A Case Study on Problematic Routes between Two Points in the Map of Modena City

Bilel Arfaoui

224618@studenti.unimore.it

University of Modena and Reggio Emilia

March 2024

Abstract

This article investigates the root cause of a routing issue between two points within the city of Modena. The study involves a comprehensive examination of the graph structure, validation of routing algorithms using the APOC library, and a consistency check with GraphHopper, OSRM, and Valhalla APIs through Python scripts. The findings shed light on potential anomalies in the graph data and offer insights into the effectiveness of various routing algorithms and external routing services in urban navigation scenarios.

1 Introduction

This article delves into the investigation of routing problems between two points in the graph dataset that represent the road network Modena City. Specifically, we aim to discern whether routing problems stem from data import issues, missing data elements, structural anomalies within the graph representation, or inherent limitations of the routing engines (GraphHopper[1], Valhalla[2] and OSRM[3]).

2 Graph Structure

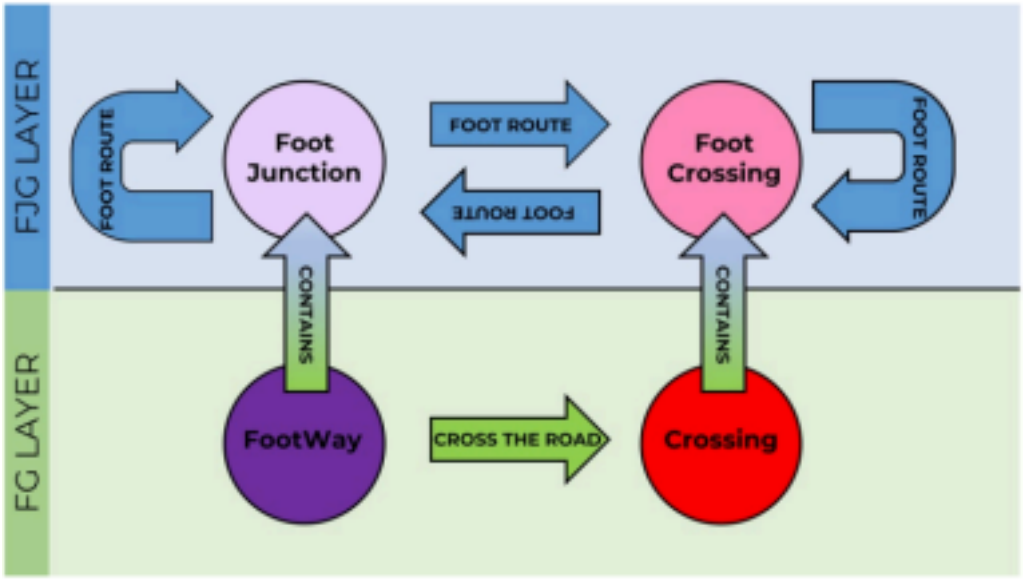
The graph that is the object of analysis consists of nodes and edges that represent a road network for pedestrians and bicycles. In this study we will consider only the pedestrian nodes and relationships.

[1] https://wiki.openstreetmap.org/wiki/GraphHopper
[2] https://wiki.openstreetmap.org/wiki/Open_Source_Routing_Machine
[3] https://wiki.openstreetmap.org/wiki/Valhalla

The graph is organized in two layers:

- Footpath Graph (FG): gives a simplified vision of the road network
- Footpaths Junctions Graph (FJG): contains further information about the road network, such as the geometry.

Figure 1: Layer Organization of the Graph[4]



Delving deeper, the FG Layer comprises:

- **FootWay nodes**: depicting pedestrian routes.
- **Crossing nodes**: signifying crossings between Footways.
- **CROSS_THE_ROAD edges**:connecting FootWay nodes with Crossing nodes.
- **CONTINUE_ON_FOOTWAY_BY_CROSSING ROAD**: linking Footway nodes connected by a nearby crossing.
- **CONTINUE_ON_LANE**: linking FootWay nodes that intersect.

Conversely, the FJG Layer encompasses:

- **FootJunction Nodes**: housing FootWay node details.
- **FootCrossing Nodes**: holding Crossing node information.
- **FOOT_ROUTE relationship**: enabling pedestrian movement between nodes, containing vital details like distance.

FJG nodes may be labeled (if a crossing is within the footpath) and include additional data such as latitude, longitude, BBOX, and community affiliation. Edges within the FJG layer

[4] Thesis “Mobility Data Management and Analytics: Digital Twins for Urban Mobility” authored by Chiara Bachechi, with Laura Po as one of the candidates and Sonia Bergamaschi serving as Director of the school.

feature essential attributes like distance, address, path type, speed, and safety level. Additionally, the FG and FJG layers are linked via a "CONTAINS" relationship, initiating from FootWay nodes to FootJunction nodes or from Crossing nodes to FootCrossing nodes.

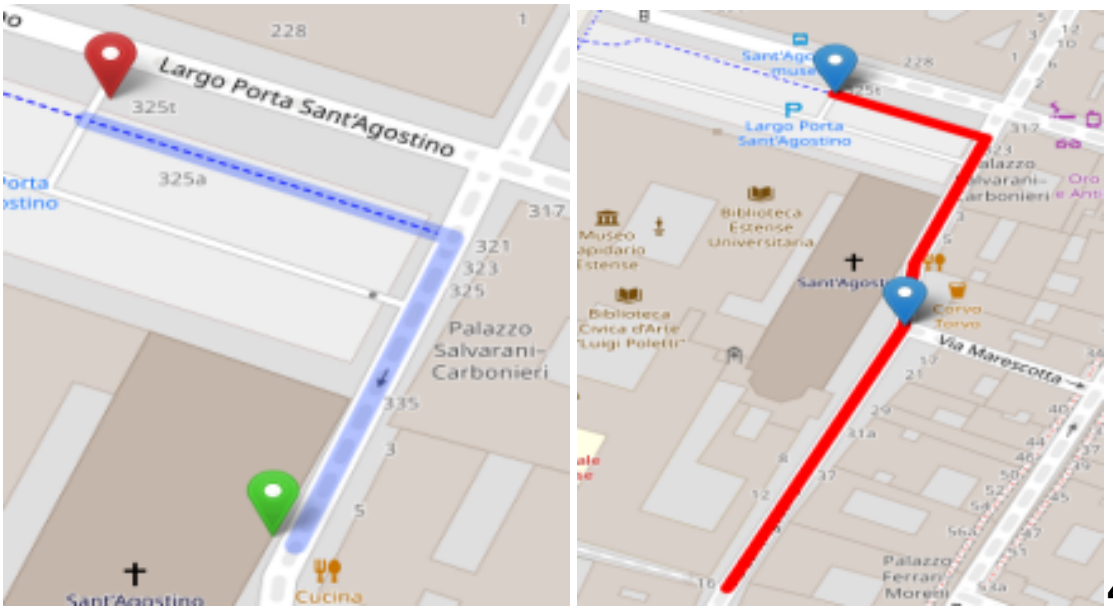
3 The problem

While utilizing our graph database, which encompasses the pedestrian network, we encountered a routing issue between two specific nodes when employing the A* algorithm[5].

Listing 1: Routing algorithm use, where C1 and C2 are two distinct nodes

```
1. CALL apoc.algo.aStar(
2. c1,
3. c2,
4. 'FOOT_ROUTE',
5. 'distance', 'lat', 'lon')
6. YIELD path, weight
```

Figure 2: The desired output(Left) and the actual output(Right)



4

Analyzing the graph

For this study we will perform a bottom-up analysis of the graph to detect anomalies. The first thing we will check is the analysis of the pedestrian road network within a BBOX (Bounding BOX), this first operation allows us to reduce the query processing time and focus better on the problem. To execute this analysis we will execute queries

[5] https://neo4j.com/labs/apoc/4.0/overview/apoc.algo/apoc.algo.aStar/

on Neo4J, the Web view given by Neo4j's GitHub repository[6] and scripts written in Python.

4.1 Node retrieval

Given the nodes in OSM[7] we need to find their counterparts residing in the graph database, as shown in the following route:

https://www.openstreetmap.org/directions?engine=fossgis_valhalla_foot&route=44.64790%2C10.92160%3B44.64840%2C10.92140#map=19/44.64813/10.92160

Figure 3: Red:44.64790, 10.92160 Green: 44.64840, 10.92140

In order to identify the nodes representing counterparts of the point under investigation, we executed a query leveraging the Neo4j Spatial library. This query utilizes proximity calculations to pinpoint nodes based on specified latitude and longitude values.

Listing 2: Queries used to find the OSM point

```
1. MATCH (n),
2. WITH point({longitude:10.92140,latitude:44.64840}) as p1,
3. point({longitude:n.lon,latitude:n.lat}) as p2
4. RETURN point.distance(p1,p2) as dist, p2.latitude,p2.longitude
5. ORDER BY dist ASC
```

Armed with these findings, we can proceed with our studies confidently. Since there is no gold standard for the geographic data, discrepancies are to be expected for the first query there is a discrepancy of 2 meters and for the second query the discrepancy is roughly 5 meters.

[6] https://neo4j.com/developer-blog/routing-web-app-neo4j-openstreetmap-leafletjs/
[7] https://www.openstreetmap.org

Figure 4:The nodes that will used for our study

With the identification of the points completed, we can now embark on a comprehensive exploration of the graph and route network, reassured by the inclusion of the corresponding OpenStreetMap (OSM) points within our graph database. Subsequent analyses will be dedicated to meticulously assessing the structural coherence, encompassing attribute values and the interconnectivity among nodes, thereby ensuring a thorough understanding of the network's integrity.

4.2 Analyzing the Area

Our initial query will involve the extraction of the road network confined within a specified bounding box (BBOX), focusing on delineating roads based on predetermined attributes and characteristics, such as:

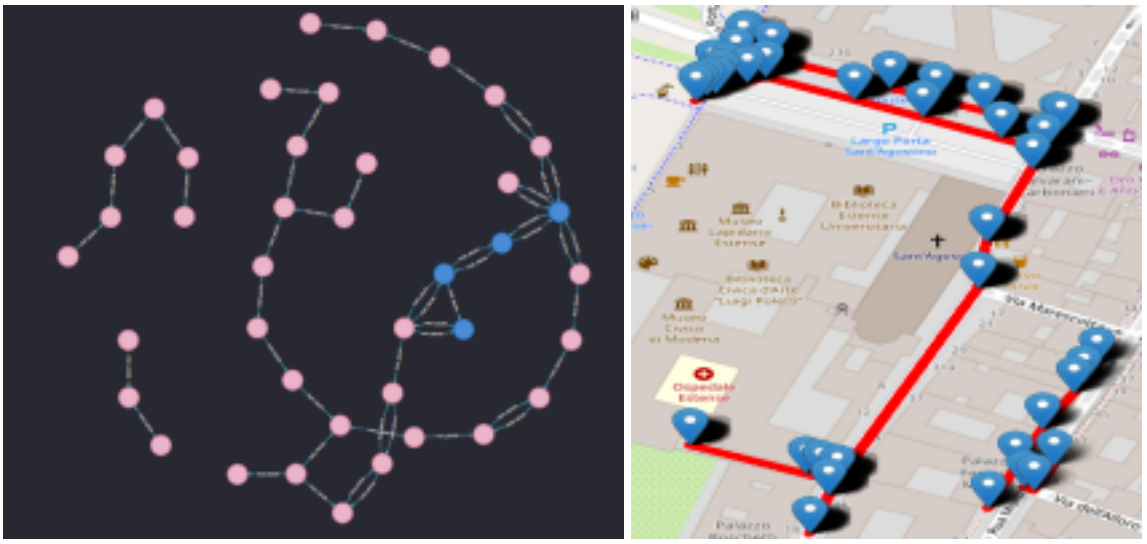
- Max Latitude=44.64868 and Min Latitude=44.46663
- Max Longitude=10.92204 and Min Longitude=10.92032

This bounding box will be the area of our study, we will analyze the road network strictly within it, doing so will allow us to execute a more effective study, assess in a more accurate way the integrity of the road network and reduce query time of execution.

Listing 3: Query used for the extraction of the road network

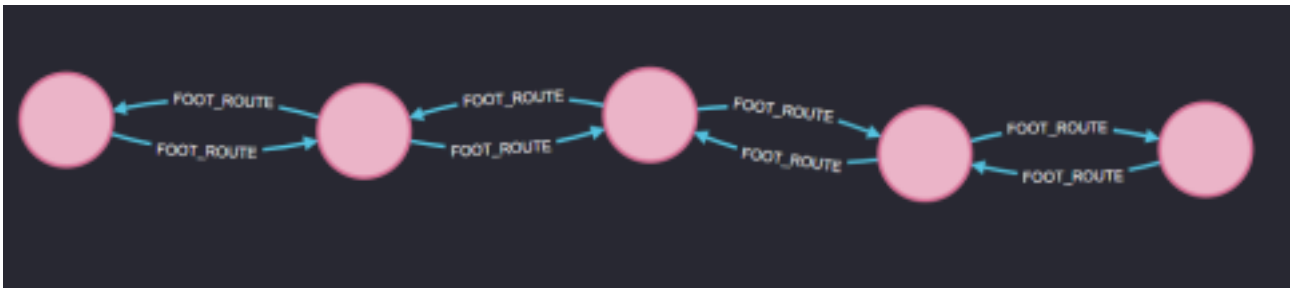
```
1. MATCH (a), (b)
2. WHERE point.withinBBox(
3. a.location,
4. point({latitude: 44.64663, longitude: 10.92032}),
5. point({latitude: 44.64868, longitude: 10.92204}))
6. AND
7. point.withinBBox(b.location, point({latitude: 44.64663, longitude: 10.92032}), point({latitude:
      44.64868, longitude: 10.92204})) AND (a:FootJunction OR a:FootCrossing)
8. AND (b:FootJunction OR b:FootCrossing)
9. MATCH path = shortestPath((a)-[:FOOT_ROUTE*]-(b))
10. WHERE a <> b AND all(node in nodes(path) WHERE
11. point.withinBBox(
12. node.location, point({latitude: 44.64663, longitude: 10.92032}),
13. point({latitude: 44.64868, longitude: 10.92204})
14. ) AND (node:FootJunction OR node:FootCrossing) )
15. RETURN *
```

Figure 5-6: Output given by Neo4j (Left) and the same network shown in the webview



Based on the provided output, we can infer that all nodes within the BBOX are interconnected in some manner. However, exceptions arise with two disconnected subgraphs, stemming from intermediate nodes falling outside the BBOX boundary. Upon closer examination, it becomes apparent that the two points in the graph are interconnected bidirectionally, emphasizing the comprehensive nature of the network connectivity.

Figure 7: Connection between node A and node B



In conclusion the portion of the graph that is the object of our studies is connected in both directions making it possible to make analysis without concerns about the direction of the relationships.

4.3 Checking for structural anomalies

Given the overall regularity of the road network, our focus shifts to examining structural anomalies in the nodes and edges. Our initial approach involves checking for abnormal values in edge attributes, particularly distance, which carries the most weight. This entails identifying edges with excessively large or negative values, as well as verifying the presence of attributes. Other structural peculiarities to consider are the direction of the relationships between nodes, since the FOOT_ROUTE is a bidirectional edge, in the database there should be 2 edges for each FOOT_ROUTE relationship. There are also other attributes that we will analyze such as the “highway” because it allows only a finite set of values.

Listing 4: Query used to check the values of the FOOT_ROUTE edges

```
1. MATCH path=(a:FootJunction)-[r:FOOT_ROUTE]-(b:FootJunction)
2. WHERE point.withinBBox(
3. a.location,
4. point({latitude: 44.64663, longitude: 10.92032}),
5. point({latitude: 44.64868, longitude: 10.92204})
6. ) AND (r.distance < 0 OR r.distance = 0 OR r.distance > 50)
7. RETURN path
```

The roads we have acquired seem to exhibit reasonable distance values, indicating that the data retrieved aligns with expected geographic distances and provides a reliable foundation for further analysis and decision-making processes.

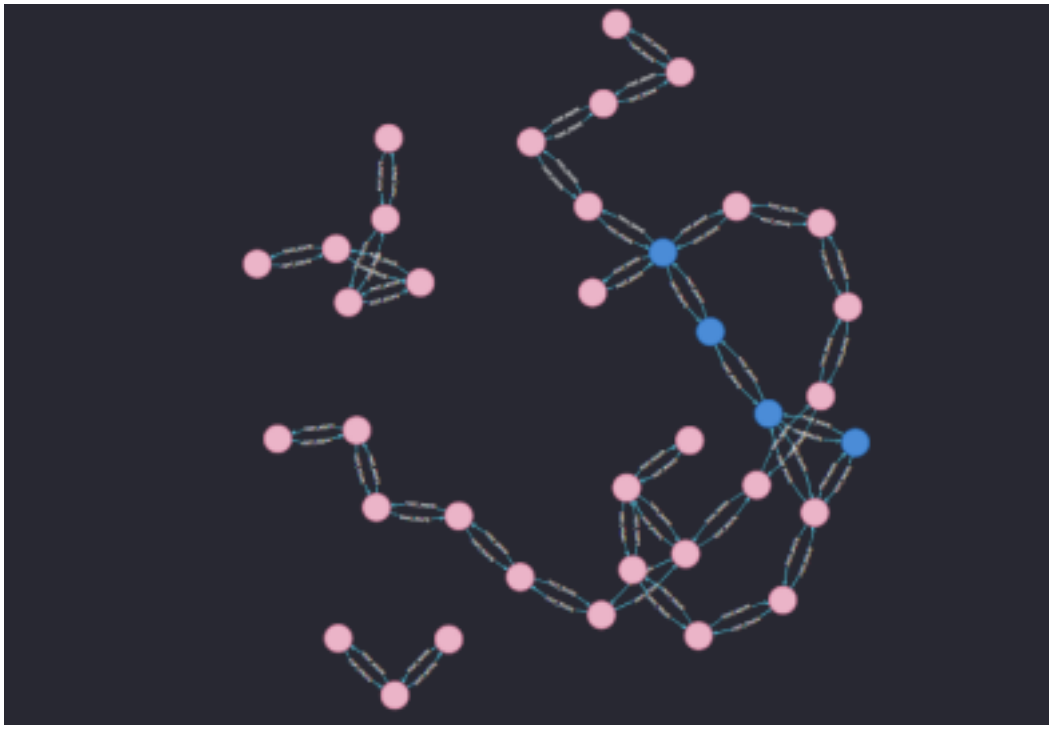
Figure 8: Query output given by the query

Other properties of the FOOT_ROUTE relationship could be crucial for selecting the optimal path. For instance, the "highway" property is significant as it denotes the type of road that pedestrians can navigate. Acceptable values for this property include "footway", "path", "track", "pedestrian", "cycleway", and "residential".

Listing 5: Query used to check if all FOOT_ROUTE relationships are traversable by pedestrians

```
1. MATCH (a)-[r:FOOT_ROUTE]->(b)
2. WHERE point.withinBBox(
3. a.location,
4. point({latitude: 44.64663, longitude: 10.92032}),
5. point({latitude: 44.64868, longitude: 10.92204})
6. ) AND
7. point.withinBBox(
8. b.location,
9. point({latitude: 44.64663, longitude: 10.92032}),
10. point({latitude: 44.64868, longitude: 10.92204})  )
AND (r.highway="footway" OR r.highway="path" OR r.highway="track" OR
        r.highway="pedestrian" OR r.highway="cycleway" OR r.highway="residential")
12. AND (a:FootJunction OR a:FootCrossing)
13. AND (b:FootJunction OR b:FootCrossing)
14. RETURN a,r,b
```

Figure 9: Query Output as shown in Neo4j

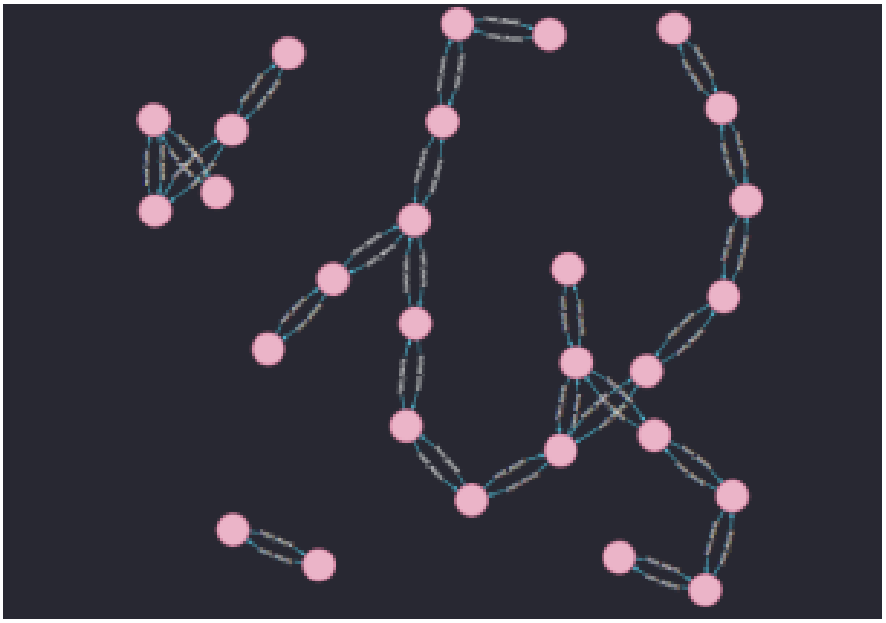


The analysis gleaned from the query output underscores the composition of the path, comprising routes characterized by acceptable "highway" values, affirming their appropriateness for pedestrian navigation. Moreover, an additional attribute deserving attention is the "danger" property, as it offers insights into the safety aspects of various routes, discerning those potentially hazardous for pedestrian transit.

Listing 6: Query used to check if all FOOT_ROUTE relationships are traversable by pedestrians

```
1. MATCH (a)-[r:FOOT_ROUTE{danger:3}]->(b)
2. WHERE point.withinBBox(
3. a.location,
4. point({latitude: 44.64663, longitude: 10.92032}),
5. point({latitude: 44.64868, longitude: 10.92204})
6. ) AND
7. point.withinBBox(
8. b.location,
9. point({latitude: 44.64663, longitude: 10.92032}),
10. point({latitude: 44.64868, longitude: 10.92204})
11. )
12. AND (a:FootJunction OR a:FootCrossing)
13. AND (b:FootJunction OR b:FootCrossing)
14. RETURN a,r,b
```

Figure 10: Query output shown in Neo4j



The result shows that the road network is mainly composed of FOOT_ROUTE relationships with danger set to the value 3, this fact may cause some issues during routing, but since the great majority of routes have the same danger value, it is safe to say that the attribute does not play a major a factor in the choice of the optimal route.

4.4 Checking with APOC algorithms

Given the specified start and end points of the route, we will undertake a comprehensive analysis utilizing routing algorithms including shortestPath[8] Dijkstra[9] and A*[10](Astar) offered by APOC[11]. Through the application of these algorithms, we aim to determine the most optimal route available.

Listing 7: Query used to get the optimal route using shortestPath

```
1. MATCH (start:FootJunction {id: "315614392"}), (end:FootJunction {id:
"1945691565"}) 2. MATCH path = shortestPath((start)-[:FOOT_ROUTE*]->(end))
3. WHERE all(node in nodes(path) WHERE (node:FootJunction OR node:FootCrossing)
AND 4. EXISTS(node.lat) AND EXISTS(node.lon) AND
5. size(apoc.coll.toSet(nodes(path))) = size(nodes(path)))
6. RETURN path
```

[8] https://neo4j.com/blog/graph-algorithms-neo4j-shortest-path/
[9] https://neo4j.com/labs/apoc/4.0/overview/apoc.algo/apoc.algo.dijkstra/
[10] https://neo4j.com/labs/apoc/4.0/overview/apoc.algo/apoc.algo.aStar/
[11] https://neo4j.com/labs/apoc/4.0/

Listing 8: Query used to get the optimal route using Dijkstra algorithm

```
1. MATCH (startNode:FootJunction {id: "315614392"})
2. MATCH (endNode:FootJunction {id: "1945691565"})
3. CALL apoc.algo.dijkstra(startNode, endNode, "FOOT_ROUTE", "length")
4. YIELD weight, path
5. RETURN path
```

Listing 9: Query used to get the optimal route using Astar algorithm

```
1. MATCH (start:FootJunction {id: "315614392"})
2. MATCH (end:FootJunction {id: "1945691565"})
3. CALL apoc.algo.aStar(start,end,
4. 'FOOT_ROUTE',
5. 'distance', 'lat', 'lon')
6. YIELD path, weight
7. RETURN path
```

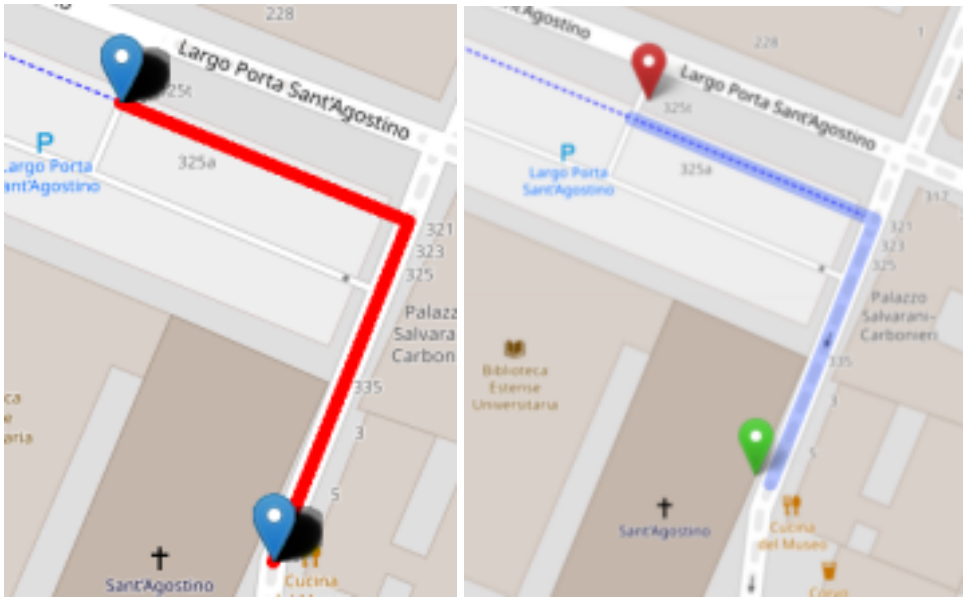
Listing 10: Variation of the Dijkstra algorithm using travel_time as weight

```
1. MATCH (start:FootJunction {id: "315614392"})
2. MATCH (end:FootJunction {id: "1945691565"})
3. CALL apoc.algo.aStar(start,end,
4. 'FOOT_ROUTE',
5. 'travel_time', 'lat', 'lon')
6. YIELD path, weight
7. RETURN path
```

In all queries conducted, it appears that the former routing algorithms consistently yield identical routes as output. This alignment with the desired correct route, featuring a distance cost of 91, validates the accuracy of our routing process. Moreover, modifying the weight parameter to duration does not result in any alterations to the output produced by our algorithms.

Figure 11: route output shown on Neo4J

Figure 12: Route output shown on the Web view(Left) and the desired route (Right)



4.5 Consistency check with Valhalla, GraphHopper and OSRM APIS

Given the result obtained with the APOC algorithms and shortestPath queries, as last check we will pass the coordinates of the start point and end point to the Valhalla, OSRM and GraphHopper APIs, to see if the distances and time results are consistent with our Graph algorithms.We will execute a Python script[12] that will do all the operations needed to fetch data from the Neo4j DB and send them to the APIs.

Listing 11: Python script that sends our point to the GraphHopper, Valhalla and OSRM APIs

```
1. apis = (
2. (Graphhopper(api_key='API-KEY'), 'foot'),
3. (Valhalla(),'pedestrian'),
4. (OSRM(),'foot'),
5. )
6. if __name__ == "__main__":
7. driver = GraphDatabase.driver(uri, auth=(user, password))
8. with driver.session() as session:
9. coords=session.execute_read(extract)
10. for api in apis:
11. client, profile = api
12. route = client.directions(locations=coords, profile=profile)
```

[12] https://github.com/NakajimaAkemi/Neo4jRoutingScript

```
13. print("Direction - {}:\n\tDuration: {}\n\tDistance: {}".format(client.__class__.__name__, 14.
route.duration,
15. route.distance))
```

Figure 13: Output given by scripts

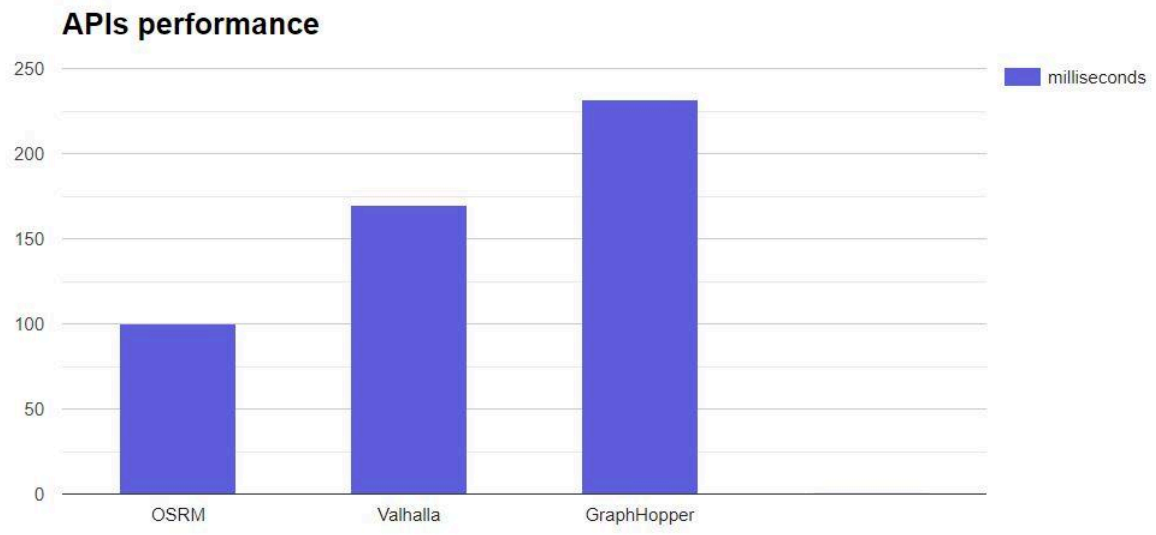
```
Direction - Graphhopper
        Distance: 27
#Geometry#
[(10.92151, 44.64824), (10.92132, 44.64829), (10.92137, 44.64839)]
################################################################################
--- 0.23209404945373535 seconds ---
################################################################################
Direction - Valhalla
        Distance: 26
#Geometry#
[(10.921513, 44.648242), (10.921323, 44.648298), (10.921371, 44.648378)]
################################################################################
--- 0.16971325874328613 seconds ---
################################################################################
Direction - OSRM
        Distance: 76
#Geometry#
[(10.92147, 44.64817), (10.9217, 44.6481), (10.9218, 44.64827), (10.92138, 44.64839)]
################################################################################
--- 0.10046100616455078 seconds ---
```

Listing 12: Results from the API and queries side by side

	Valhalla	OSRM	GraphHopper	ShortestPath	Dijkstra	Astar
distance(m)	85	85	85	91	91	91

The findings exhibit a remarkable level of consistency across the board, with negligible disparities observed among them.

Figure 14: Time performance of the 3 routing algorithms



4.6 Additional tests with other random nodes

To conduct additional studies on routing algorithms and compare the results obtained from different queries, we will execute three separate queries involving randomly selected pairs of FootJunction nodes. Below are the details for the three specified pairs:

1. Pair: 10759828564 -----> 5567795278
2. Pair: 2069714527 -----> 5567795284
3. Pair: 1958312060 -----> 2031475982

We will analyze the paths generated, the number of hops in each path, and the sum of the weights of the relationships involved in the paths. This comparative analysis will help us evaluate the performance of the routing algorithm.

Listing 13: Script that fetches random pairs of FootJunction Nodes and computes the routing

```
apis = (
    (Graphhopper(api_key='API-KEY'), 'foot'),
    (Valhalla(),'pedestrian'),
    (OSRM(),'foot'),
)
def extract_AtoB(tx):
    query = """
        MATCH (n1:FootJunction), (n2:FootJunction)
        WHERE n1 <> n2
        WITH n1, n2, rand() AS r
        ORDER BY r
        LIMIT 3
        RETURN n1.lat as lat,  n1.lon as lon, n2.lat as lat2,  n2.lon as lon2
        """
    data = []
    for record in tx.run(query):
        data.append([[record["lon"], record["lat"]],[record["lon2"], record["lat2"]]])
    return data

if __name__ == "__main__":
    driver = GraphDatabase.driver(uri, auth=(user, password))
    with driver.session() as session:
```

```
23.     coordinates=session.execute_read(extract_AtoB)
24.     for coords in coordinates:
25.         print(coords)
26.         for api in apis:
27.             client, profile = api
28.             start_time = time.time()
29.             route = client.directions(locations=coords, profile=profile)
30.             print("Direction - {}\n\tDistance: {}".format(client.__class__.__name__,
31.                                                           route.distance))
32.             print("number of steps:")
33.             print(len(route.geometry))
34.             print("--- %s seconds ---" % (time.time() - start_time))
```

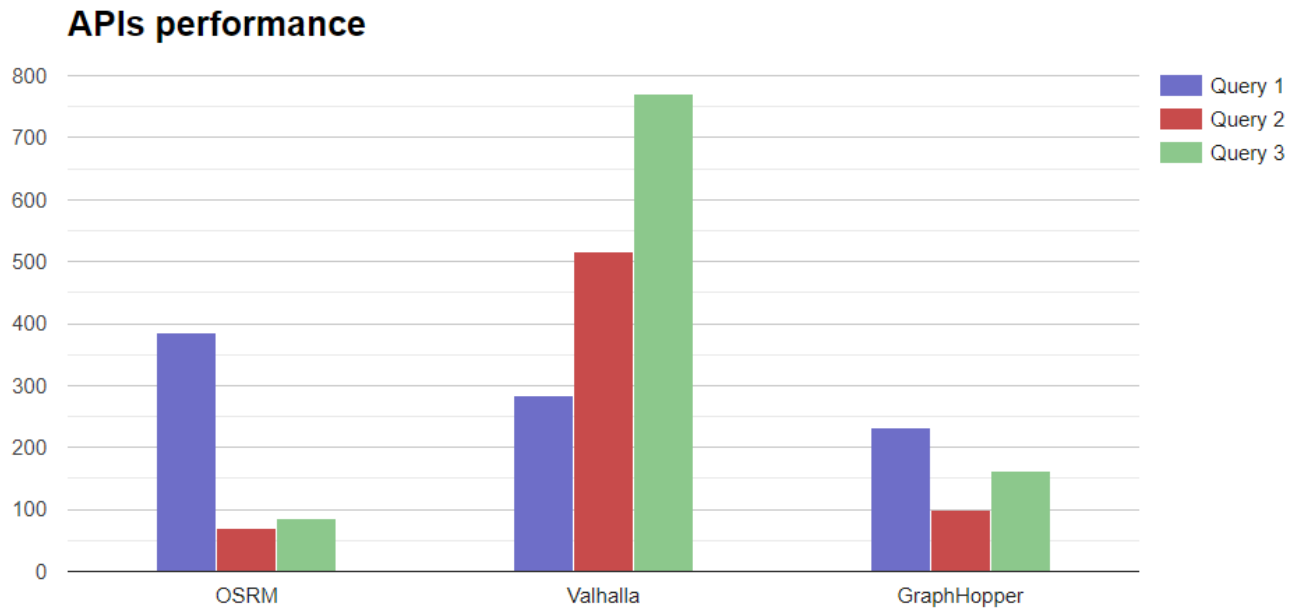
Listing 14: Results from the API and queries side by side

	Valhalla	**OSRM**	**GraphHopper**	**A***	**Dijkstra**	**ShortestPath**
hops in query 1	335	44	204	313	313	313
hops in query 2	295	42	168	282	282	282
hops in query 3	139	27	68	146	146	146

Listing 15: Distance in meters results from the API and queries side by side

	Valhalla	**OSRM**	**GraphHopper**	**A***	**Dijkstra**	**ShortestPath**
distance in query 1	10382	13440	10692	28184	28184	28184
Distance in query 2	9382	12484	9589	25631	25631	25631
Distance in query 3	3728	4330	3968	11542	11542	11542

Figure 15: Time performance of the 3 routing algorithms



Based on the outcomes, it appears that Valhalla possesses the most densely interconnected graph, which might account for its querying time. Meanwhile, OSRM demonstrates lower latency, except for the initial query, although this could potentially be attributed to a delay within the API.

5 Conclusions

In conclusion, after extensive analysis, the graph database exhibits no anomalies. The routing remains unchanged even when considering the direction or changing the weight from distance to duration. I was unable to replicate the reported error using the provided code. Moreover, routing with OSRM, Valhalla, and GraphHopper consistently aligns with local graph routing using both Dijkstra and A* algorithms. Therefore, it is likely that the routing issue stems from an abnormal loading or import process.