neo

The final step for this project is being able to interact with the database like in section 5 but without using Cypher. For that Py2neo [23] will be used, which is a client library and toolkit for working with Neo4j from within Python applications. The library supports both Bolt and HTTP and provides a high level API.

```
from py2neo import Graph, RelationshipMatcher
```

Inside this library Graph and RelationshipMatcher will be used. The Graph class represents a Neo4j graph database. It is used to establish a connection to a Neo4j database instance, allowing the user to execute queries, create or retrieve nodes and relationships, and perform various operations on the graph. When a Graph object is instanced, connection details are normality given (such as the database URL, username, and password) to connect to the Neo4j database. This object then serves as the primary interface for interacting with the database through Python. Then the RelationshipMatcher class is designed to help the user find and work with relationships in the graph database. It allows users to construct queries to search for relationships based on specific criteria, such as the types of relationships, properties of relationships, or the nodes they connect. This makes it easier to retrieve and manipulate data within a Neo4j graph, especially when dealing with complex relationship patterns or needing to filter relationships for analysis or modification.

In order to showcase how the interaction with Neo4j databases using python with Py2neo works, a script was written for receiving the name of a street and returning all the nodes intersecting with it and its coordinates. When 2 nodes ids are given, it calculates the best route from one to another using an own implementation of Dijkstra shortest path [18], A* [17] and Breadth-first search (BFS) [19] algorithms.

FIrst the connection with the database is set by using the Graph class, and the function to find the nodes from a street is called with the graph and the street.

```
# Connect to Neo4j
graph = Graph("bolt://localhost:7687", auth=None)

# Call the function with the name of the street
street_name = "Avenida de la Reina Mercedes"
street_nodes = find_street_nodes(graph, street_name)
```

This function uses the RelationshipMatcher class to search within edges of the graph. First it takes all edges of type ROAD_SEGMENT and with the name property equal to the given street name. Then it takes all the nodes interacting with them and returns their id and location coordinates.

```
# Function to search for nodes connected to a specific street
def find_street_nodes(graph, street_name):
    # Create a RelationshipMatcher
    rel_matcher = RelationshipMatcher(graph)

    # Search for relationships connected to the specified street
    relationships = rel_matcher.match(r_type="ROAD_SEGMENT", name=street_name)

    # Set to store unique node identities
    unique_node_ids = set()

    # Process relationships and add unique nodes to the set
    for rel in relationships:
        unique_node_ids.add(rel.start_node.identity)

    # Return the details of unique nodes
    nodes_info = []
    for node_id in unique_node_ids:
        node = graph.nodes.get(node_id)
        nodes_info.append((node.identity, node.get('location', None)))

    return nodes_info
```

```
Nodes connected to 'Avenida de la Reina Mercedes':
2227 POINT(-5.9863768 37.3566984)
1076 POINT(-5.9860847 37.3635235)
2260 POINT(-5.9863248 37.3578945)
2230 POINT(-5.9861974 37.3607887)
4566 POINT(-5.9863973 37.356142)
2393 POINT(-5.9863599 37.3571474)
2234 POINT(-5.9861395 37.362196)
2235 POINT(-5.9862607 37.3592587)
```

For computing the different routes. The 3 algorithms follow the same structure for how they are called and what they return. The 3 of them receive the graph, the id of the starting node and the id of the finishing node and return the cost of the route in meters and the route as a list of nodes.

```
dijkstra(graph, start_node_id, end_node_id)
```

```
astar(graph, start_node_id, end_node_id)
```

```
bfs(graph, start_node_id, end_node_id)
```

In dijkstra a list with all nodes and edges (or relationships) is calculated using the given graph and then an adjacency list created for computing the algorithm. This adjacency list is a map where the keys are the identifier of each node and the values are all the identifiers and length in meters of the adjacent nodes to the given one. With that data the dijkstra algorithm can be calculated and it is out of the scope of this project to explain how it works as well as for the other algorithms.

```python
def dijkstra(graph, start_id, end_id):
    # Fetch all nodes and relationships to build the graph structure
    nodes = list(graph.nodes.match("Intersection"))
    rels = list(graph.relationships.match(r_type="ROAD_SEGMENT"))

    # Create adjacency list representation of the graph
    adjacency_list = {node.identity: [] for node in nodes}
    for rel in rels:
        adjacency_list[rel.start_node.identity].append((rel.end_node.identity, rel['length']))
```

For A* the start and end nodes are gotten from their id and then their latitude and longitude are retrieved. Then a priority queue is set for the functioning of the implementation.

```
def astar(graph, start_id, end_id):
    start_node = graph.nodes.get(start_id)
    end_node = graph.nodes.get(end_id)

    # Extract latitude and longitude from the start and end nodes
    start_lat, start_lon = start_node['location'].latitude, start_node['location'].longitude
    end_lat, end_lon = end_node['location'].latitude, end_node['location'].longitude

    # Priority queue for A* algorithm
    open_set = [(0 + haversine(start_lat, start_lon, end_lat, end_lon), 0, start_id, [])]  # (f_score, g_score, node_id, path)

    # Visited and cost dictionaries
    visited = set()
    g_score = {start_id: 0}
```

It is important to note how Py2neo helps in the process of the algorithm with things like graph.nodes.get(node), relationship.end_node or node.identity() which makes the way of interacting with the graph much more natural and straightforward.

```
neighbors = graph.relationships.match(nodes=[graph.nodes.get(current)], r_type="ROAD_SEGMENT")
for rel in neighbors:
    neighbor = rel.end_node
    temp_g_score = current_g + rel['length']

    if neighbor.identity in visited and temp_g_score >= g_score.get(neighbor.identity, float('inf')):
        continue
```

FInally in BFS the algorithm is applied normally but Py2neo helps to get the properties and nodes directly more easily.

```
visited.add(current_node)
neighbors = graph.relationships.match(nodes=[graph.nodes.get(current_node)], r_type="ROAD_SEGMENT")

for rel in neighbors:
    neighbor = rel.end_node
    queue.append((neighbor.identity, distance + rel['length'], path + [current_node]))
```

Once all of them are executed they return a response as follows, with the distance in meters and the nodes that make up the route. Normally Dijkstra and A* will always give the same solution but just with a difference in execution time depending on the case. However BFS will give a different result this is because BFS is a purely academical algorithm and it does not have the precision of the other two. The only reason why BFS is included in this project is because when calculating a route in Neo4j using Cypher this uses the

shortestpath() function which is built using BFS, so this way a direct comparison can be made from the route that this program computes and the one computed by Cypher.

```
Dijkstra shortest path from node 4566 to node 766 :
Shortest path cost: 6221.508000000001 meters
Shortest path: [4566, 2227, 2393, 2260, 2235, 2230, 2234, 1076, 2336, 2442, 9149, 6867, 2335, 8915, 8916, 4633, 265, 4629, 4630,
1541, 264, 9010, 263, 262, 261, 260, 259, 258, 257, 9467, 9458, 4710, 166, 154, 146, 144, 142, 8212, 2, 6466, 7811, 9608, 244, 24
3, 6454, 6860, 6858, 6859, 6861, 6855, 6856, 6857, 6853, 6854, 241, 233, 8555, 214, 229, 230, 231, 6719, 766]


A* shortest path from node 4566 to node 766 :
Shortest path cost: 6221.508000000001 meters
Shortest path: [4566, 2227, 2393, 2260, 2235, 2230, 2234, 1076, 2336, 2442, 9149, 6867, 2335, 8915, 8916, 4633, 265, 4629, 4630,
1541, 264, 9010, 263, 262, 261, 260, 259, 258, 257, 9467, 9458, 4710, 166, 154, 146, 144, 142, 8212, 2, 6466, 7811, 9608, 244, 24
3, 6454, 6860, 6858, 6859, 6861, 6855, 6856, 6857, 6853, 6854, 241, 233, 8555, 214, 229, 230, 231, 6719, 766]


BFS shortest path from node 4566 to node 766 :
Shortest path length: 9061.377999999997
Shortest path: [4566, 2227, 2393, 2260, 2235, 2230, 2506, 2336, 10040, 10039, 2337, 8829, 8831, 4626, 4625, 7878, 7739, 7741, 184
7, 1789, 1792, 741, 742, 743, 744, 731, 735, 754, 4426, 6883, 4768, 1175, 1176, 3418, 3415, 10137, 4650, 4653, 4658, 1019, 1825,
4701, 4703, 1130, 9114, 1020, 762, 763, 767, 766]
```

Apart from that result another visualization using matplotlib was created where the two routes would be superposed. This visualization consists of the route between the starting and finishing nodes with all the nodes in between then respecting their coordinates of latitude and longitude. As it can be seen in figure 34 the A* and the Dijkstra paths are exactly the same and the BFS one is different.

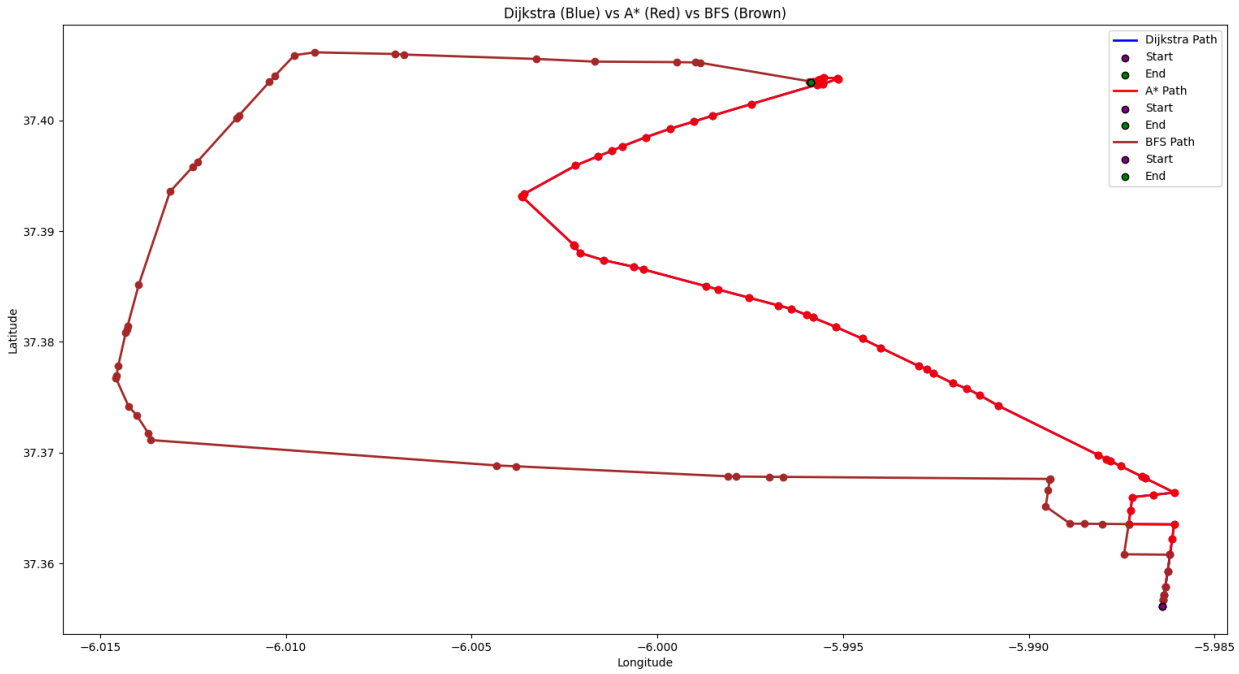


Fig. 34 Dijskrta, A* and BFS paths represented with Matplotlib

If the same source and destination is calculated with Cypher in the Neo4j console it can be observed that it is exactly the same route as the one calculated with BFS. This is very good for comparing the work done in Python with what the Neo4j console returns but BFS is not an ideal algorithm for real life situations and Dijsktra and A* are proven to be much better algorithms for the computation of paths like this one. The only purpose of BFS here is to make the comparison with Neo4j. Figure 35 represents the same route as in BFS in figure 34.

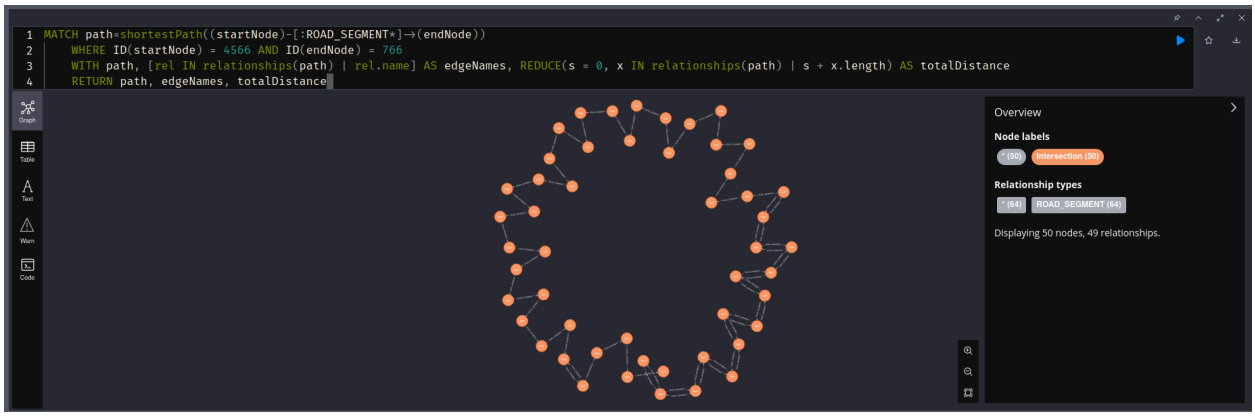


Fig. 35 Same BFS Route as in fig 34 represented in Neo4j

9. Final script

As a final step all the processes related to interacting with the database using Python were gathered in an interactive script using tkinter so it would be easy to use and understand. When executing this script a window opens with the possibility of searching the nodes that interact with one street and calculating the path between two nodes using Dijkstra, A* or BFS.

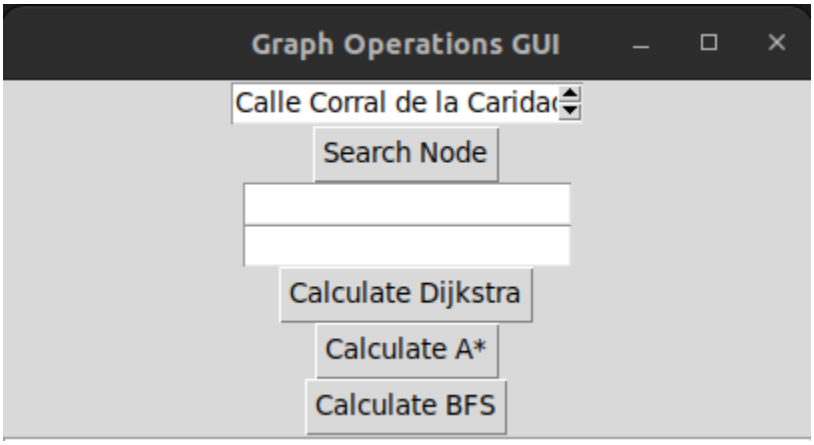


Fig. 36 Tkinter UI overview

The street names are in a Spinbox so they can be selected by pressing the two rows at the right of the box, but there is also the possibility of manually typing the name of the street and then searching it. Once a street name is selected and “Search Node” is clicked the different nodes that are connected to it will appear with their id and latitude and longitude points.

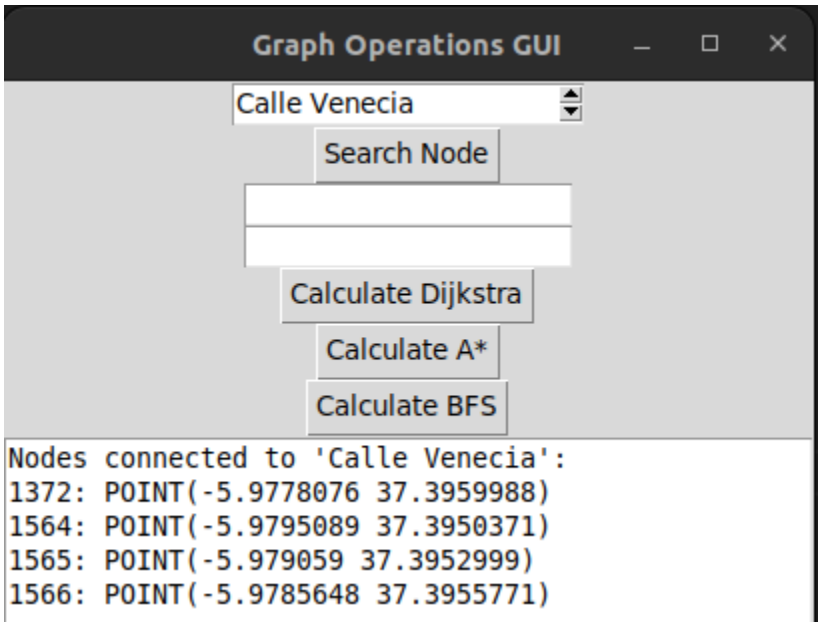


Fig. 37 Nodes intersecting with street in UI

These ids are the ones that will be used to perform the search, so it is as easy as just typing two nodes id and clicking on any of the three available options.

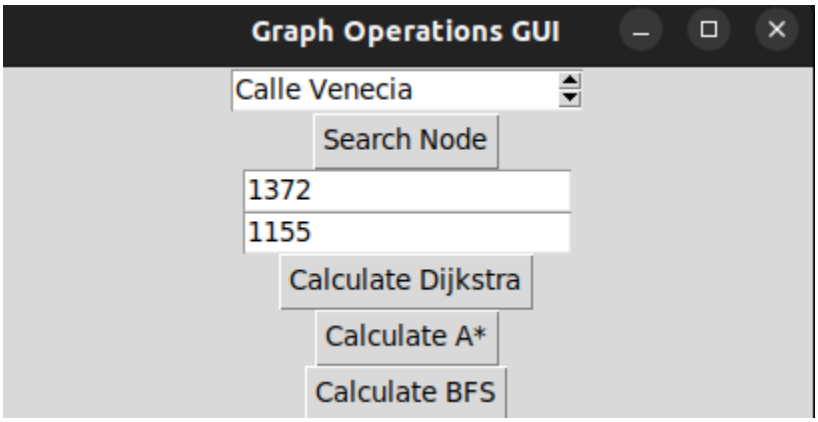


Fig. 38 Selecting nodes in UI

A visualization of the path will be shown created with matplotlib, this visualization will be more detailed than the one created in the last section, since this one will also contain information about the name of the streets that the route goes through, the distance between nodes and the total distance from source to destination.

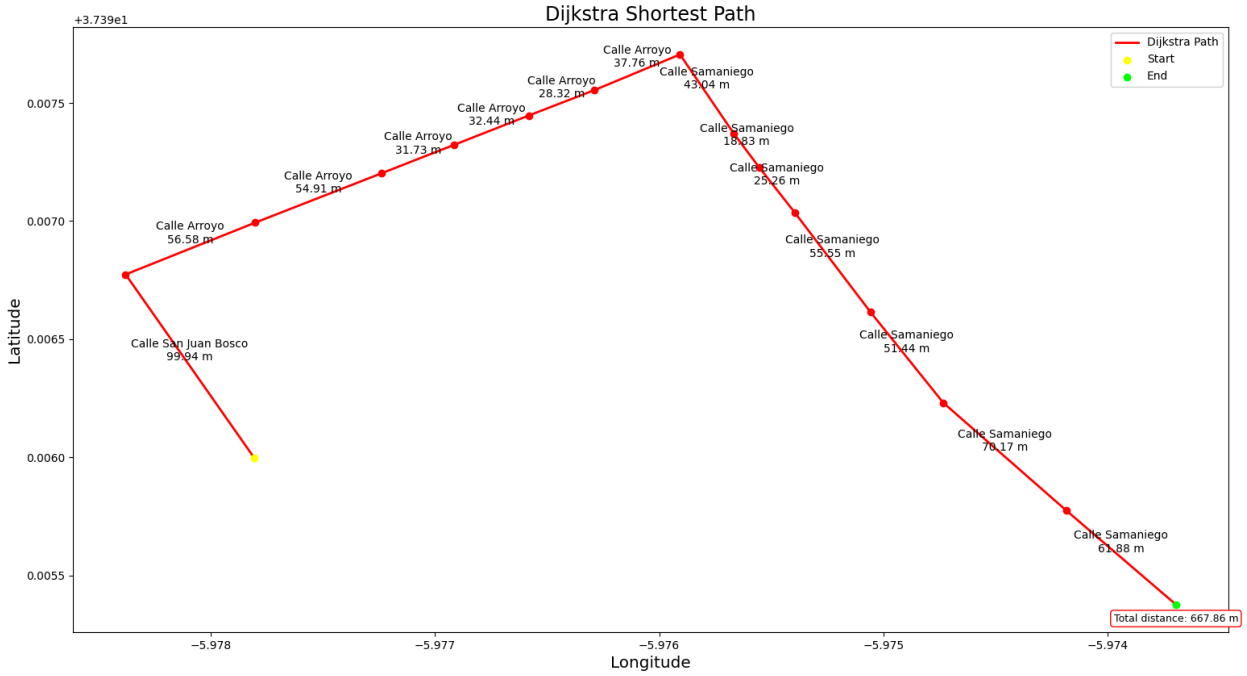


Fig. 39 DijKstra path generated from UI

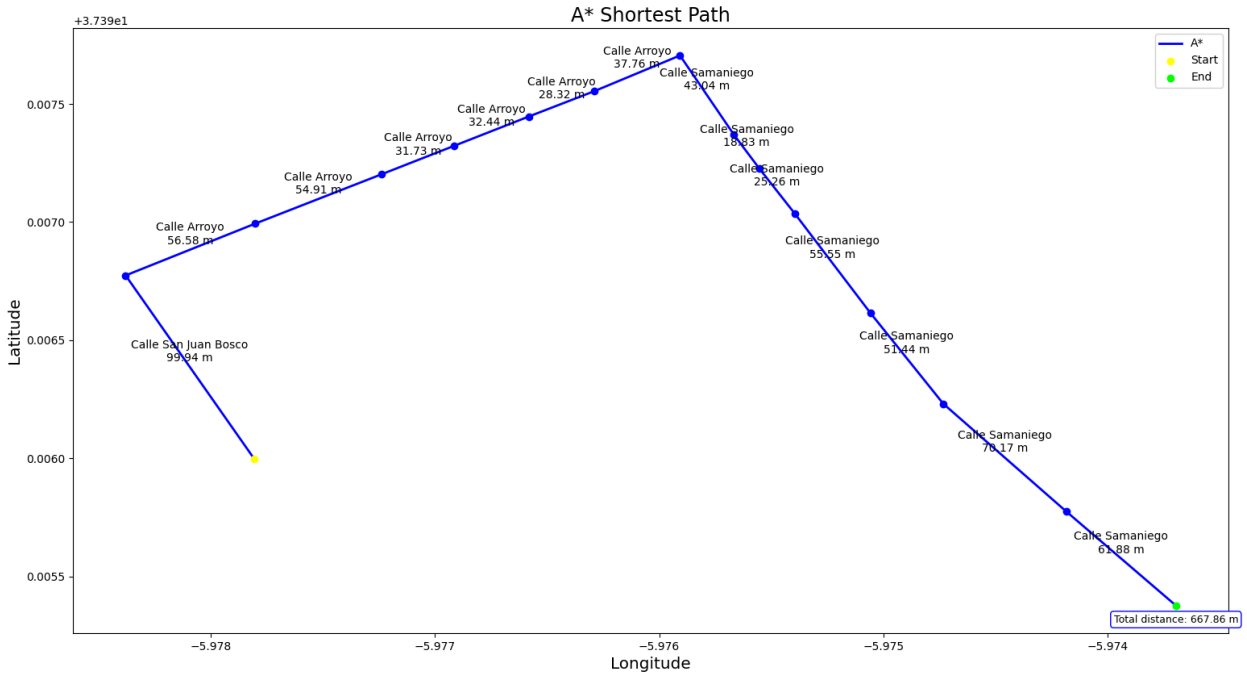


Fig. 40 A* path generated from UI

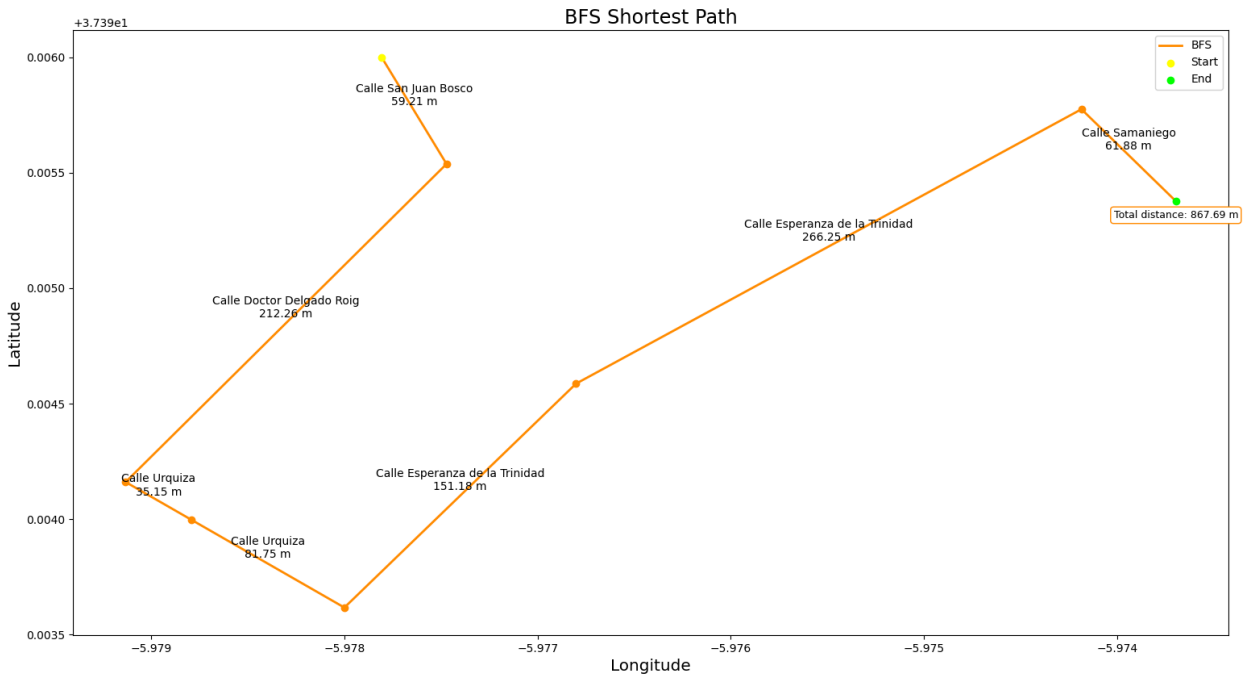


Fig. 41 BFS path generated from UI

There is also the chance to comparte the paths. If the matplotlib generated window is not closed after computing a path and another one is selected it will be generated in the same window. In figure 42 there is a representation of the 3 paths altogether (A* and Dijkstra are the same but take different response times).

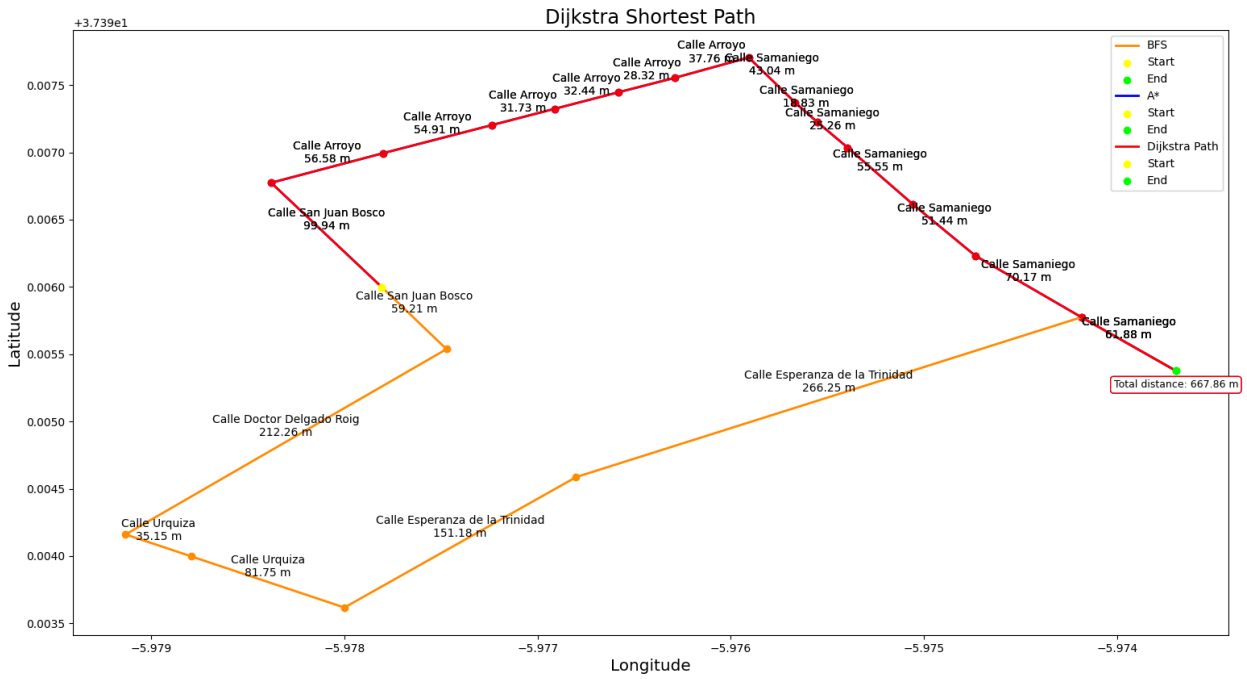
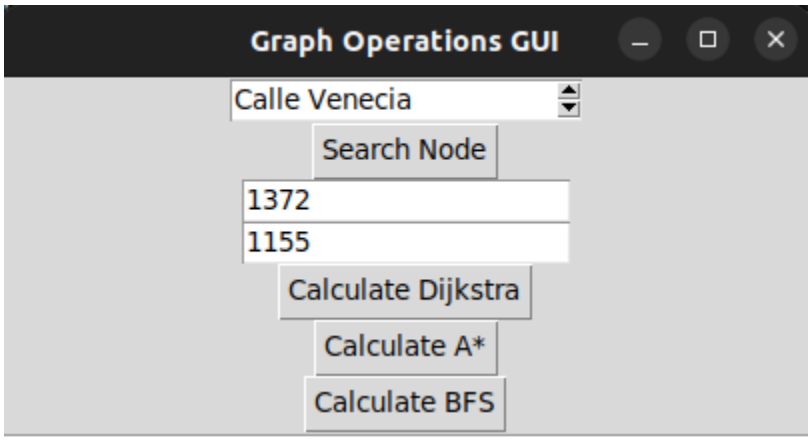
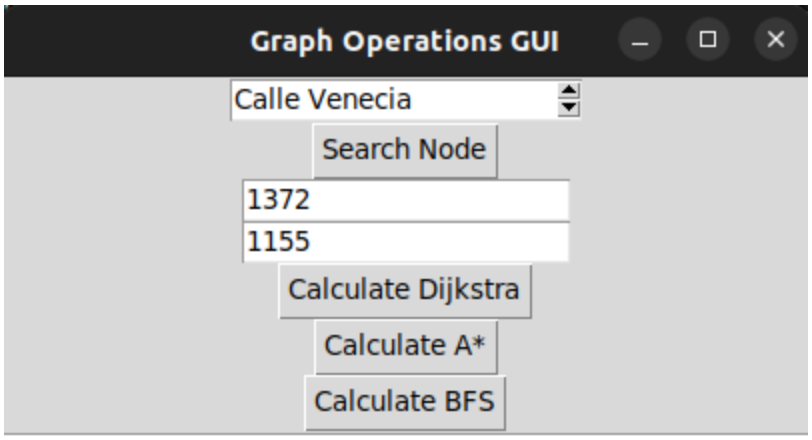


Fig. 42 Dijsktra VS BFS path generated from UI

Besides if the information in the image is not too clear, the original tkinter window will always contain the information of the nodes of the route and the distance after calculating it.




```
Dijkstra shortest path from node 1372 to node 1155
:
Shortest path cost: 667.858 meters
Shortest path: [1372, 1161, 1211, 1214, 1216, 1217
, 1222, 1153, 6840, 1253, 1374, 1249, 6837, 1154,
1155]
```




```
A* shortest path from node 1372 to node 1155:
Shortest path cost: 667.858 meters
Shortest path: [1372, 1161, 1211, 1214, 1216, 1217
, 1222, 1153, 6840, 1253, 1374, 1249, 6837, 1154,
1155]
```

Fig. 43 Dijkstra & A*Path results in UI

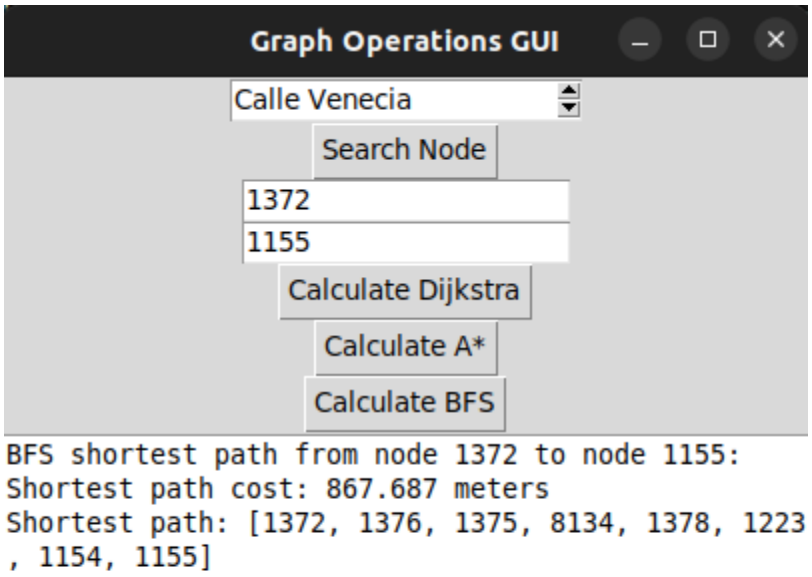


Fig. 44 BFS path result in UI

In conclusion, this demonstrates the successful implementation of an interactive script utilizing tkinter, facilitating direct interaction with the Neo4j database without relying on Cypher queries. By leveraging Py2neo. The script provides an intuitive interface for users to search nodes connected to a specific street, calculate routes using advanced algorithms such as Dijkstra, A*, and BFS, and visually compare these routes. This approach not only simplifies the interaction with the graph database but also enriches the project's capability to analyze and visualize traffic patterns in a more user-friendly manner. The development of this script signifies a pivotal step towards creating accessible and efficient tools for traffic management applications, showcasing the project's innovation in applying software engineering principles to solve real-world problems in urban planning and traffic navigation.

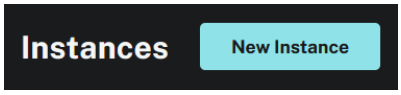
Deployment

This section outlines the processes and methodologies employed for deploying the project using two distinct environments: Neo4j Aura in a cloud-based setting and a local Linux system. Deployment refers to the configuration and availability of the project's database and related applications, ensuring they are operational for development, testing, and potentially for real-world application. By detailing the deployment steps for both environments, this guide aims to provide a comprehensive understanding of how the project's technological infrastructure can be set up and utilized effectively. Whether through the managed service of Neo4j Aura for ease of use and scalability or by leveraging the control and customization offered by a local Linux deployment, each approach caters to different project needs and stages.

1. Using Neo4j Aura

All the code related to the deployment using Neo4j Aura can be found in the GitHub repository MarioArocaPaez /OSM-to-Neo4j which can be executed in any machine running Python. The requirements for this deployment are the code in the given repository and having an instance for a Database in Neo4j Aura.

In order to use Neo4j Aura, the user can go to their official webpage and click on Start Free. Once there an account must be created, the log in can also be done using Google or Organization SSO. Once done, a new instance must be created. Here it is when the difference between a paid version and a free version is highlighted. The main restrictions for projects like this one are the memory available which for free users is 1GB and for paid users is 64 GB and the Graph Size, a free user can only have up to 200K nodes and 400K relationships. That is one of the reasons why this project is only intended for medium cities because a big city like Tokyo or New York or even a whole region like the



repository mentioned in the state of art would not be able to be loaded in a database like this one.

The user will select “Create Free instance” and a password will be generated. It is important to store this password safely. When the instance is created and running everything will be done in Neo4j Aura.

In the Github project there are different dependencies required to run all scripts, they are all listed in the requirements.txt file and can be all installed at the same time by running:

```
pip install -r requirements.txt
```

Once everything is installed it is fundamental to create a .env file where the Neo4j credentials will be stored because by taking a look at the script it can be seen that in order to do the connection with Neo4j Aura the credentials are required and the scripts are prepared to read them from an .env file. In this environment variable file it is necessary to have the uri of the instance which is the uri that we use to connect to the instance from the Neo4j Aura webpage, the Neo4j user that unless changed it is just neo4j by default and the password generated before.

```
NEO4J_URI = os.getenv("NEO4J_URI")
NEO4J_USER = os.getenv("NEO4J_USER")
NEO4J_PASSWORD = os.getenv("NEO4J_PASSWORD")
```

This project contains multiple scripts. OSMSmallPlot.py creates the matplotlib plot of the graph mentioned in section 2 of Methodology. Then neo4jImport is the script intended to create the Neo4j database from the graph. If in this script or the previous one the city to be transformed wants to be changed it is possible just by simply replacing the name of Sevilla with the one from the required city.

```
# Search OpenStreetMap and create an OSMNx graph
G = ox.graph_from_place("Sevilla, Andalucía, España", network_type="drive")
```

The following script, neo4jOperations.py is the one intended for calculating the routes given a Cypher query and depicting it using matplolib, mentioned in section 5 of Methodology. And finally cyperQueries.py is just a list of Cypher queries with a description of what they do that can be used in the Neo4j Aura console in order to interact directly with the database.

2. Locally on Linux

All the code related to the deployment using Neo4j Locally on Linux can be found in the GitHub repository MarioArocaPaez/neo4jLinuxConnection which can be executed in any machine running Python and Linux. The requirements for this deployment are the code in the given repository and having a Neo4j database running on your local LInux machine. It is important to take into account that this project has been built and the deployment is explained for Ubuntu 22.04.3 LTS.

The official Ubuntu package repositories do not contain a copy of the Neo4j database engine [13]. To install the upstream supported package from Neo4j the user will need to add the GPG key from Neo4j to ensure package downloads are valid. Then add a new package source pointing to the Neo4j software repository, and finally install the package. Firstly the following command is needed:

wget -O - https://debian.neo4j.com/neotechnology.gpg.key | sudo apt-key add -

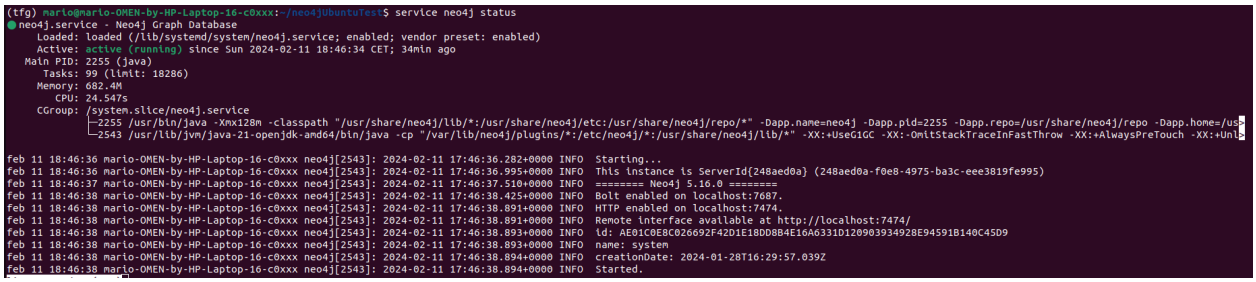
Afterwards the Neo4j repository must be added to the system's APT sources:

echo 'deb https://debian.neo4j.com stable latest' | sudo tee /etc/apt/sources.list.d/neo4j.list

Once this is done it is only necessary to run *sudo apt-get update* and then:

sudo apt install neo4j

When finished neo4j will be installed in the LInux machine. To start using it is crucial to understand that it works as a service so in order to use it the user will have to run *service neo4j start* to stop it *service neo4j stop* and to check the status *service neo4j status*.



After running it, it can be seen working at http://localhost:7474/browser/ however it will ask for user and password. This can be changed so no authentication is needed by going to the neo4j configuration file, neo4j.conf which in most cases will be in the direction /etc/neo4j/neo4j.conf. And including the following line:

dbms.security.auth_enabled=false

The line might be already included but commented or set to false it is important that it is not repeated. Once the line is added, neo4j service must be restarted and the user will be able to connect to the database without any type of authentication required, just by clicking on connect.

With the database running the repository contains multiple scripts. First it is important to install all dependencies by running:

```
pip install -r requirements.txt
```

Next, within all the scripts connection.py is a simple script just intended to fill the database with some test nodes, osmToNeo4j.py is the important one that runs all the process of extracting the data from Sevilla, converting it into a graph and inserting it into the Neo4j database but now on a local database. As in the last section, if the city wants to be changed it can be done by changing this line to the desired city:

```
# Search OpenStreetMap and create an OSMNx graph
G = ox.graph_from_place("Sevilla, Andalucía, España", network_type="drive")
```

visualization.py generates a visualization of the whole script using matplotlib which is not really precise and does not contain any visual important data, but is part of the process for searching for the right visualization tool.

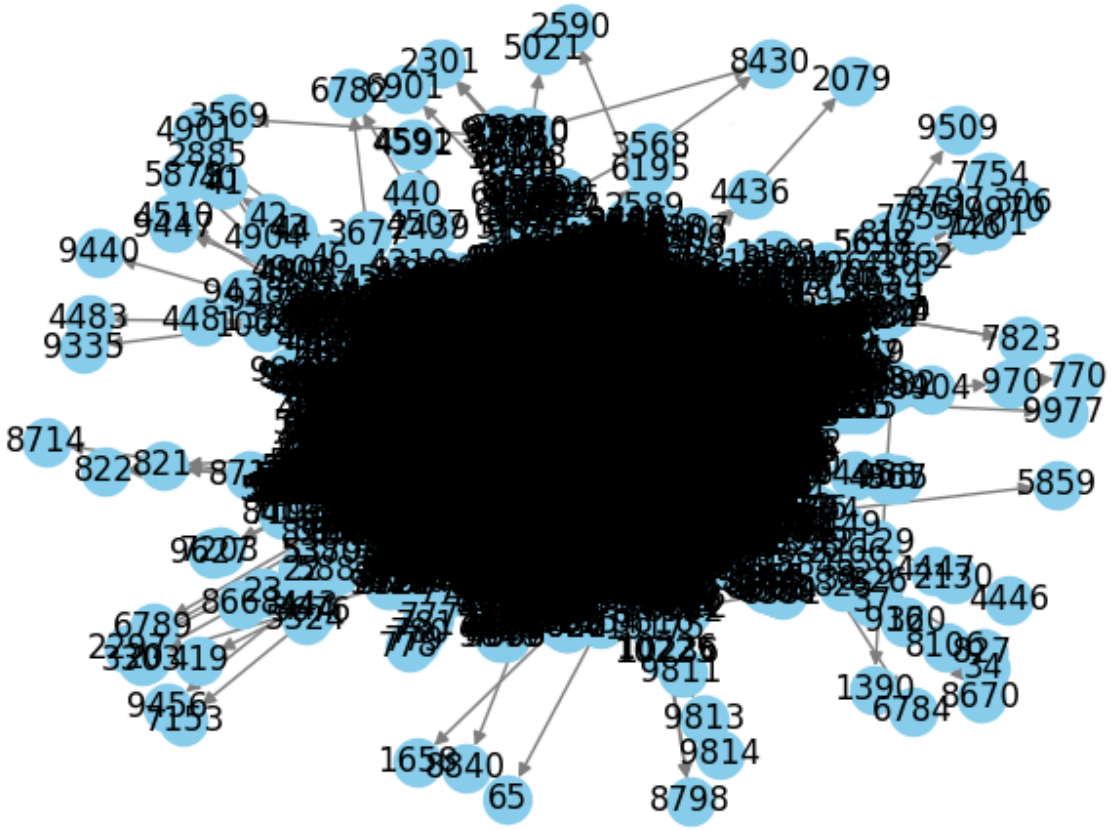


Fig. 44 Neo4j graph plotted with Matplotlib

visualizationCytoscape.html provides a visualization of the graph using cytoscape. By default it shows the graph with only 1000 nodes. The user is free to change that limit knowing that it can result in performance issues with the visualization.

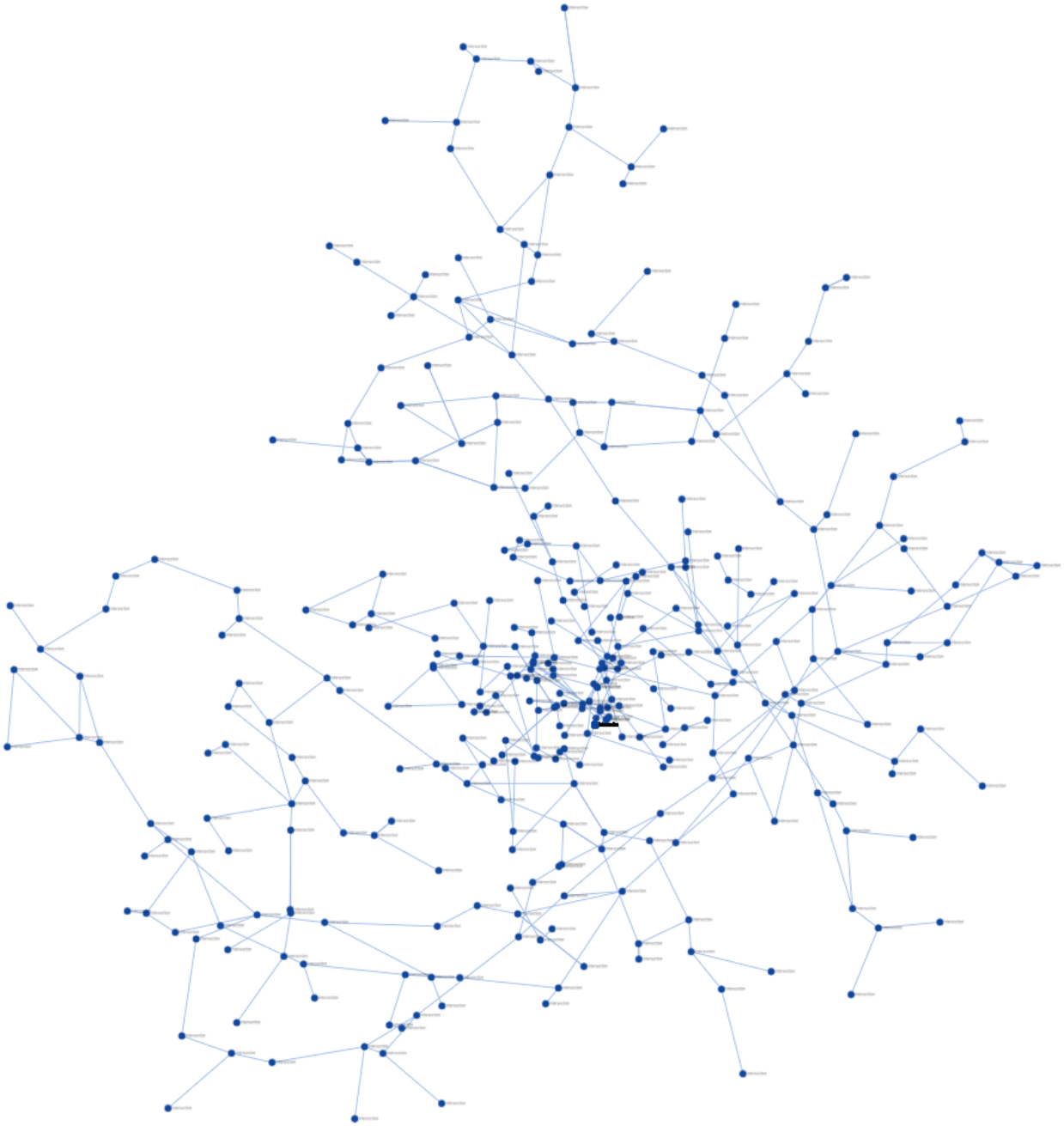


Fig. 45 Neo4j graph plotted with Cytoscape.js

And visualizationNeovis.html also provides a visualization of the graph but using Neovis, it has the same issues with the number of nodes but this one performs slightly better and by default in the file it has a limit of 5000 nodes.

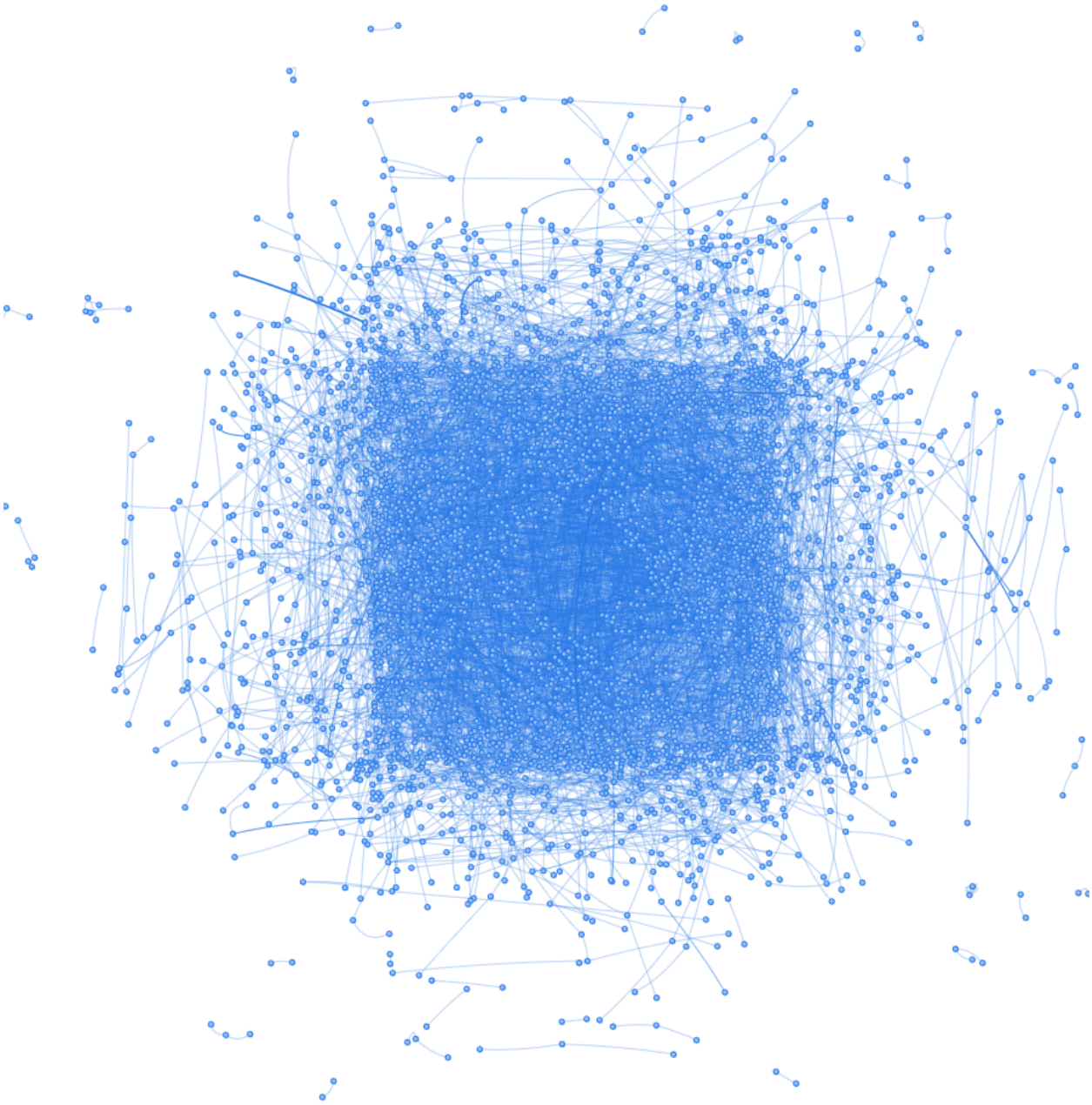


Fig. 46 Neo4j graph plotted with Neovis.js

Next, cyperQueries.py is just a simple script with some Cypher queries that can be performed in the Neo4j console, operations.py is the script described in section 8 of the Methodology to perform the operations. By default it shows the nodes intersecting with Avenida de la Reina Mercedes and Calle Torneo and then computes the route between two

nodes from those streets with the different algorithms. This can be easily changed in the following sections:

Nodes from the streets:

```
# Call the function with the name of the street
street_name = "Avenida de la Reina Mercedes"
street_nodes = find_street_nodes(graph, street_name)

# Print the details of the nodes returned from the function
print(f"Nodes connected to '{street_name}':")
for node_id, location in street_nodes:
    print(node_id, location)

# Call the function with the name of the street
street_name = "Calle Torneo"
street_nodes = find_street_nodes(graph, street_name)
print("\n")

# Print the details of the nodes returned from the function
print(f"Nodes connected to '{street_name}':")
for node_id, location in street_nodes:
    print(node_id, location)
print("\n")
```

Dijkstra path:

```
# Calculate Dijkstra shortest path
start_node_id = 4566
end_node_id = 766
cost, dijkstra_path = dijkstra(graph, start_node_id, end_node_id)

print("Dijkstra shortest path from node", start_node_id, "to node", end_node_id, ":")
print("Shortest path cost:", cost, "meters")
print("Shortest path:", dijkstra_path)
print("\n")
```

A* path:

```
# Calculate A* shortest path
start_node_id = 4566
end_node_id = 766
cost, astar_path = astar(graph, start_node_id, end_node_id)

print("A* shortest path from node", start_node_id, "to node", end_node_id, ":")
print("Shortest path cost:", cost, "meters")
print("Shortest path:", astar_path)
print("\n")
```

BFS path:

```
# Calculate BFS shortest path
start_node_id = 4566
end_node_id = 766
cost, bfs_path = bfs(graph, start_node_id, end_node_id)

print("BFS shortest path from node", start_node_id, "to node", end_node_id, ":")
print("Shortest path length:", cost)
print("Shortest path:", bfs_path)
print("\n")
```

And finally there is tkinterOperations.py which is the script described in section 9 of Methodology, this script needs no customization since it is completely interactive, it is only necessary to execute it and start looking for nodes and calculating routes using the tkinter interface.

Time and Budget for the project

The time spent on this project has been tracked using the Forest App [7] resulting in 118 hours of individual work, including research, development and documentation and 4 hours of meetings with the tutor of this project, resulting in 122 hours.

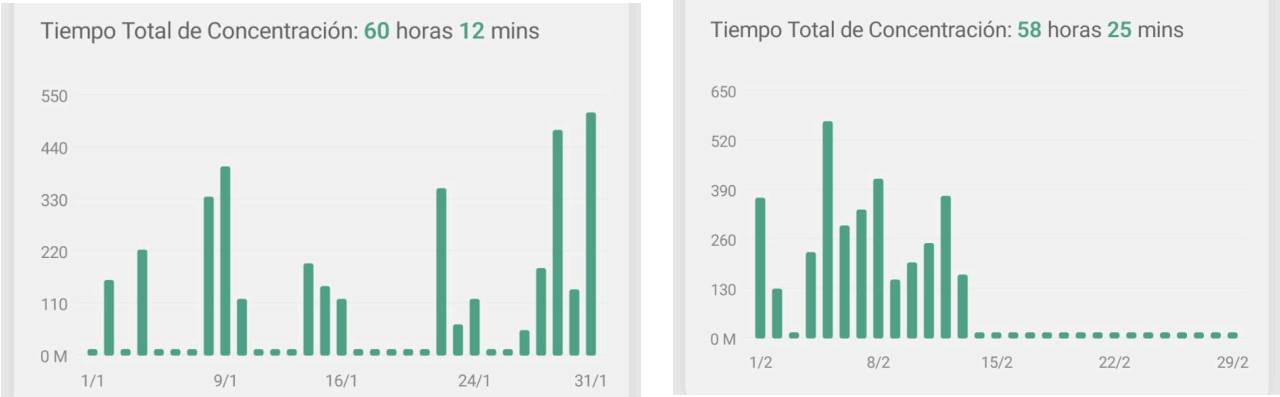


Fig. 47 Time statistics from Forest

All this time can be divided into 32 hours of research, 66 hours of development, 20 hours of documentation and 4 hours of meetings.

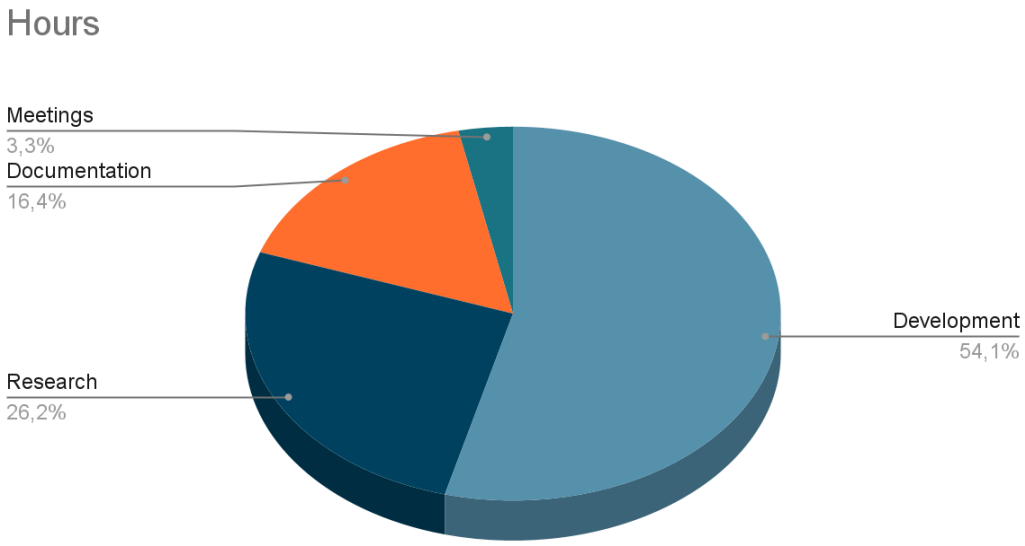


Fig. 48 Pie chart for time spent

So taking the amount a Junior database administrator gets paid by the data from the Junta de Andalucía [16], the cost of the GitHub Pro license and the laptop that has been used for the project. The total cost of this project would sum up to 4040.64 €.

Item	**Price**	**Time**	**Total Cost**
Junior Database Administrator	33,12 € / H	122 H	4040,64 €
GitHub Pro Licence	4€ / M	2 M	8 €
Laptop	800€		800 €

Future Work

The successful completion of this project's main objectives, including the conversion of OSM maps into a Neo4j database, local installation of Neo4j on Linux, interaction with the database both with and without Cypher, and the visualization of graphs and operation results, sets a solid foundation for further development. Building on this groundwork, the next logical step would be the development of a comprehensive traffic simulation tool. This expansion would utilize the Neo4j database of a city, such as Sevilla, to create an advanced simulation that can model complex traffic scenarios, including dynamic events such as road closures and traffic congestion.

This proposed extension of the project aligns with the idea of treating this work as a preliminary phase towards a more ambitious goal: constructing a traffic simulator that leverages the rich, interconnected data model provided by Neo4j. Such a simulator would not only enhance our understanding of urban traffic dynamics but also serve as a powerful tool for urban planning and traffic management.

In addition to the development of a traffic simulator, further refinement of this project could benefit from exploring additional visualization options. While the incorporation of premium Neo4j services like Neo4j Bloom presents an opportunity for enhanced data visualization, alternative tools that are not directly linked to Neo4j, yet capable of handling large datasets, should also be considered. Gephi [5], an open-source network analysis and visualization software, could potentially offer sophisticated visual insights into the data, despite its requirement for an intermediate conversion step. Similarly, implementing interactive map visualizations using Leaflet.js [6] could significantly improve the user interface for data interaction, offering a more intuitive and engaging way to explore traffic data and simulation results.

Ultimately, by advancing towards the creation of a traffic simulator and expanding the project's visualization capabilities, the future work would aim not just at enriching the current project but at setting a new standard for traffic analysis and simulation studies.

This progression would enable a deeper, more nuanced understanding of traffic patterns and contribute to the development of smarter, more efficient urban environments.

Conclusion

In conclusion, this project has successfully demonstrated the innovative integration of OpenStreetMap (OSM) data with the Neo4j graph database to address the complexities of urban traffic management. Through a methodical approach, extensive geographical data has been transformed into a dynamic and interconnected graph structure, harnessing the power of Neo4j to unlock advanced traffic analysis and simulation capabilities.

The installation and configuration of the Neo4j database on a Linux-based system, along with the development of a toolkit for database interaction both with and without the use of Cypher, have established a robust framework for traffic management solutions. Various visualization tools have been explored, enhancing the ability to interpret and utilize the data within urban planning and traffic infrastructure contexts effectively.

By achieving the primary objectives, a solid foundation for future innovations in traffic management and urban planning has been laid. The project highlights the significant potential of combining open-source geographic datasets with advanced graph database technologies, presenting a compelling case for the adoption of such integrations in addressing real-world challenges.

Looking forward, avenues for further exploration have been opened, particularly in the realm of visualization and the development of more complex traffic simulations that incorporate real-time data and constraints such as road closures and traffic congestion. The pursuit of these enhancements promises to elevate the utility and impact of traffic management systems, contributing to more efficient, safe, and sustainable urban environments.

This project stands as a testament to the transformative power of combining geographic information systems with graph theory and data processing. It enriches the academic and practical experiences, preparing the individual behind it to contribute meaningfully to the evolving landscape of software engineering, data analysis, and intelligent traffic management systems.

Annex

Code

OSM to Neo4j

```
import neo4j

import osmnx as ox



# Local bolt connection

NEO4J_URI = "bolt://localhost:7687"



# Neo4j driver with no auth

driver = neo4j.GraphDatabase.driver(NEO4J_URI, auth=None)



# Cypher queries to delete all nodes and relationships

delete_all_nodes_query = "MATCH (n) DETACH DELETE n"

delete_all_rels_query = "MATCH ()-[r]-() DELETE r"



# Execute queries to delete all nodes and relationships

with driver.session() as session:

   session.run(delete_all_rels_query)

   session.run(delete_all_nodes_query)



# Search OpenStreetMap and create an OSMNx graph
```

```
G = ox.graph_from_place("Sevilla, Andalucía, España",
network_type="drive")


gdf_nodes, gdf_relationships = ox.graph_to_gdfs(G)

gdf_nodes.reset_index(inplace=True)

gdf_relationships.reset_index(inplace=True)


# Define Cypher queries to create constraints and indexes

constraint_query = "CREATE CONSTRAINT IF NOT EXISTS FOR (i:Intersection)
REQUIRE i.osmid IS UNIQUE"

rel_index_query = "CREATE INDEX IF NOT EXISTS FOR ()-[r:ROAD_SEGMENT]-()
ON r.osmids"

point_index_query = "CREATE POINT INDEX IF NOT EXISTS FOR (i:Intersection)
ON i.location"


# Cypher query to import road network nodes GeoDataFrame

# UNWIND -> iterate over rows

# MERGE -> create or update nodes and relationships

node_query = '''

   UNWIND $rows AS row

   WITH row WHERE row.osmid IS NOT NULL

   MERGE (i:Intersection {osmid: row.osmid})

        ON CREATE SET i.location = point({latitude: row.y, longitude: row.x
}),

            i.ref = row.ref,
```

```
            i.highway = row.highway,
            i.street_count = toInteger(row.street_count)
    RETURN COUNT(*) as total
'''


# Cypher query to import road network relationships GeoDataFrame
rels_query = '''
    UNWIND $rows AS road
    MATCH (u:Intersection {osmid: road.u})
    MATCH (v:Intersection {osmid: road.v})
    MERGE (u)-[r:ROAD_SEGMENT {osmid: road.osmid}]->(v)
        SET r.oneway = road.oneway,
            r.lanes = road.lanes,
            r.ref = road.ref,
            r.name = road.name,
            r.highway = road.highway,
            r.max_speed = road.maxspeed,
            r.length = toFloat(road.length)
    RETURN COUNT(*) AS total
'''


# Function to execute constraint / index queries
def create_constraints(tx):
    tx.run(constraint_query)
```

```
    tx.run(rel_index_query)

    tx.run(point_index_query)


# Function to batch GeoDataFrames

def insert_data(tx, query, rows, batch_size=10000):

    total = 0

    batch = 0


    while batch * batch_size < len(rows):

        results = tx.run(query, parameters={'rows':
rows[batch*batch_size:(batch+1)*batch_size].to_dict('records')}).data()

        print(results)

        total += results[0]['total']

        batch += 1


# Run our constraints queries and nodes GeoDataFrame import

with driver.session() as session:

    session.execute_write(create_constraints)

    session.execute_write(insert_data, node_query,
gdf_nodes.drop(columns=['geometry']))


# Run our relationships GeoDataFrame import

with driver.session() as session:

    session.execute_write(insert_data, rels_query,
gdf_relationships.drop(columns=['geometry']))
```

```
driver.close()
```

Python Operations

```
from collections import deque

import tkinter as tk

from tkinter import Spinbox, ttk

import heapq

from py2neo import Graph, RelationshipMatcher

from math import radians, cos, sin, asin, sqrt

import matplotlib.pyplot as plt


# Function to search for nodes connected to a specific street

def find_street_nodes(graph, street_name):

   rel_matcher = RelationshipMatcher(graph)

   relationships = rel_matcher.match(r_type="ROAD_SEGMENT",
name=street_name)


   # Initialize a set for unique node identities and a list for nodes
information

   nodes_info = []


   node_cache = {}


   for rel in relationships:
```

```
        # Check if start_node is already processed
        if rel.start_node.identity not in node_cache:
            # If not in cache, fetch and store it
            start_node_info = (rel.start_node.identity,
rel.start_node.get('location', None))
            node_cache[rel.start_node.identity] = start_node_info
            nodes_info.append(start_node_info)
        else:
            # If in cache, directly append from cache
            nodes_info.append(node_cache[rel.start_node.identity])

    return nodes_info

def dijkstra(graph, start_id, end_id):
    # Fetch all nodes and relationships to build the graph structure
    nodes = list(graph.nodes.match("Intersection"))
    rels = list(graph.relationships.match(r_type="ROAD_SEGMENT"))

    # Create adjacency list representation of the graph
    adjacency_list = {node.identity: [] for node in nodes}
    for rel in rels:

adjacency_list[rel.start_node.identity].append((rel.end_node.identity,
rel['length']))
```

```
# Initialize data structures for Dijkstra's algorithm
queue = [(0, start_id)]  # Priority queue: (distance, node_id)
distances = {node_id: float('inf') for node_id in adjacency_list}
distances[start_id] = 0
predecessors = {node_id: None for node_id in adjacency_list}

while queue:
    current_distance, current_node = heapq.heappop(queue)

    if current_node == end_id:
        break  # Stop if the target node has been reached

    for neighbor, weight in adjacency_list[current_node]:
        distance = current_distance + weight
        if distance < distances[neighbor]:
            distances[neighbor] = distance
            predecessors[neighbor] = current_node
            heapq.heappush(queue, (distance, neighbor))

# Reconstruct the shortest path from end_id to start_id
path = []
current_node = end_id
while current_node is not None:
    path.insert(0, current_node)
```

```
        current_node = predecessors[current_node]


    if path[0] == start_id:  # Ensure the path is valid
        return distances[end_id], path
    else:
        return float('inf'), []


# Haversine formula to calculate the distance between two points on the
Earth's surface
def haversine(lat1, lon1, lat2, lon2):
    # Convert decimal degrees to radians
    lon1, lat1, lon2, lat2 = map(radians, [lon1, lat1, lon2, lat2])


    # Haversine formula
    dlon = lon2 - lon1
    dlat = lat2 - lat1
    a = sin(dlat/2)**2 + cos(lat1) * cos(lat2) * sin(dlon/2)**2
    c = 2 * asin(sqrt(a))
    r = 6371  # Radius of Earth in kilometers. Use 3956 for miles
    return c * r


# A* algorithm implementation
def astar(graph, start_id, end_id):
    start_node = graph.nodes.get(start_id)
```

```
    end_node = graph.nodes.get(end_id)


    # Extract latitude and longitude from the start and end nodes
    start_lat, start_lon = start_node['location'].latitude,
start_node['location'].longitude
    end_lat, end_lon = end_node['location'].latitude,
end_node['location'].longitude


    # Priority queue for A* algorithm
    open_set = [(0 + haversine(start_lat, start_lon, end_lat, end_lon), 0,
start_id, [])]  # (f_score, g_score, node_id, path)


    # Visited and cost dictionaries
    visited = set()
    g_score = {start_id: 0}


    while open_set:
        _, current_g, current, path = heapq.heappop(open_set)


        if current in visited:
            continue


        visited.add(current)
        path = path + [current]
```

```
        if current == end_id:
            return current_g, path

        neighbors = graph.relationships.match(nodes=[graph.nodes.get(current)], r_type="ROAD_SEGMENT")
        for rel in neighbors:
            neighbor = rel.end_node
            temp_g_score = current_g + rel['length']

            if neighbor.identity in visited and temp_g_score >= g_score.get(neighbor.identity, float('inf')):
                continue

            if temp_g_score < g_score.get(neighbor.identity, float('inf')):
                g_score[neighbor.identity] = temp_g_score
                f_score = temp_g_score + haversine(neighbor['location'].latitude, neighbor['location'].longitude, end_lat, end_lon)
                heapq.heappush(open_set, (f_score, temp_g_score, neighbor.identity, path))

    return float('inf'), []

def bfs(graph, start_id, end_id):
    queue = deque([(start_id, 0, [])])  # Queue: (node_id, distance, path)
```

```
    visited = set()


    while queue:
        current_node, distance, path = queue.popleft()
        if current_node == end_id:
            return distance, path + [current_node]


        if current_node in visited:
            continue


        visited.add(current_node)
        neighbors =
graph.relationships.match(nodes=[graph.nodes.get(current_node)],
r_type="ROAD_SEGMENT")


        for rel in neighbors:
            neighbor = rel.end_node
            queue.append((neighbor.identity, distance + rel['length'], path
+ [current_node]))


    return float('inf'), []


def get_street_suggestions(graph, partial_street_name):
    rel_matcher = RelationshipMatcher(graph)
```

```
    # Search for relationships with a name that begins with the partial
text
    relationships = rel_matcher.match(r_type="ROAD_SEGMENT")
    streets = set()
    for rel in relationships:
        name = rel.get('name', '')
        # Check if name is a list and take the first element, or ignore if
it's NaN or an empty list
        if isinstance(name, list):
            name = name[0] if name else ''
        if isinstance(name, str) and name and
name.lower().startswith(partial_street_name.lower()):
            streets.add(name)
    return list(streets)


def execute_find_street_nodes():
    street_name = street_name_var.get()
    nodes = find_street_nodes(graph, street_name)
    result_text.delete('1.0', tk.END)
    result_text.insert(tk.END, f"Nodes connected to '{street_name}':\n")
    for node_id, location in nodes:
        result_text.insert(tk.END, f"{node_id}: {location}\n")


def execute_dijkstra():
    start_node_id = int(start_node_var.get())
```

```
    end_node_id = int(end_node_var.get())
    cost, path = dijkstra(graph, start_node_id, end_node_id)
    result_text.delete('1.0', tk.END)
    result_text.insert(tk.END, f"Dijkstra shortest path from node
{start_node_id} to node {end_node_id}:\n")
    result_text.insert(tk.END, f"Shortest path cost: {cost} meters\n")
    result_text.insert(tk.END, f"Shortest path: {path}\n")
    plot_route(graph, path, 'red', 'Dijkstra Path')
    plt.xlabel('Longitude', fontsize='x-large')
    plt.ylabel('Latitude', fontsize='x-large')
    plt.legend()
    plt.title('Dijkstra Shortest Path', fontsize='xx-large')
    plt.show()


def execute_astar():
    start_node_id = int(start_node_var.get())
    end_node_id = int(end_node_var.get())
    cost, path = astar(graph, start_node_id, end_node_id)
    result_text.delete('1.0', tk.END)
    result_text.insert(tk.END, f"A* shortest path from node {start_node_id}
to node {end_node_id}:\n")
    result_text.insert(tk.END, f"Shortest path cost: {cost} meters\n")
    result_text.insert(tk.END, f"Shortest path: {path}\n")
    plot_route(graph, path, 'blue', 'A*')
```

```
    plt.xlabel('Longitude', fontsize='x-large')
    plt.ylabel('Latitude', fontsize='x-large')
    plt.legend()
    plt.title('A* Shortest Path', fontsize='xx-large')
    plt.show()


def execute_bfs():
    start_node_id = int(start_node_var.get())
    end_node_id = int(end_node_var.get())
    cost, path = bfs(graph, start_node_id, end_node_id)
    result_text.delete('1.0', tk.END)
    result_text.insert(tk.END, f"BFS shortest path from node
{start_node_id} to node {end_node_id}:\n")
    result_text.insert(tk.END, f"Shortest path cost: {cost} meters\n")
    result_text.insert(tk.END, f"Shortest path: {path}\n")
    plot_route(graph, path, 'darkorange', 'BFS')
    plt.xlabel('Longitude', fontsize='x-large')
    plt.ylabel('Latitude', fontsize='x-large')
    plt.legend()
    plt.title('BFS Shortest Path', fontsize='xx-large')
    plt.show()


def plot_route(graph, path, color, label):
    x_coords = []
```

```
    y_coords = []
    total_distance = 0  # Initialize total distance to 0

    # Fetch nodes
    for node_id in path:
        node = graph.nodes.get(node_id)
        x_coords.append(node['location'].longitude)
        y_coords.append(node['location'].latitude)

    # Plot the route
    plt.plot(x_coords, y_coords, color=color, label=label, linewidth=2)
    # Starting point
    plt.scatter(x_coords[0], y_coords[0], c='yellow', edgecolor='yellow',
label='Start', zorder=5)
    # Finishing point
    plt.scatter(x_coords[-1], y_coords[-1], c='lime', edgecolor='lime',
label='End', zorder=5)
    # Rest of the route
    plt.scatter(x_coords[1:-1], y_coords[1:-1], c=color)

    # Fetch relationships and annotate the distance
    for i in range(1, len(path)):
        start_node = graph.nodes.get(path[i-1])
        end_node = graph.nodes.get(path[i])
```

```
        rel = graph.relationships.match((start_node, end_node),
r_type="ROAD_SEGMENT").first()

        if rel:

            distance = rel['length']

            street_name = rel['name'] if 'name' in rel else ''

            total_distance += distance

            # Get the midpoint for the label

            mid_x = (x_coords[i-1] + x_coords[i]) / 2

            mid_y = (y_coords[i-1] + y_coords[i]) / 2

            # Annotate the segment with the distance

            plt.annotate(f"{street_name}\n{distance:.2f} m", (mid_x,
mid_y), textcoords="offset points", xytext=(0,5), ha='center')



    # Annotate the total distance

    plt.annotate(f"Total distance: {total_distance:.2f} m", (x_coords[-1],
y_coords[-1]), textcoords="offset points", xytext=(0,-15), ha='center',
fontsize=9, bbox=dict(boxstyle="round,pad=0.3", edgecolor=color,
facecolor='white'))




# Connection to Neo4j

graph = Graph("bolt://localhost:7687", auth=None)



# Get all the streets for the Spinbox

all_streets = get_street_suggestions(graph, '')
```

```
# Create the main window
root = tk.Tk()
root.title("Graph Operations GUI")

# Spinbox to select or enter the street name
street_name_var = tk.StringVar()
street_name_spinbox = Spinbox(root, values=all_streets,
textvariable=street_name_var, wrap=True)
street_name_spinbox.pack()

# Button to search for nodes
search_button = ttk.Button(root, text="Search Node",
command=execute_find_street_nodes)
search_button.pack()

# Input fields for Dijkstra and A*
start_node_var = tk.StringVar()
end_node_var = tk.StringVar()
start_node_entry = ttk.Entry(root, textvariable=start_node_var)
end_node_entry = ttk.Entry(root, textvariable=end_node_var)
start_node_entry.pack()
end_node_entry.pack()
dijkstra_button = ttk.Button(root, text="Calculate Dijkstra",
command=execute_dijkstra)
```

```
dijkstra_button.pack()

astar_button = ttk.Button(root, text="Calculate A*",
command=execute_astar)

astar_button.pack()

bfs_button = ttk.Button(root, text="Calculate BFS", command=execute_bfs)

bfs_button.pack()



# Text area for results

result_text = tk.Text(root, height=10, width=50)

result_text.pack()



# Run the application

root.mainloop()
```

References

1. (n.d.). Planet OSM. Retrieved January 31, 2024, from https://planet.openstreetmap.org/
2. (n.d.). overpass turbo. Retrieved January 31, 2024, from https://overpass-turbo.eu/
3. (n.d.). Science Direct. Retrieved January 31, 2024, from https://www.sciencedirect.com/science/article/abs/pii/S0198971516303970?via%3Dihub
4. (n.d.). Matplotlib — Visualization with Python. Retrieved January 31, 2024, from https://matplotlib.org/

5. (n.d.). Gephi - The Open Graph Viz Platform. Retrieved February 13, 2024, from https://gephi.org/
6. (n.d.). Leaflet - a JavaScript library for interactive maps. Retrieved February 13, 2024, from https://leafletjs.com/
7. (n.d.). Forest - Stay focused, be present. Retrieved February 13, 2024, from https://www.forestapp.cc/
8. (2015, September 28). Cytoscape.js. Retrieved February 11, 2024, from https://js.cytoscape.org/
9. *About Neo4j Bloom*. (n.d.). Neo4j. Retrieved February 12, 2024, from https://neo4j.com/docs/bloom-user-guide/current/about-bloom/
10. Aroca Páez, M. (2024, January 8). *OSM-to-Neo4j*. GitHub. Retrieved February 9, 2024, from https://github.com/MarioArocaPaez/OSM-to-Neo4j
11. *Bloom - Graph Database & Analytics*. (n.d.). Neo4j. Retrieved February 12, 2024, from https://neo4j.com/product/bloom/
12. *Bolt Protocol - Bolt Protocol*. (n.d.). Neo4j. Retrieved February 12, 2024, from https://neo4j.com/docs/bolt/current/bolt/
13. Camisso, J. (2020, September 15). *How To Install and Configure Neo4j on Ubuntu 20.04*. DigitalOcean. Retrieved February 11, 2024, from https://www.digitalocean.com/community/tutorials/how-to-install-and-configure-neo4j-on-ubuntu-20-04
14. de Jong, N. (2021, April 13). *15 Best Graph Visualization Tools for Your Neo4j Graph Database*. Neo4j. Retrieved February 12, 2024, from https://neo4j.com/developer-blog/15-tools-for-visualizing-your-neo4j-graph-database/

15. *Fully Managed Graph Database Service*. (n.d.). Neo4j. Retrieved February 1, 2024, from https://neo4j.com/cloud/platform/aura-graph-database/
16. *Informe de precios para perfiles profesionales en el ámbito de las Tecnologías de la Informaci*. (n.d.). Trabajo de fin de grado etsii. Retrieved February 11, 2024, from https://tfc.eii.us.es/TFG/
17. Jain, S. (n.d.). *A* Search Algorithm*. GeeksforGeeks. Retrieved February 18, 2024, from https://www.geeksforgeeks.org/a-search-algorithm/
18. Jain, S. (2024, January 11). *Find Shortest Paths from Source to all Vertices using Dijkstra's Algorithm*. GeeksforGeeks. Retrieved February 18, 2024, from https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/
19. Jain, S. (2024, February 16). *Breadth First Search or BFS for a Graph*. GeeksforGeeks. Retrieved February 18, 2024, from https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/
20. *Neo4j Aura overview*. (n.d.). Neo4j. Retrieved February 9, 2024, from https://neo4j.com/docs/aura/
21. *neo4j-contrib/osm: OSM Data Model for Neo4j*. (n.d.). GitHub. Retrieved January 30, 2024, from https://github.com/neo4j-contrib/osm
22. *neovis.js - npm*. (2023, May 17). NPM. Retrieved February 11, 2024, from https://www.npmjs.com/package/neovis.js?activeTab=explore
23. *The Py2neo Handbook - EOL ! — py2neo 2021.2*. (n.d.). Neo4j Contrib Repositories. Retrieved February 12, 2024, from https://neo4j-contrib.github.io/py2neo/
24. *What is a Graph Database? - Developer Guides*. (n.d.). Neo4j. Retrieved January 14, 2024, from https://neo4j.com/developer/graph-database/