Advanced Routing algorithm for Modena City using Public Transports and walking paths

Giacomo Reggianini

285040@studenti.unimore.it

October 2024

Università di Modena e Reggio Emilia

Abstract

In this paper, we aim to explore the integration of Modena's public transport network with its pedestrian pathway network, a task that holds the potential to significantly enhance the accuracy and efficiency of urban mobility solutions. Our primary objective is to develop a system that more accurately calculates the distance between two points when a journey involves walking. Instead of relying on the traditional Euclidean distance, which assumes a straight line path between points, our approach will consider the actual pedestrian pathways. This method allows for a more realistic estimation of the distance one must travel on foot before reaching the public transport network, ultimately leading to better route planning and more efficient use of the available infrastructure.

To achieve this, we will create a comprehensive graph that overlays the public transport network with the pedestrian pathways. By doing so, we can compute the shortest and most practical walking distances to public transport access points, ensuring that the calculated routes reflect the true experience of pedestrians. This integration will not only improve the accuracy of distance measurements but also contribute to a more user-friendly urban navigation system.

In addition to this primary focus, our analysis will incorporate the inclusion of various constraints that influence the final route selection. These additional parameters will allow for more nuanced route optimization, beyond simple distance calculations. For example, rather than merely considering the shortest path or earliest start time, we might impose constraints such as a desired arrival time. This would allow users to specify that they need to arrive at their destination by a certain time, leading the system to prioritize routes that meet this requirement, even if it means deviating from the shortest possible path.

By addressing these factors, our work will provide a more holistic approach to urban route planning in Modena, taking into account both the real-world pedestrian experience and the diverse needs of urban travelers. Ultimately, this thesis will contribute to the development of smarter, more efficient urban mobility solutions that better serve the residents and visitors of Modena.

Sommario

1 Introduction

In this chapter, we explore the structure and integration of two distinct graph datasets used in our analysis: the Modena Footway Graph and the Modena Transport Graph. Both graphs are crucial components of our study, each representing a different aspect of Modena's urban infrastructure. By merging these graphs, we aim to create a unified system that accurately models both pedestrian and public transport routes within the city.

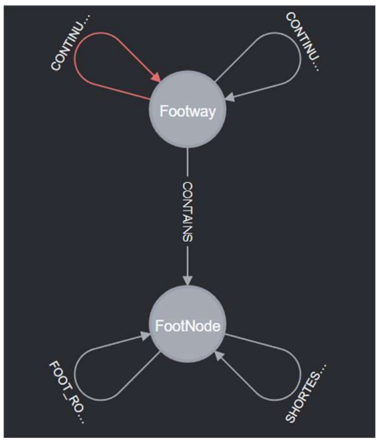
1.1 Modena Footway Graph

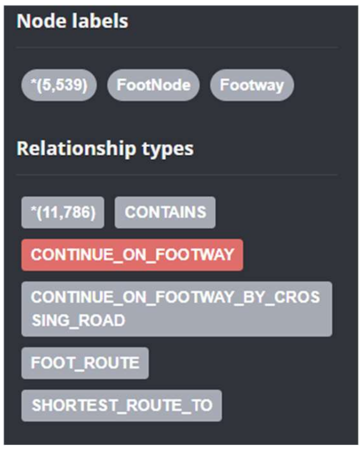
The Modena Footway Graph is designed to capture the pedestrian pathways across the city. The version used in this study is *“modena-footway-graph (no amenity)-4.4.0-12-mar-2024-18-27-23”*. This graph consists of two primary node types: FootNode and Footway. There are a total of 5,539 nodes within this graph, with 11,786 relationships connecting these nodes.

- FootNode represents specific points or intersections along the pedestrian pathways.

- Footway represents the segments of the pedestrian routes that connect these nodes.

The relationships in this graph, such as CONTAINS, CONTINUE_ON_FOOTWAY, and FOOT_ROUTE, define the possible paths and transitions between different footpaths, providing a detailed map of pedestrian movement through the city.




Node labels

*(5,539) FootNode Footway

Relationship types

*(11,786) CONTAINS

CONTINUE_ON_FOOTWAY

CONTINUE_ON_FOOTWAY_BY_CROSSING_ROAD

FOOT_ROUTE

SHORTEST_ROUTE_TO

Figure 1a/1b: Structure and property of the Modena Footway Graph, taken from Neo4J Platform

1.2 Modena Transport Graph

The Modena Transport Graph, version *"modena-transport-graph-4-4-4-neo4j-16-ott-2023-10-36-15"*, models the public transport network in Modena. This graph is more complex, containing 249,052 nodes and 738,424 relationships. The node types include Agency, Day, Route, Service, Stop, Stoptime, and Trip.

- **Agency** represents the transport providers.

- **Route** and **Service** nodes describe the different transport routes and their schedules.

- **Stop** and **Stoptime** nodes detail the individual stops and the times at which services operate.

- **Trip** nodes capture individual instances of a journey made along a route.

The relationships such as LOCATED_AT, OPERATES, PART_OF_TRIP, and WALK_TO provide the connectivity between these nodes, enabling a comprehensive representation of the public transport system's operations and routes.

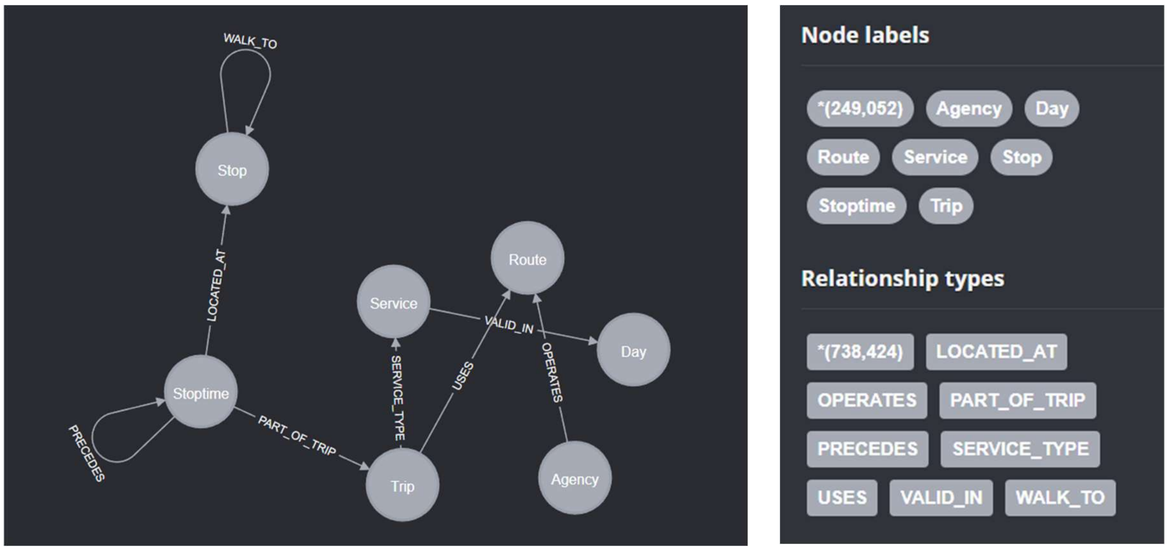


Figure 2a/2b: Structure and property of the Modena Transport Graph, taken from Neo4J Platform

1.3 Purpose of Merging the Graphs

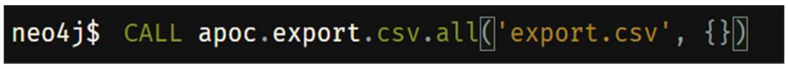
The integration of these two graphs aims to bridge the gap between pedestrian and public transport data, enabling the calculation of the most efficient routes within Modena. By merging the Footway Graph with the Transport Graph, we can more accurately calculate pedestrian access to public transport stops and identify the optimal paths for combined pedestrian and public transport journeys.

The process of merging will be carried out using CSV files to iteratively batch and import data into Neo4j. This approach ensures that large datasets can be managed effectively, maintaining the integrity and performance of the database while creating a comprehensive model of Modena's urban mobility network.

2 Merging Two Graphs in Neo4j Using CSV and Iterative Batching

2.1 Exporting a Graph to CSV Format

The first step in merging two graphs in Neo4j involves exporting one of the graphs to CSV format. This process is made possible by the `apoc.export.csv.all` procedure, which allows you to export all nodes, relationships, and their properties into a single CSV file.

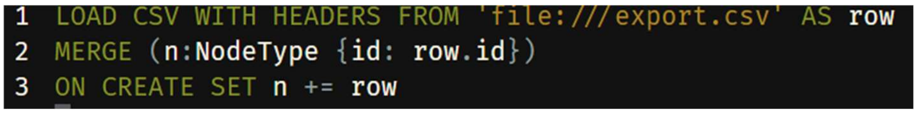


```
neo4j$  CALL apoc.export.csv.all('export.csv', {})
```

In this command, `export.csv` is the name of the CSV file where the graph data will be saved. The second parameter `{}` represents a set of options that can be configured according to your needs. For example, you can specify whether to export only nodes, relationships, or both, whether to exclude certain properties, and so on.

2.2 Importing the CSV into Another Graph

Once the first graph is exported to a CSV file, the next step is to import the data into the second graph. This can be done using the `LOAD CSV` command, which allows you to read the contents of a CSV file and create nodes and relationships within Neo4j.

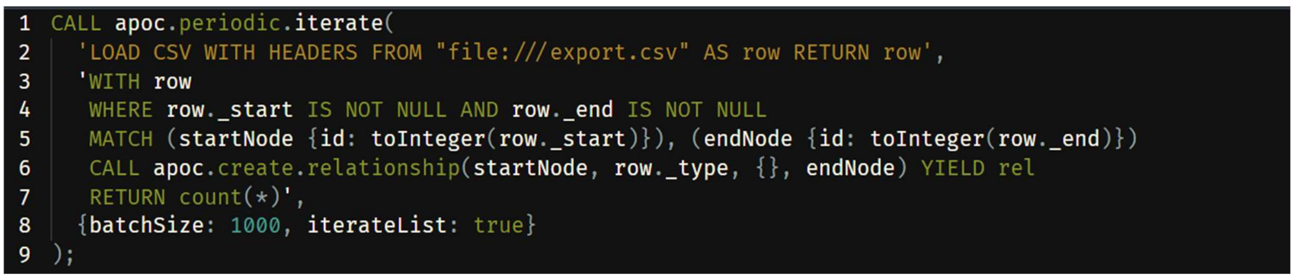


```
LOAD CSV WITH HEADERS FROM 'file:///export.csv' AS row
MERGE (n:NodeType {id: row.id})
ON CREATE SET n += row
```

In this example, the `LOAD CSV WITH HEADERS` command reads the CSV file, and `MERGE` attempts to find an existing node with the specified ID. If the node does not exist, it is created, and the properties are added using `ON CREATE SET`.

2.3 Handling Large Data Volumes with Iterative Batching

If the imported data is large, as in this case, the import process may exceed the maximum execution time (timeout). To handle this situation, Neo4j offers the ability to divide the operation into small batches using the `apoc.periodic.iterate` function. This method allows you to iterate over small groups of data, reducing the load on Neo4j and preventing timeout errors.



```
CALL apoc.periodic.iterate(
  'LOAD CSV WITH HEADERS FROM "file:///export.csv" AS row RETURN row',
  'WITH row
   WHERE row._start IS NOT NULL AND row._end IS NOT NULL
   MATCH (startNode {id: toInteger(row._start)}), (endNode {id: toInteger(row._end)})
   CALL apoc.create.relationship(startNode, row._type, {}, endNode) YIELD rel
   RETURN count(*)',
  {batchSize: 1000, iterateList: true}
);
```

In this code:

- The `apoc.periodic.iterate` command breaks the import operation into batches of 1000 rows each (specified by `batchSize: 1000`).

- The first part of the command (`'LOAD CSV WITH HEADERS FROM "file:///export.csv" AS row RETURN row'`) loads the data from the CSV file.

- The second part (`'WITH row WHERE row._start IS NOT NULL AND row._end IS NOT NULL MATCH (startNode {id: toInteger(row._start)}), (endNode {id: toInteger(row._end)}) CALL apoc.create.relationship(startNode, row._type, {}, endNode) YIELD rel RETURN count()'`) handles creating relationships between nodes, using the `_start` and `_end` IDs contained in the CSV to identify the start and end nodes of the relationship.

3 Creating Relationships Between Nodes of Two Graphs Using Geographical Distance

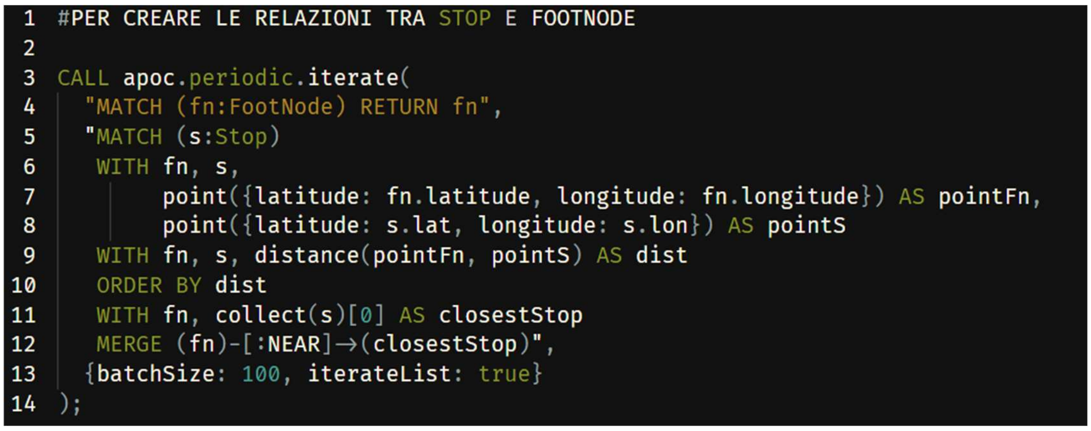
3.1 Process Description

The process of creating these relationships was based on calculating the distance between the geographical coordinates of each pair of `FootNode` and `Stop` nodes. Neo4j provides advanced tools for handling geographical data, such as the `point()` function to create geospatial points and the `distance()` function to calculate the distance between two points.

3.2 Creating Relationships with `apoc.periodic.iterate`

To execute the operation efficiently, we used the `apoc.periodic.iterate` procedure. This function allowed us to process the data in batches, reducing the risk of timeout errors and improving the overall performance of the operation.

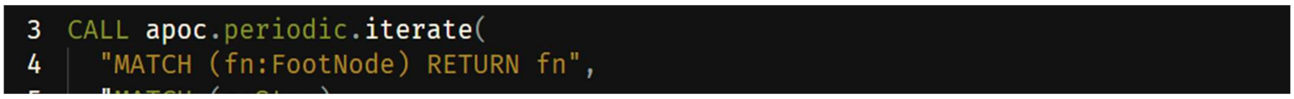
Here is the code used to create the relationships:



```
#PER CREARE LE RELAZIONI TRA STOP E FOOTNODE

CALL apoc.periodic.iterate(
  "MATCH (fn:FootNode) RETURN fn",
  "MATCH (s:Stop)
   WITH fn, s,
        point({latitude: fn.latitude, longitude: fn.longitude}) AS pointFn,
        point({latitude: s.lat, longitude: s.lon}) AS pointS
   WITH fn, s, distance(pointFn, pointS) AS dist
   ORDER BY dist
   WITH fn, collect(s)[0] AS closestStop
   MERGE (fn)-[:NEAR]->(closestStop)",
  {batchSize: 100, iterateList: true}
);
```

3.3 Code Analysis

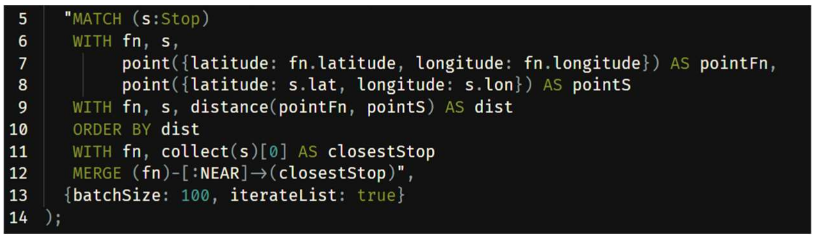
- First part:



```
CALL apoc.periodic.iterate(
  "MATCH (fn:FootNode) RETURN fn",
```

This query selects all `FootNode` nodes from the graph. Using `apoc.periodic.iterate` allows these nodes to be processed in batches, reducing the load on Neo4j.

- Second part:

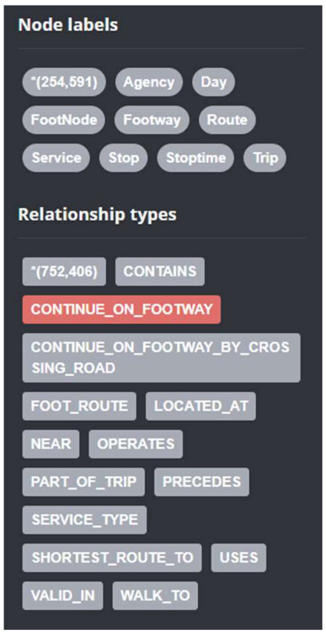


```
 "MATCH (s:Stop)
  WITH fn, s,
       point({latitude: fn.latitude, longitude: fn.longitude}) AS pointFn,
       point({latitude: s.lat, longitude: s.lon}) AS pointS
  WITH fn, s, distance(pointFn, pointS) AS dist
  ORDER BY dist
  WITH fn, collect(s)[0] AS closestStop
  MERGE (fn)-[:NEAR]->(closestStop)",
 {batchSize: 100, iterateList: true}
);
```

This section performs the actual work of calculating and creating relationships:

- MATCH (s:Stop): Selects all `Stop` nodes.
- Calculating Geographical Points: Uses the `point()` function to convert the geographical coordinates of `FootNode` and `Stop` nodes into geospatial points.
- Calculating Distance: The `distance()` function calculates the distance between `pointFn` (FootNode) and `pointS` (Stop).
- Ordering by Distance: `Stop` nodes are ordered based on their distance from `FootNode` nodes.
- Selecting the Nearest Node: `collect(s)[0]` selects the nearest `Stop` node.
- Creating the Relationship: `MERGE (fn)-[:NEAR]->(closestStop)` creates a `NEAR` relationship between `FootNode` and the nearest `Stop` node. `MERGE` ensures that the relationship is created only if it does not already exist.

Results:



Node labels

*(254,591) Agency Day FootNode Footway Route Service Stop Stoptime Trip

Relationship types

*(752,406) CONTAINS CONTINUE_ON_FOOTWAY CONTINUE_ON_FOOTWAY_BY_CROSSING_ROAD FOOT_ROUTE LOCATED_AT NEAR OPERATES PART_OF_TRIP PRECEDES SERVICE_TYPE SHORTEST_ROUTE_TO USES VALID_IN WALK_TO

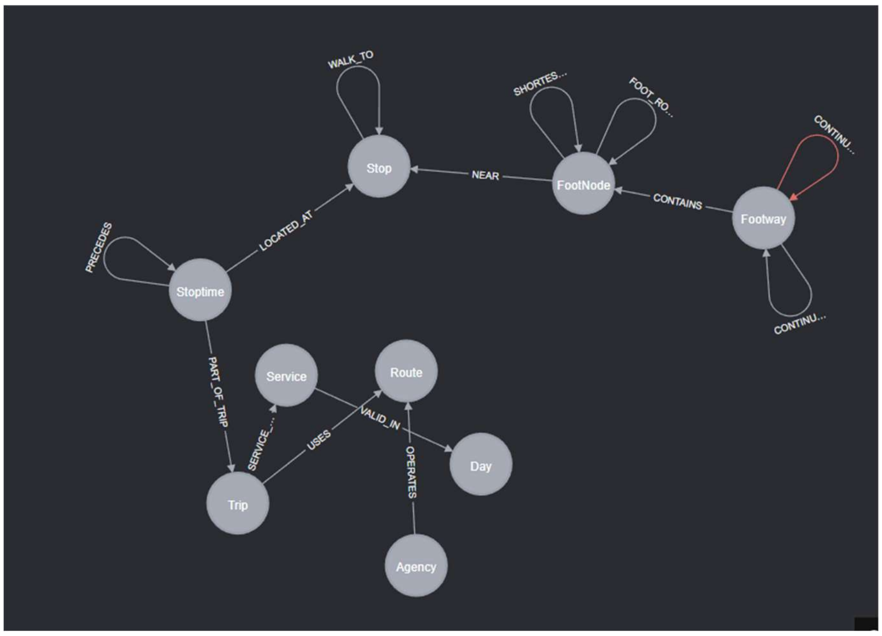


Figure 3a/3b: Structure and property of the final Modena Transport Graph, obtained from the union of the graph of public transports and the footway graph

4 Improving the Routing Algorithm for Advanced Transportation Graph

4.1 Limitations of Euclidean Distance in Routing

The Euclidean distance, while simple to compute, assumes a straight line between two points, which is rarely practical in urban environments. Roads, buildings, and natural obstacles often force pedestrians and vehicles to deviate from the idealized straight path. This becomes especially evident in multimodal routing, where transitions between walking and public transport add complexity to the pathfinding problem. In earlier models, the algorithm might compute the distance between two points purely based on Euclidean geometry, failing to capture the intricacies of actual pedestrian routes, street networks, and transit schedules. In this chapter i'll describe how i tried to improve the routing algorithm through the use of the footway graph added before.

4.2 Integrating Pedestrian Networks and Public Transport

In this enhanced routing algorithm, a hybrid approach is introduced, combining the strengths of both Euclidean distance and network-based pathfinding. Instead of calculating the entire route based on straight-line distances, the algorithm now incorporates dedicated pedestrian paths (represented as `FootNodes` in the graph) and public transportation stops (`Stop` nodes). This structure allows for more accurate modeling of real-world walking routes and their interaction with public transport systems.

4.3 Connecting to FootNodes for Accurate Walking Distances

The key innovation of the algorithm is the introduction of `FootNode` elements in the graph, which represent nodes along pedestrian pathways. Given a starting point (e.g., a user's geolocation) and an end point (e.g., the destination), the algorithm first computes the Euclidean distance to the nearest `FootNode`. By using the actual geographic coordinates of both the user and the `FootNode`, the system avoids calculating inaccurate straight-line distances across non-walkable areas like rivers or private property.

Once the nearest `FootNode` is identified, the algorithm employs existing pedestrian pathways to route the user through the city. This is done by leveraging a Neo4j graph that models footpaths and pedestrian routes. The calculated paths are based on actual distances between the nodes, ensuring the route reflects realistic walking distances.

4.4 Integrating Public Transport Stops into the Pathfinding Process

After calculating the walking distance to a nearby public transport stop (modeled as `Stop` nodes in the graph), the algorithm switches to public transport mode. This is a significant improvement over traditional routing algorithms, as it allows for an efficient combination of walking and riding. The algorithm uses predefined relationships such as `NEAR`, `CONTAINS`, and `CONTINUE_ON_FOOTWAY` to establish connections between `FootNode` and `Stop` nodes, enabling it to calculate a path that switches seamlessly between walking and public transport.

For instance, once a user reaches the nearest `Stop` node, the system uses transit routes (e.g., buses, trams) to identify the next `Stop` that brings them closer to their destination. After the public transport segment is completed, the algorithm finds the closest `FootNode` to the user's destination and resumes pedestrian routing to the final point.

4.5 Euclidean Distance and Pedestrian Network Synergy

The final refinement occurs at the start and end of the route. At both points, the algorithm employs Euclidean distance to connect the user’s exact location (latitude and longitude) to the nearest `FootNode`, ensuring the algorithm accounts for the real-world starting and ending points. This synergistic use of Euclidean distance and actual walking paths ensures that even the portions of the route outside the pedestrian network (i.e., the first few meters from the user’s location to the nearest footpath) are accurately represented.

4.6 Problem during the work

During the improvement process of the routing algorithm, a significant challenge emerged due to the low density and occasional discontinuity of the footway graph. In certain areas, the pedestrian network lacks adequate coverage, leading to scenarios where the nearest `FootNode` is located much farther from the user’s starting or destination point than expected. This results in unnecessarily long detours, as the algorithm has no choice but to connect to these distant nodes, thereby increasing the overall walking distance. This limitation in the footway graph reduces the effectiveness of the improved algorithm, particularly in regions where pedestrian paths are sparse or fragmented.

4.7 Algorithm Overview

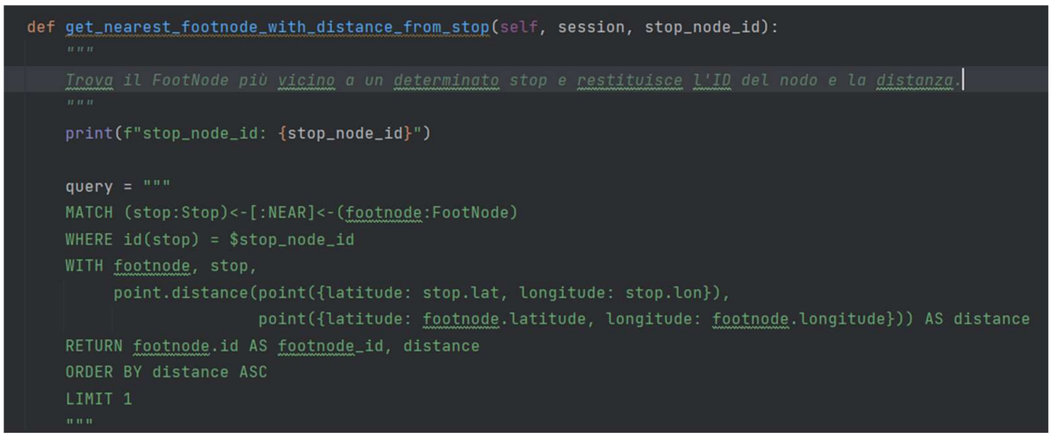
The algorithm proceeds through the following steps:

1. **Starting Point to FootNode**: Given a user's starting latitude and longitude, the algorithm calculates the Euclidean distance to the nearest `FootNode`. Once the nearest node is identified, it switches to using the pedestrian path network.

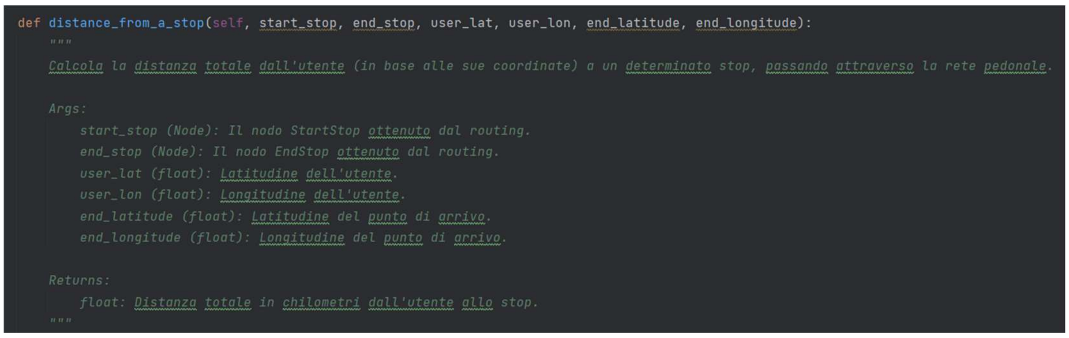
2. **Walking Path to Public Transport Stop**: The algorithm calculates the walking route through the pedestrian network to the nearest public transport stop (`Stop`). This is done by selecting the shortest route between the identified `FootNode` and the closest `Stop` that aligns with the user's intended destination.

3. **Using Public Transport**: From the `Stop`, the algorithm switches modes and uses public transport routes (e.g., buses or trams) to traverse the longer portion of the journey. The route is chosen based on existing transit data within the Neo4j graph.

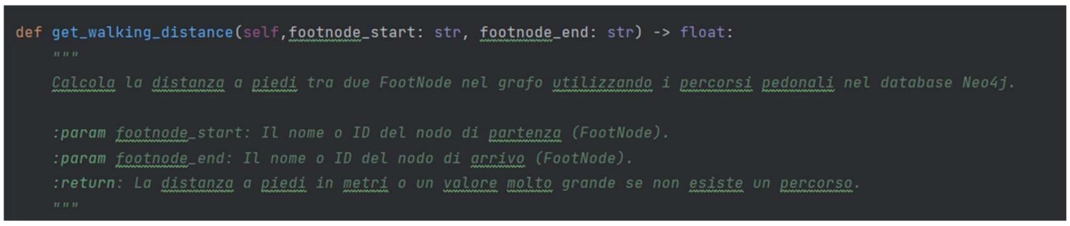
4. **End Stop to Final Destination**: After leaving the public transport system at the nearest `Stop`, the algorithm calculates the pedestrian route from that stop to the nearest `FootNode` near the destination. Finally, the Euclidean distance is used to compute the path from the `FootNode` to the exact final destination.



```
def get_nearest_footnode_with_distance_from_stop(self, session, stop_node_id):
    """
    Trova il FootNode più vicino a un determinato stop e restituisce l'ID del nodo e la distanza.
    """
    print(f"stop_node_id: {stop_node_id}")

    query = """
    MATCH (stop:Stop)<-[:NEAR]<-(footnode:FootNode)
    WHERE id(stop) = $stop_node_id
    WITH footnode, stop,
         point.distance(point({latitude: stop.lat, longitude: stop.lon}),
                        point({latitude: footnode.latitude, longitude: footnode.longitude})) AS distance
    RETURN footnode.id AS footnode_id, distance
    ORDER BY distance ASC
    LIMIT 1
    """
```



```python
def distance_from_a_stop(self, start_stop, end_stop, user_lat, user_lon, end_latitude, end_longitude):
    """
    Calcola la distanza totale dall'utente (in base alle sue coordinate) a un determinato stop, passando attraverso la rete pedonale.

    Args:
        start_stop (Node): Il nodo StartStop ottenuto dal routing.
        end_stop (Node): Il nodo EndStop ottenuto dal routing.
        user_lat (float): Latitudine dell'utente.
        user_lon (float): Longitudine dell'utente.
        end_latitude (float): Latitudine del punto di arrivo.
        end_longitude (float): Longitudine del punto di arrivo.

    Returns:
        float: Distanza totale in chilometri dall'utente allo stop.
    """
```

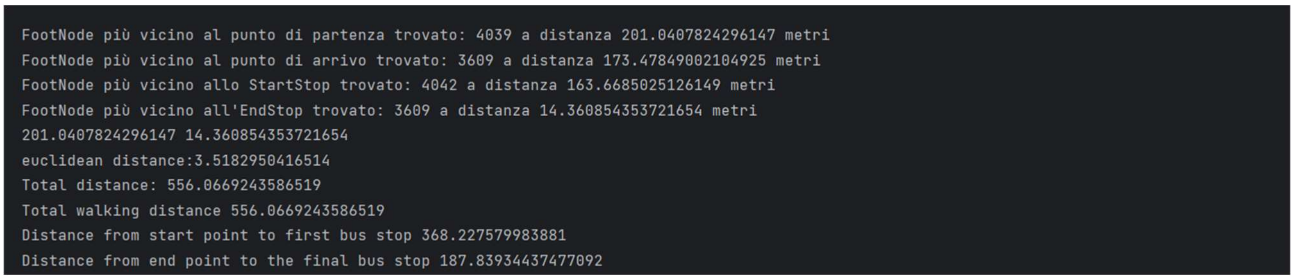


```python
def get_walking_distance(self,footnode_start: str, footnode_end: str) -> float:
    """
    Calcola la distanza a piedi tra due FootNode nel grafo utilizzando i percorsi pedonali nel database Neo4j.

    :param footnode_start: Il nome o ID del nodo di partenza (FootNode).
    :param footnode_end: Il nome o ID del nodo di arrivo (FootNode).
    :return: La distanza a piedi in metri o un valore molto grande se non esiste un percorso.
    """
```

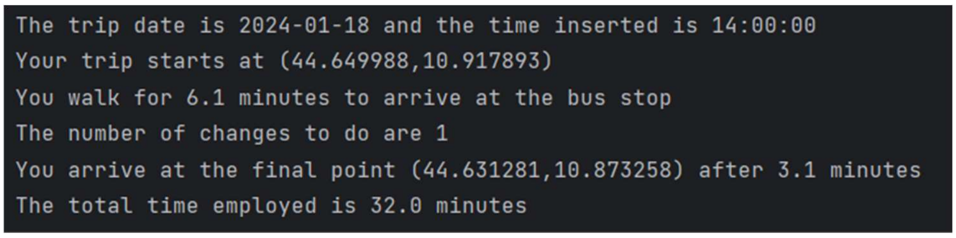
Figure 4a/4b/4c: Functions used to implement the improvements in the routing algorithm

4.8 Results

Here an example of the results of the routing algorithm working (from DIEF – Dipartimento di Ingegneria Enzo Ferrari, to Cognento - Tonini:



```
FootNode più vicino al punto di partenza trovato: 4039 a distanza 201.0407824296147 metri
FootNode più vicino al punto di arrivo trovato: 3609 a distanza 173.47849002104925 metri
FootNode più vicino allo StartStop trovato: 4042 a distanza 163.6685025126149 metri
FootNode più vicino all'EndStop trovato: 3609 a distanza 14.360854353721654 metri
201.0407824296147 14.360854353721654
euclidean distance:3.5182950416514
Total distance: 556.0669243586519
Total walking distance 556.0669243586519
Distance from start point to first bus stop 368.227579983881
Distance from end point to the final bus stop 187.83934437477092
```



```
The trip date is 2024-01-18 and the time inserted is 14:00:00
Your trip starts at (44.649988,10.917893)
You walk for 6.1 minutes to arrive at the bus stop
The number of changes to do are 1
You arrive at the final point (44.631281,10.873258) after 3.1 minutes
The total time employed is 32.0 minutes
```