UNIVERSITY OF SCIENCE,

HO CHI MINH

ENHANCING BUS POSITION ESTIMATION DURING DATA DISRUPTIONS

TECHNICAL REPORT

Task 03

submitted for the Solo Project in Semester 3 – First Year

IT DEPARTMENT

Computer Science

by

Phan Tuan Kiet

Full name: Phan Tuan Kiet

ID: 23125062

Class: 23APCS2

Tasks achieved: 03/03

Lecturer:
Professor Dinh Ba Tien (PhD)
Teaching Assistants:
Mr. Ho Tuan Thanh (MSc)
Mr. Truong Phuoc Loc (MSc)

2024

CONTENTS

1) INTRODUCTION

1.1) Problem’s Analysis

1.1.1) Overview

- Accurate bus position estimation is a critical component of intelligent transportation systems in urban environments. This accuracy becomes even more crucial in a city like Ho Chi Minh, where the efficient operation of public transportation directly impacts the daily commute of millions of residents. However, ensuring this accuracy is challenging, particularly during periods of data disruptions. This report addresses the problem of estimating the position of buses during such disruptions by leveraging historical data and advanced data processing techniques.

1.1.2) Challenges

a) Data Disruptions:

+ **Intermittent GPS Signals:** In densely built urban environments like Ho Chi Minh City, GPS signals can be obstructed by tall buildings or other structures, leading to intermittent or lost location data.

+ **Network Connectivity Issues:** The bus tracking system relies on continuous data streams from various sensors and networks. However, connectivity issues, such as network outages or low signal areas, can result in gaps in data transmission.

+ **Sensor Failures:** Hardware malfunctions or failures in the sensors that track the bus's location can also contribute to data disruptions.

b) Impact of Disruptions:

+ **Inaccurate Bus Locations:** When data disruptions occur, the system's ability to accurately determine the bus's current position is compromised. This can lead to incorrect location updates or even complete loss of tracking information.

+ **Operational Inefficiencies:** Inaccurate location data can cause delays, inefficient route management, and increased operational costs, all of which reduce the overall efficiency of the public transportation system.

+ **Passenger Dissatisfaction:** Inaccurate or delayed bus position updates can lead to longer wait times for passengers, causing frustration and a potential decline in public transportation usage.

1.2) Objectives

1.2.1) Map Data Organization:

+ The first step is to process the map data of Ho Chi Minh City, stored in the OpenStreetMap (OSM) format. This data needs to be organised into a graph structure where the nodes represent locations (points), and the edges represent the connections (roads or paths) between these locations.

+ The process of map handling needs to catch every unique feature of the .osm file, including the coordinates of each point, a list of points in a specific way, and relations between ways.

1.2.2) Historical Data Utilization:

+ The historical bus trip data, which includes sequences of edges that buses have previously travelled, will be analysed to identify patterns. Each edge represents a segment of the bus route between two points on the map.

+ By analysing this historical data, an edge matrix will be created. This matrix will capture the frequency of occurrence of each edge (road segment) between any two points. This information will be critical in estimating bus positions during data disruptions.

1.2.3) Edge Matrix Creation:

+ The edge matrix will serve as a reference for predicting the most likely path a bus would take during a disruption based on historical patterns. This matrix will be saved in a file format that allows quick retrieval of the relevant rows corresponding to any given list of edges.

+ The corresponding rows to any given list of edges will be hashed into indexes for better retrieval, hence, the rows will contain indexes as keys and the most occurring edge between any two edges as a value.

1.3) Expected Outcomes

1.3.1) Accurate Bus Position Estimation During Disruptions

a) Enhanced Accuracy:

+ The primary expected outcome is the development of a system capable of accurately estimating the position of a bus during periods of data disruptions. By utilizing historical data, the system will predict the bus's most probable location based on previously observed patterns. This approach ensures that even in the absence of real-time data, the system can maintain a high level of accuracy in estimating bus positions.

b) Mitigating Disruption Impacts:

+ With improved position estimation, the system will reduce the negative impacts of data disruptions on bus operations. This will lead to more reliable bus schedules, optimized route management, and minimized delays, thereby enhancing the overall efficiency of the transportation network.

1.3.2) Visualization of Bus Trips on Ho Chi Minh's Map

a) Comprehensive Trip Visualization:

+ The project will include the capability to visualize the entire trip of each bus within Ho Chi Minh City on the map. This visualization will be based on the sequence of edges (road segments) that the bus travels during its route. Such visualizations will offer valuable insights into the movement patterns of buses and help in identifying any anomalies or inefficiencies in the routes.

b) Historical Data Integration:

+ For each trip, the list of edges travelled will be compared against the historical data stored in the edge matrix. This matrix contains information on the most frequently occurring edge transitions between any two points in the city. By referencing this matrix, the system will identify and highlight the most probable routes taken by the bus, even during data disruptions. This will allow for a clear and precise depiction of the bus's journey, aiding in both real-time tracking and retrospective analysis.

1.3.3) Development of an Edge Matrix for Predictive Analysis

a) Edge Matrix Construction:

+ Another crucial outcome is the creation of an edge matrix derived from historical bus trip data. This matrix will serve as a predictive tool, enabling the system to estimate the most likely path a bus would take between any two edges during a disruption. Each row in the matrix will represent the frequency of occurrence of a specific edge transition, providing a statistical basis for predicting bus movements.

b) Quick Data Retrieval:

+ The edge matrix will be structured in a way that allows for quick and efficient retrieval of relevant data. Given a list of edges, the system will be able to identify the corresponding rows in the matrix, thereby determining the most frequently occurring edge transitions. This capability will be crucial for real-time decision-making and for improving the reliability of bus position estimates during periods of data disruption.

2) OSM STRUCTURE

2.1) Node

- A node represents a specific location on Earth's surface, defined by its latitude and longitude according to the World Geodetic System 1984. Each node includes at least an identification number and a pair of coordinates.

- Nodes can be used to represent standalone point features, such as a park bench or a water well.

- In addition, nodes are used to define the shape of a way. When serving as points along ways, nodes typically do not carry tags, although some may. For example, '*highway=traffic_signals'* indicates traffic signals on a road, while *'power=tower'* represents a pylon along a power line.

- Nodes can also be members of a relation, which may specify the role of the node, indicating its function within that particular set of related data elements.

2.2) Way

- A way is an ordered list of between 1 and 2,000 nodes that define a polyline. Ways are used to represent linear features such as rivers and roads.

- Ways can also represent the boundaries of areas, such as buildings or forests. In this case, the first and last node of the way will be the same, which is known as a closed way.

- It is important to note that closed ways sometimes represent loops, such as roundabouts on highways, rather than solid areas. This is typically inferred from the tags on the way. For example, land use cannot pertain to a linear feature. However, some real-life objects, such as man-made piers, can have both a linear closed way or an areal representation. The tag area yes or area no can be used to avoid ambiguity or misinterpretation.

- Areas with holes, or with boundaries exceeding two thousand nodes, cannot be represented by a single way. Instead, the feature will require a more complex multi-polygon relation data structure.

2.3) Relation

- A relation is a versatile data structure that defines a relationship between two or more data elements, such as nodes, ways, and other relations:

+ A route relation that lists the ways forming a major highway, cycle route, or bus route.

+ A turn restriction that prohibits turning from one way into another.

+ A multi-polygon that describes an area with a boundary, known as the outer way, and holes (inner ways).

- The meaning of a relation is determined by its tags, with at least the type tag being required. Other tags are interpreted based on the type tag.

- A relation mainly consists of an ordered list of nodes, ways, or other relations, which are referred to as the relation's members.

- Each member can optionally have a role within the relation. For instance, in a turn restriction, members may have roles such as 'from' and 'to,' describing the specific turn that is restricted.

2.4) Tag

- All types of data elements, including nodes, ways, relations, and changesets, can have tags. Tags provide information about the element to which they are attached.

- A tag consists of two free-form text fields: a key and a value. Each of these is a Unicode string of up to 255 characters.

- Not all elements have tags. Nodes are often untagged if they are part of a way or a relation.

3) VEHICLE’S OBJECT

3.1) Vehicles' properties

- There are four main properties of a vehicle in the file:

+ Vehicle number

+ Route_ID

+ Var_ID

+ Trip list

- For each trip list, there are smaller trips for each time stamp with corresponding edges in edgeOfPath2.

- We can turn each vehicle in the file into an object so that we can query it faster and handle more data with each vehicle.

3.2) Codes

- Class **Vehicle:**

```
class Vehicle:
    def __init__(self, vehicleNumber, routeId, varId, tripList):
        self._vehicleNumber = vehicleNumber
        self._routeId = routeId
        self._varId = varId
        self._tripList = tripList
```

- **Getter – Setter** for each property:

```
    @property
    def getVehicleNumber(self):        You, last week • Building Graph (.osm-based)
        return self._vehicleNumber
    @getVehicleNumber.setter
    def setVehicleNumber(self, vehicleNumber):
        self._vehicleNumber = vehicleNumber
```

- Class **VehicleQuery:**

```
You, last week | 1 author (You)
class VehicleQuery:
    def __init__(self, fileName):
        self.vehicleList = []
        self.loadJson(fileName)

    def loadJson(self, fileName):
        with open(f"jsonFiles/{fileName}", "r") as file:
            for line in file:
                data = json.loads(line)
                vehicle = Vehicle(**data)
                self.vehicleList.append(vehicle)
```

4) GRAPH STRUCTURE (TASK1)

4.1) Libraries

- networkx:

+ NetworkX is a Python library designed for the creation, manipulation, and study of complex networks or graphs. It provides tools for working with both undirected and directed graphs, allowing users to analyze the structure and dynamics of networks. NetworkX supports various graph algorithms, such as shortest paths, clustering, and connectivity measures, and it can handle large datasets, making it useful for research and practical applications in network science, biology, social science, and more.

- osmium:

+ Osmium is a C++ and Python library designed for working with OpenStreetMap (OSM) data. It provides efficient tools for reading, writing, and processing OSM data in various formats, such as PBF and XML. Osmium is known for its performance and scalability, making it suitable for handling large OSM datasets. It supports tasks like filtering, merging, and extracting specific elements from OSM files, as well as performing geospatial queries. Osmium is widely used in applications involving map rendering, geospatial analysis, and data processing in the OpenStreetMap ecosystem.

4.2) OSM Handler

4.2.1) Initialization

+ Initializes three dictionaries: *'self.node', 'self.ways', 'self.relations'*, which store the information extracted from the OSM file. Each dictionary is keyed by the unique ID of the respective element.

+ Code:

```
class OSMHandler(osmium.SimpleHandler):
    def __init__(self):
        super(OSMHandler, self).__init__()
        self.nodes = {}
        self.ways = {}
        self.relations = {}
```

4.2.2) Node handling

+ The *'node'* method method processes each node encountered in the OSM data. It extracts the node's latitude and longitude from its location and stores these, along with any tags associated with the node, in the *'self.nodes'* dictionary. Tags are stored as key-value pairs in a nested dictionary, making it easy to look up specific attributes of each node.

+ Code:

```
def node(self, n):
    self.nodes[n.id] = {
        'lat': n.location.lat,
        'lon': n.location.lon,
        'tags': {t.k: t.v for t in n.tags}
    }
```

4.2.3) Way handling

+ The *'way'* method processes each way element in the OSM data. It extracts a list of node references that define the way's geometry and any associated tags. This information is stored in the *'self.ways'* dictionaries. The *'nodes'* key in the dictionary contains a list of node IDs, which can be cross-referenced with the *'self.nodes'* dictionaries to obtain the full geographic details of the way.

+ Code:

```
def way(self, w):
    self.ways[w.id] = {
        'nodes': [n.ref for n in w.nodes],
        'tags': {t.k: t.v for t in w.tags}
    }
```

4.2.4) Relation handling

+ The *'relation'* method processes each relation in the OSM data. Relations in OSM are complex structures that define relationships between multiple nodes, ways, and even other relations. The method captures each member's

type, reference ID, and role within the relation, storing this information alongside any associated tags in the *'self.relations'* dictionary.

+ Code:

```
def relation(self, r):
    self.relations[r.id] = {
        'members': [(m.type, m.ref, m.role) for m in r.members],
        'tags': {t.k: t.v for t in r.tags}
    }
```

4.3) Graph Creation

- We handle the data in the *'HoChiMinh.osm'* file first, then create a graph using networkx. Finally, we add edges and nodes from the handler data. The graph is constructed by iterating through the nodes and ways extracted from the OSM file.

- While traversing the ways, we continuously add the corresponding tag data and node data to each edge. This ensures that every connection between intersections (nodes) is accurately represented. Notably, saving the *way_id* in each edge is crucial because it allows us to reference back to the original data in the OSM file whenever necessary. This is particularly important when we need to retrieve additional information or verify the integrity of the connections.

*** Edge: Connection between two intersections**

- The choice to use a directed multigraph (nx.MultiDiGraph) allows for the representation of multiple edges between the same pair of nodes, which is essential in modelling the various roads and paths in the city's transportation network. Each edge carries not only the connection information between two nodes but also the associated highway type, which plays a significant role in subsequent analyses.

- This approach enables a detailed and flexible representation of the transportation network, allowing for advanced queries and operations, such as finding the most frequently occurring edges between intersections.

- Code:

```
def create_graph_from_osm(osm_file):
    handler = OSMHandler()
    handler.apply_file(osm_file)

    G = nx.MultiDiGraph()

    total_edges = {}

    # Add nodes
    for node_id, node_data in handler.nodes.items():
        G.add_node(node_id, **node_data)

    # Add edges from ways

    """
    edges are defined as the line between two intersection
    highway is connected by intersections --> map bus_history      You, 20 hours ago • Changing edge proper
    to intersection for each edge.
    """

    for way_id, way_data in handler.ways.items():
        nodes = way_data['nodes']
        tags = way_data['tags']
        if 'highway' in tags:
            G.add_edge(nodes[0], nodes[len(nodes) - 1], wayid=way_id, **tags)
            total_edges[way_id] = nodes

    return G, total_edges
```

4.4) Total edges and total sub-edges (hashed sub-edges)

- Total_edges will store all the edges in the graph, while total_sub_edges will contain all the nodes associated with each edge. The total_edges dictionary is kept for later use in indexing. Saving the entire matrix in a straightforward manner would consume a significant amount of memory. Therefore, by indexing each edge in the graph, we can efficiently create an edge matrix where the weight represents the most frequently occurring edge between any two edges, as determined by the bus history data.

- Notice that in total_sub_edges, we assume the presence of two-way lanes, so we include both forward and backward sub-edges in the dictionary. This approach (which can be considered a useful trick) allows us to efficiently handle cases where buses travel in both directions. When traversing through the bus history, if a bus travels in both directions, we can directly pick up the corresponding edge and add it to our matrix.

- Finally, we save these dictionaries into files for later use in the edge_matrix module. This module integrates the hashed sub-edges from the bus history with the corresponding real edges in our graph. We then calculate the most frequently occurring edge using a sliding window and prefix sum method.

- Code:

```python
def save_total_edges(total_edges):
    with open("output/total_edges", 'w') as f:
        json.dump(total_edges, f)
    print("Save total_edges successfully")

def convert_sub_edges_to_edge(total_edges):
    total_sub_edges = {}
    for edge, nodes in total_edges.items():
        for i in range(len(nodes) - 1):
            total_sub_edges[nodes[i], nodes[i+1]] = edge
            total_sub_edges[nodes[i+1], nodes[i]] = edge

    return total_sub_edges

def save_total_sub_edges(total_sub_edges, filename):
    # Convert tuple keys to string representations and keep values as edge IDs
    total_sub_edges_str_keys = {f'("{k[0]}", "{k[1]}")': v for k, v in total_sub_edges.items()}

    with open(filename, 'w') as file:
        json.dump(total_sub_edges_str_keys, file)

    print("Save total_sub_edges successfully")
```

```python
G, total_edges = create_graph_from_osm(osm_file="osmFiles/HoChiMinh.osm")
total_sub_edges = convert_sub_edges_to_edge(total_edges)
save_total_sub_edges(total_sub_edges, filename="output/total_sub_edges")
save_total_edges(total_edges)      You, 1 second ago • Uncommitted changes
```

- Result:


```
{"31096786": [5758104203, 5738158912, 373543511], "32575737": [366459052, 10941379759, 6627102193, 5764885888], "32575738": [366459052,
2763159054, 2763159055, 366409867], "32575749": [366454835, 5778111885, 366427628, 5778134381, 5778132548, 366369758, 5778111884,
366408990, 5778134380, 366405484], "32575751": [5738009490, 5738009489, 3132376460, 3132376488, 3132376442, 5738009759, 3132376495,
3132376450, 5738009503, 366379008, 366384631, 366427334, 5738009505, 366447957, 366454002, 366451260], "32575754": [366442479, 2477301439,
366473779, 3202547145, 5735403820], "32575768": [366369613, 5795144851, 366418963, 366373068, 5753228946, 5795144815, 366378718,
5795144828, 5795144841, 5795144839, 4107389720, 5752843056, 366442490, 5753023085, 366387538, 5793457602, 5752843088, 5753023079,
366444706, 5752843089, 5795145096, 5752843085], "32575769": [366379023, 366407925, 3110867010, 8360852434, 3110867029], "32575784":
[5811673555, 366453642, 366419656, 366454788, 7393930257, 7393930258, 7393930259, 7393930260, 7393930261, 7393930262], "32575788":
[366391224, 2925962164, 5733448362, 1671468698, 366418696], "32575790": [366414641, 5733448375, 2030638785, 2030639268, 2030639086,
5737826545, 2030638786, 366404504], "32575792": [366452436, 5737751039, 366379167, 366404193, 366390786, 2030639087, 2030639497,
1516645265, 3631214263, 1516645264, 1516645263, 1516645262, 1516645259, 1516645256, 1516645250, 1516645253], "32575794": [370818177,
696860119], "32575795": [366452436, 5737751041, 366478400, 5733055246, 366467734, 5881087223, 366442186, 366390118, 3631214262, 366446042,
```

5) ALGORITHMS (TASK2 + TASK3)

5.1) Data hashing & Data cleaning

- As we know, the bus_history data does not represent the real edges directly; instead, it contains sub-edges of an edge. Therefore, we will map these sub-edges to their corresponding real edges in each path and then calculate the most frequently occurring edge.

- Hint for algorithms: Given that an edge in a path consists of sub-edges, we can optimize the process using the sliding window method. This method allows us to process smaller segments of sub-edges rather than traversing all sub-edges in a single edge at once. However, for 100% accuracy, we will also implement a naive approach. Both methods will be described in detail later. For now, we will focus on data cleaning and hashing.

- First, we have to load the total_sub_edges and total_edges that we prepared before in the module graph:

```
def load_total_sub_edges(filename):
    total_sub_edges = {}

    with open(filename, 'r') as file:
        data = json.load(file)
    for key, value in data.items():
        node1, node2 = key.strip('()').split(', ')
        node1 = node1.strip('"')
        node2 = node2.strip('"')
        total_sub_edges[(node1, node2)] = value

    print("Load total_sub_edges successfully")

    return total_sub_edges

def load_total_edges():
    with open("output/total_edges", "r") as file:
        total_edges = json.load(file)

    print("Load total_edges successfully")

    return total_edges
```

- Then, for sub_edges in the bus_history, we will hash/map it to the edge in our graph (‘highway’):

```
def process_intermediate_edges(trip, total_sub_edges):
    edge_freq = defaultdict(default_factory)
    sub_edges = trip['edgesOfPath2']

    # Map to edge: connection between two intersections
    edges = []
    for sub_edge in sub_edges:
        if tuple(sub_edge) not in total_sub_edges:
            continue
        edge = total_sub_edges[tuple(sub_edge)]
        if edge not in edges:
            edges.append(edge)
    n = len(edges)
```

- Notice that:

+ If any sub_edge is not in total_sub_edges → that means the bus goes through some unidentified points that don’t exist in the HoChiMinh.osm. However, these unidentified edges are not a lot so our function runs smoothly.

+ Also, in the load total_sub_edges, I want to convert data so that we can hash it easily:

Example: total_sub_edges[tuple([‘sub_edge1’, ‘sub_edge2’])] = edge_in_our_graph.

5.2) Naive Implementation

- After the cleaning and hashing/mapping process, we apply our algorithms normally with the edges in our graph ('highway'). In this stage, I will divide it into smaller sub-tasks:

+ Finding the algorithms to find the most occurring intermediate edge.

+ Saving the matrix efficiently so that we can get the corresponding row in our matrix with a list of edges.

5.2.1) Finding the most occurring intermediate edge

- For this method, we will parse the data from the JSON file, then loop normally through each trip in each vehicle. Continue with looking for each pair of edges and count the intermediate edges between. Doing this for the trip, we will get the answer for each pair and then save it in a file.

- Notation:

+ E: total edges.

+ T: total trips.

- Time complexity: $O(E^3)$.

- Space complexity: $O(T + E^2)$.

- This method is inefficient and impractical for laptops with small RAM or not powerful CPUs to handle large data.

- Code:

```
# O(N^3): Max accuracy (Naive Approach)
for i in range(n - 2):
    edge_i = edges[i]
    for j in range(i + 2, n):
        edge_j = edges[j]
        for k in range(i + 1, j):
            intermediate_edge = edges[k]
            edge_freq[(edge_i, edge_j)][intermediate_edge] += 1
```

- Comments: Although this method is inefficient, it gets the most accurate information. However, for the most occurring edge between any two given edges, it can be approximated so that we can pass through some of the redundant works of checking. A little bit tradeoffs in accuracy and time complexity will be discussed later on for the optimization process.

5.2.2) Saving matrix

- In the naive implementation, once we have identified the most occurring intermediate edge between any two given edges, the next step is to save this information in an edge matrix. This matrix is essentially an adjacency matrix where each entry *(i, j)* represents the weight, which corresponds to the most frequently occurring intermediate edge between edges *i* and *j*.

- Space complexity: $O(E^2)$.

- Comments: Given that many pairs of edges might not have any intermediate edge, the adjacency matrix will be sparse. In such cases, storing the matrix as a sparse matrix (only storing non-zero entries) could save significant space. While this approach ensures completeness, it also results in storing a lot of redundant or unnecessary information, which could impact both storage space and processing efficiency.

5.3) Optimizations - Proposals

- Given the inefficiencies of the naive implementation, it's essential to explore optimizations that reduce both time and space complexity. Below is a discussion of an optimized approach for finding the most occurring intermediate edge using a combination of sliding windows and prefix sum updating with efficient matrix structure saving.

5.3.1) Finding the most occurring intermediate edge

**** Sliding Windows + Prefix Sum updating**

- Overview:

+ The goal of this optimization is to reduce the time complexity associated with finding the most occurring intermediate edge between pairs of edges. The naive method checks each pair of edges and counts all intermediate edges, leading to an inefficient $O(E^3)$ time complexity. Instead, the sliding window technique, combined with prefix sum updating, allows us to perform this task more efficiently.

- Sliding Windows:

+ **Concept:** The idea behind sliding windows is to limit the range of edges that are considered potential intermediate edges. Instead of checking every possible edge as an intermediate, we only check those within a specified window size.

+ **Implementation:** For each edge *edge_i,* we consider edges within a window size, that are at least two edges away (to ensure they are intermediates). The code snippet provided in this section shows how the sliding window method is implemented.

+ Code (Sliding windows):

```
# O(N * Windows^2) (Sliding Windows)
windows = 10
for i in range(n):
    edge_i = edges[i]
    for j in range(i + 2, min(n, i + windows)):
        edge_j = edges[j]
        for k in range(i + 2, j):
            intermediate_edge = edges[k]
            edge_freq[(edge_i, edge_j)][intermediate_edge] += 1
```

- Prefix Sum Updating:

+ **Concept:** To further optimize the process, we use prefix sums to keep track of the count of intermediate edges efficiently. This avoids recalculating the counts for every pair of edges, significantly reducing the computational burden.

+ **Implementation:** As we slide the window across the edges, we maintain a running count of intermediate edges using a prefix sum approach. For each pair *(edge_i, edge_j)*, we update the frequency of each intermediate edge found in the prefix sum. This allows us to update the counts for multiple pairs simultaneously, rather than one by one.

+ **Time complexity:** (*W: Window_Size)*

* Best case: $O(E * W)$.

* Average case: $O(E * W^2) - C(constant)$.

* Worst case: $O(E * W^2)$.

+ Code (Sliding Windows + Prefix Sum Updating):

```
# O(N * Windows) (Average case), O(N * Windows^2) (Worst case) (Sliding Windows + Prefix Sum)
windows = 10 # Can be enlarged (based on computer's cpu performance)
for i in range(n):
    edge_i = edges[i]
    inter_counts = defaultdict(int)
    for j in range(i + 1, min(n, i + windows + 1)):
        edge_j = edges[j]

        if j > i + 1:
            for middle_edge, count in inter_counts.items():
                edge_freq[(edge_i, edge_j)][middle_edge] += count

        inter_counts[edge_j] += 1

return {k: dict(v) for k, v in edge_freq.items()}
```

5.3.2) Parallel Processing and Merge Results

- As datasets grow in size, the need for efficient data processing becomes increasingly important. One of the most effective ways to improve performance is through parallel processing. By distributing tasks across multiple CPU cores, we can significantly reduce the time required to process large amounts of data. This section explores how parallel processing can be leveraged to optimize the process of finding the most occurring intermediate edge between pairs of edges, followed by merging the results.

- Parallel processing:

+ **Concept:** Parallel processing involves dividing the task into smaller, independent sub-tasks that can be executed simultaneously across multiple CPU cores. This approach is particularly effective when dealing with large datasets, such as a JSON file containing trip data for numerous vehicles.

+ **Implementation:** In this approach, the trips from each vehicle are processed in parallel. Each parallel task focuses on a subset of trips, processing them to find the most occurring intermediate edges. The results from each parallel task are then combined to form the final result.

+ Code:

```
def parse_raw_data(json_file, total_sub_edges):
    data = []

    try:
        with open(json_file, 'r') as f:
            for line in f:
                data.append(json.loads(line.strip()))

        print(f"Loaded {len(data)} vehicles from {json_file}")

        # Process the trips in parallel
        with Pool(cpu_count()) as pool:
            try:

                trip_edges_freq = pool.starmap(process_intermediate_edges,
                                               [(trip, total_sub_edges) for vehicle in tqdm(data) for trip in vehicle['tripList']])

                print(f"Processed {len(trip_edges_freq)} trips")

                return trip_edges_freq

            except Exception as e:
                print(f"Error during parallel processing: {e}")
                raise
```

+ **Efficiency:** This method allows the processing of large datasets much faster than sequential processing, as it takes advantage of multiple cores to handle multiple trips simultaneously.

+ **Scalability:** This method scales well with the number of CPU cores available, making it suitable for large datasets and high-performance computing environments.

- Merging Results:

+ **Concept:** After processing each trip in parallel, the results need to be combined into a cohesive whole. This involves merging the frequency counts of intermediate edges across all processed trips to produce the final edge frequency matrix.

+ **Implementation:** Once all trips are processed, the individual results are aggregated. The aggregation involves summing the counts of intermediate edges for each pair of edges across all trips.

+ Code:

```
def merge_edges_frequency(edge_freqs_list):
    merged_freq = defaultdict(lambda: defaultdict(int))
    for edge_freq in edge_freqs_list:
        for (edge_i, edge_j), inter_edges in edge_freq.items():
            for intermediate_edge, count in inter_edges.items():
                merged_freq[(edge_i, edge_j)][intermediate_edge] += count
                #print(f"From {edge_i} to {edge_j} : {intermediate_edge, count}")
    return merged_freq
```

- Summary:

+ Parallel processing, combined with an efficient merging process, offers a powerful optimization for handling large datasets. By distributing the workload across multiple CPU cores, we can significantly reduce the time required to find the most occurring intermediate edges between pairs of edges. The final merged result provides a comprehensive view of the data, ready for further analysis or storage in an adjacency matrix.

+ This approach is highly adaptable, making it suitable for various real-world applications, from transportation network analysis to large-scale data processing tasks. By leveraging the full computational power of modern multi-core systems, this method ensures that even the most demanding datasets can be processed efficiently and accurately.

+ This method combined with Sliding Windows and Prefix Updating would be a very efficient approach!

5.3.3) Saving matrix efficiently (Indexing)

- After optimizing the process of finding the most occurring intermediate edges and processing them in parallel, the next step is to store the resulting matrix efficiently. Given the potentially large size of the edge frequency matrix, an efficient storage strategy is crucial to avoid excessive memory usage. This section outlines how to achieve this by hashing edges into indices and using a sparse matrix representation to save only non-zero elements.

- Hashing Edges into Indices:

+ **Concept:** Instead of storing edges as tuples directly in the matrix, which can be space-inefficient and slow for lookups, we hash each edge into a unique index. This index can then be used as the row and column identifiers in the matrix.

+ **Implementation:** The *get_edge_index* function iterates through the dataset, assigns a unique index to each edge, and stores this mapping in a dictionary. This indexed representation allows for efficient matrix operations.

+ Code:

```
def get_edge_index(total_edges):
    edge_index = {}
    index_count = 1

    for edge, nodes in total_edges.items():
        edge_index[edge] = index_count
        index_count += 1

    return edge_index

def from_index_to_edge(total_edges, index):
    edge_index = load_edge_index(index_file)
    for edge in total_edges:
        if edge_index[edge] == index:
            return edge
    return None
```

+ Edge Indexing: Each unique edge in the dataset is assigned a sequential index starting from 1. This index is used later to map edges to matrix rows and columns.

+ Efficiency: By using a dictionary to store the edge-to-index mapping, lookups and insertions are efficient, ensuring the indexing process is fast even for large datasets.

- Saving Only Non-Zero Elements in a Sparse Matrix

+ Concept: The edge frequency matrix is typically sparse, meaning lots of entries are zero (no intermediate edge between the corresponding edges). To save space, we store only the non-zero entries using a sparse matrix representation. This is both memory-efficient and allows for faster matrix operations.

+ Implementation: The function constructs a sparse matrix by iterating through the edge frequency data, recording only the most frequent intermediate edges for each edge pair.

+ Code matrix creation:

```
def create_inter_edges_matrix(edge_freq, edge_size, edge_index):
    row_indices = []
    col_indices = []
    data = []

    for (edge_i, edge_j), inter_edges in edge_freq.items():
        most_frequent_edge = max(inter_edges, key=inter_edges.get)
        row_indices.append(edge_index[str(edge_i)])
        col_indices.append(edge_index[str(edge_j)])
        data.append(edge_index[str(most_frequent_edge)])

        # if (edge_index[edge_i] == 1):
        #     print(f"From {edge_i, edge_index[edge_i]} to {edge_j, edge_index[edge_j]} : {most_frequent_edge, edge_index
        [most_frequent_edge]}, ")

    inter_edges_matrix = sparse.csr_matrix((data, (row_indices, col_indices)), shape=(edge_size+1, edge_size+1))

    return inter_edges_matrix
```

+ Sparse Matrix Construction: The function uses the compressed sparse row (CSR) format from the *'spicy.sparse'* module to store the matrix. This format is particularly efficient for storing sparse matrices where most elements are zero.

+ **Matrix Population**: For each pair of edges *(edge_i, edge_j)*, the function finds the most frequent intermediate edge and adds its index to the corresponding position in the sparse matrix.

+ **Memory Efficiency:** By storing only non-zero elements, the sparse matrix significantly reduces the amount of memory required, especially for large datasets where the number of non-zero entries is small compared to the total possible pairs.

+ Code loading and saving matrix:

```python
def save_inter_edges_matrix(inter_edges_freq, matrix_size, edge_index):
    sparse_matrix = create_inter_edges_matrix(inter_edges_freq, matrix_size, edge_index)
    sparse.save_npz(matrix_file, sparse_matrix)
    print("Save edge_matrix successfully")

def load_inter_edges_matrix(matrix_file):
    sparse_matrix = sparse.load_npz(matrix_file)
    print("Load edge_matrix successfully")
    return sparse_matrix

def save_edge_index(index_file, edge_index):
    with open(index_file, 'wb') as f:      You, 2 days ago • Finish module intermediate_frequency_matrix
        pickle.dump(edge_index, f)
    print("Save edge_index successfully")

def load_edge_index(index_file):
    with open(index_file, 'rb') as f:
        edge_index = pickle.load(f)
    print("Load edge_index successfully")
    return edge_index
```

- Summary:

+ Efficiently saving the matrix involves two key steps: hashing edges into unique indices and using a sparse matrix representation to store only non-zero elements.

+ This approach not only reduces memory usage but also improves performance, making it possible to handle large datasets effectively. By focusing on space efficiency and leveraging sparse matrix techniques, this method provides a robust solution for storing and analyzing edge frequency data in transportation networks or similar domains.

6) RESULTS

6.1) Get rows from the matrix

- In the context of analyzing historical bus trip data, it's essential to extract specific rows from the edge matrix efficiently. The *'get_rows_from_matrix'* function is designed to achieve this by extracting rows corresponding to a given list of edges, using an edge index for quick lookup.

- Logic:

+ Starting by initializing a defaultdict of dictionaries, which will store the relevant rows from the matrix. The function then iterates through each edge in the provided *'edge_list'*, converting the edge to string to ensure consistent formatting. For each edge, it checks if the edge exists in the *'edge_index'*, a dictionary mapping edges to their corresponding row indices in the matrix. If the edge is found, the function retrieves the corresponding row index and accesses the associated row in the edge matrix.

+ It then extracts the non-zero entries from this row, representing connected edges and their respective weights, and stores them in the rows dictionary under the appropriate row index. After processing all edges in the list, the function returns a dictionary containing only the extracted rows, effectively filtering the matrix to include only the relevant data. This approach optimizes both memory usage and computation time by focusing on necessary information and leveraging efficient lookups.

- Code:

```
def get_rows_from_matrix(edge_list, edge_matrix, edge_index):
    rows = defaultdict(dict)
    for edge in edge_list:
        if edge in edge_index:
            idx = edge_index[edge]
            current_row = edge_matrix[idx]
            for col, val in zip(current_row.indices, current_row.data):
                rows[idx][col] = val
    return dict(rows)
```

6.2) Get the corresponding row for each edge

6.2.1) Initialization

- **Processing edge_index, matrix_size, and total_sub_edges:** Begin by processing the edge_index, matrix_size, and total_sub_edges. These components are crucial for handling the edge mappings and matrix operations. Load the edge_index and total_sub_edges from their respective files. The matrix_size is determined by the length of the edge_index.

- **Loading the Matrix:** After initializing the necessary components, load the edge matrix (inter_edges_matrix) from the saved file. This matrix contains the precomputed data that will be used for analysis.

- Code:

```
# Must-have
edge_index = load_edge_index(index_file)
matrix_size = len(edge_index)
total_sub_edges = load_total_sub_edges(total_sub_edges_file)

inter_edges_matrix = load_inter_edges_matrix(matrix_file)
```

- **Verify Accuracy:** To ensure the accuracy of the program, take sample edges from edgeHistory, map these to their corresponding edges in HoChiMinh.osm, and check the relevant rows in the sparse matrix (inter_edges_matrix). This verification helps confirm that the sub-edges are correctly mapped to the real edges and that the matrix reflects these mappings accurately.

- Code:

```
edgeHistory = [["3342113667","6768412184"], ["5738080746","3220146382"], ["3915814322","5777195880"], ["5740180436","5755254579"],
               ["2240993963","9432516568"]]
edgeList = []

# Check for accuracy of the program, take some edge on the bus_history, map with its edge in HoChiMinh.osm, then check for
# the corresponding row in the sparse matrix (edge matrix)

print()

for edge in edgeHistory:
    corresponding_edge = total_sub_edges[tuple(edge)]
    print(f"History sub_edge: {edge} -> edge (HoChiMinh.osm) : {corresponding_edge} -> index : {edge_index[str(corresponding_edge)]}")
    edgeList.append(str(corresponding_edge))

res = get_rows_from_matrix(edgeList, inter_edges_matrix, edge_index)

print()

for data, inters in res.items():
    print(f"Row {data}:")
    print(inters)
    print()
```

- In this process:

1. We check and print the mapping of each edge from edgeHistory to its corresponding real edge and index.
2. We then retrieve and print the rows from the edge matrix that correspond to the edges in edgeList, ensuring that the matrix entries align with our expectations based on the bus history data.

→ This step is essential for validating that our preprocessing and matrix loading steps are correct and that the data is ready for further analysis.

6.2.2) Images of results

```
Load edge_index successfully
Load total_sub_edges successfully
Load edge_matrix successfully

History sub_edge: ['3342113667', '6768412184'] -> edge (HoChiMinh.osm) : 327427168 -> index : 15348
History sub_edge: ['5738080746', '3220146382'] -> edge (HoChiMinh.osm) : 1017702932 -> index : 97468
History sub_edge: ['3915814322', '5777195880'] -> edge (HoChiMinh.osm) : 192512367 -> index : 6091
History sub_edge: ['5740180436', '5755254579'] -> edge (HoChiMinh.osm) : 713568405 -> index : 74178
History sub_edge: ['2240993963', '9432516568'] -> edge (HoChiMinh.osm) : 1015714684 -> index : 97441

Row 15348:
{1288: 79995, 2006: 79995, 2037: 64769, 3281: 1932, 5420: 79995, 6986: 1932, 6987: 1932, 6989: 1932, 8328: 1932, 8329: 79995, 8335: 1932, 9212: 1932, 9872: 79995, 12322: 79995, 12323: 34946, 12326: 1932, 12332: 79995, 12602: 79995, 12744: 79995, 12745: 79995, 12746: 12745, 14262: 2006, 15148: 1932, 15341: 12745, 15822: 1932, 15831: 1932, 19974: 12323, 26799: 1932, 34946: 12745, 39669: 79995, 39670: 79995, 43732: 12323, 51193: 79995, 56351: 79995, 57826: 12745, 64769: 12746, 66743: 64769, 71380: 12745, 72896: 12323, 73259: 12745, 75977: 79995, 79347: 15349, 79537: 12746, 79830: 79995, 79920: 12745, 79921: 64769, 79949: 64769, 87676: 64769, 87680: 34946, 87683: 79995, 95774: 64769, 95775: 79995, 95776: 64769, 96922: 87680, 96923: 12323, 96924: 87680, 97029: 1932, 97270: 1932, 97274: 1932, 99779: 79995, 99909: 1932, 99911: 1932, 99916: 1932, 99917: 1932, 100811: 12745, 100812: 79995, 101114: 64769, 101115: 64769, 101116: 79995, 101118: 79995, 101120: 64769, 101226: 12745, 102391: 79995, 105185: 12745, 105186: 1932, 105187: 1932, 105644: 12745, 105645: 79995, 105706: 79995, 106254: 1932}

Row 97468:
{68: 1745, 260: 97469, 444: 1745, 500: 97469, 616: 97469, 1057: 1745, 1391: 18112, 1745: 97469, 1842: 1745, 2155: 99267, 4294: 1745, 4298: 99267, 6587: 18112, 6671: 97469, 6943: 97469, 6944: 18112, 6945: 97469, 6947: 18112, 6948: 97469, 6966: 18112, 6970: 97469, 6975: 97469, 8091: 97469, 8179: 99267, 8181: 99267, 8620: 99267, 8743: 8091, 8805: 18112, 10765: 97469, 10768: 97469, 10770: 97469, 11988: 18112, 11990: 97469, 11992: 18112, 12004: 18112, 12033: 99267, 14683: 1745, 14692: 1745, 14725: 18112, 14731: 97469, 14804: 18112, 14810: 1745, 14813: 18112, 15772: 97469, 15774: 18112, 15775: 18112, 15777: 18112, 15778: 97469, 15851: 97469, 16464: 99267, 16905: 97469, 16908: 97469, 17865: 99267, 17930: 1745, 17953: 18112, 18112: 97469, 26250: 1745, 26251: 97469, 26472: 39108, 26477: 18112, 26845: 97469, 26846: 97469, 28216: 18112, 28256: 18112, 31012: 1745, 33757: 97469, 35206: 97469, 37275: 97469, 38208: 18112, 39106: 18112, 39107: 97469, 39108: 18112, 39109: 18112, 41890: 97469, 44095: 18112, 46926: 1745, 58461: 3878, 58470: 1745, 72243: 97469, 73213: 97469, 73216: 18112, 73222: 18112, 73232: 18112, 74178: 97469, 81158: 18112, 81306: 99267, 81417: 97469, 81418: 18112, 81419: 18112, 82026: 97469, 87723: 18112, 87725: 18112, 96812: 18112, 97053: 1745, 97400: 18112, 97403: 18112, 97464: 1745, 97465: 1745, 97469: 18112, 97484: 18112, 97486: 18112, 97489: 97469, 97490: 18112, 97653: 1745, 97654: 1745, 99267: 1745, 104551: 39109, 105389: 1745, 105742: 18112, 105743: 18112, 106013: 18112}
```

```
Row 6091:
{358: 64628, 370: 81355, 490: 18095, 600: 81355, 608: 81355, 634: 81355, 657: 107285, 770: 16880, 1203: 16880, 1295: 16880, 1312: 15791, 1362: 81355, 1429: 81355, 1518: 18095, 1593: 81355, 1672: 27692, 1756: 16706, 1792: 81355, 1816: 9579, 1900: 57474, 1916: 81355, 1923: 81355, 1951: 81355, 2026: 57485, 2150: 1362, 3279: 15791, 3283: 101172, 3285: 6094, 3299: 18102, 3313: 71480, 3340: 81355, 3517: 6094, 3579: 101172, 3689: 81355, 3821: 81355, 3837: 81355, 3904: 16880, 3912: 101172, 3925: 15791, 3952: 15791, 4407: 27692, 4418: 27693, 4435: 16880, 4447: 16880, 4471: 81355, 4501: 81355, 5647: 16880, 6048: 101172, 6049: 15791, 6074: 15791, 6090: 15791, 6094: 15791, 6105: 15791, 6122: 15791, 8092: 57485, 8139: 81355, 8143: 1362, 8167: 81355, 8180: 101172, 8233: 81142, 8265: 18095, 8269: 16706, 8514: 1362, 8534: 16880, 8558: 101172, 8621: 16880, 8793: 8233, 8844: 81357, 9232: 64628, 9577: 16880, 9578: 16880, 9579: 81355, 9604: 81355, 9606: 101172, 9843: 16706, 9847: 15791, 9848: 15791, 9849: 15791, 9850: 15791, 9851: 15791, 10288: 16706, 10431: 81355, 10583: 1362, 12068: 81355, 12069: 81355, 12070: 81355, 13012: 81355, 13433: 15791, 14158: 81355, 14193: 16880, 14230: 81355, 15162: 81142, 15769: 107285, 15790: 32838, 15791: 101172, 15807: 15791, 15810: 15791, 16695: 69874, 16703: 81355, 16704: 81355, 16705: 81355, 16706: 81355, 16713: 81355, 16763: 64628, 16765: 1816, 16880: 101172, 17800: 15791, 17803: 15791, 17826: 81355, 17836: 81355, 17866: 16706, 17888: 81355, 17992: 15791, 18041: 81355, 18042: 81355, 18095: 64628, 18102: 1816, 18103: 81355, 18651: 81355, 18734: 16880, 18743: 81355, 19737: 15791, 19738: 15791, 19739: 15791, 26521: 27692, 26592: 81355, 26869: 81355, 26872: 16880, 27692: 16880, 27694: 16880, 27988: 15791, 27995: 77167, 27996: 81355, 27997: 16880, 28389: 27693, 28410: 16880, 28415: 16880, 28422: 64628, 28877: 16880, 29553: 16880, 29562: 16880, 29603: 12068, 29714: 16880, 29718: 15791, 29823: 15791, 29825: 15791, 29974: 16880, 29987: 16880, 30044: 16880, 32849: 81355, 32851: 16880, 32863: 12070, 33555: 81355, 33569: 16880, 34877: 81355, 38972: 81355, 40774: 16880, 45699: 81355, 45701: 29713, 45833: 81355, 45840: 101172, 50177: 64628, 50604: 81355, 52731: 81355, 53649: 81355, 53650: 101172, 53651: 27692, 53912: 16880, 54031: 16880, 56234: 101173, 56939: 101173, 56941: 101172, 56943: 81355, 56944: 81355, 56945: 101172, 56952: 101173, 56953: 81355, 56961: 81355, 56965: 81355, 56970: 27692, 56972: 81355, 56973: 81355, 56974: 81355, 56975: 16880, 56976: 81355, 56999: 81355, 57000: 16880, 57001: 16880, 57004: 57485, 57005: 16880, 57064: 101172, 57068: 16880, 57071: 16880, 57076: 81355, 57097: 15791, 57098: 81355, 57099: 81355, 57100: 16880, 57101: 81355, 57206: 9849, 57207: 15791, 57314: 15791, 57337: 15791, 57457: 101172, 57469: 15791, 57470: 15791, 57485: 29713, 57496: 16880, 57501: 16880, 57508: 81355, 57509: 81355, 57511: 16880, 57515: 81355, 57516: 81355, 57517: 81355, 59144: 81355, 59146: 101172, 59159: 16880, 59164: 81355, 59165: 16880, 59166: 81355, 59167: 81355, 59169: 81355, 59183: 101172, 59185: 81355, 60682: 81355, 60696: 16880, 61604: 81355, 61605: 16880, 61609: 16880, 61625: 81355, 61627: 81355, 61653: 16706, 61663: 18042, 61673: 16880, 61674: 16706, 61675: 16880, 61676: 81355, 64579: 16706, 64582: 81355, 64628: 81357, 64914: 81355, 64930: 81355, 64953: 81355, 64956: 81355, 64960: 16880, 64962: 1362, 64964: 16880, 65012: 81355, 65045: 77167, 65047: 81355, 65051: 81355, 65052: 79989, 65054: 81355, 65056: 101172, 65058: 16880, 65123: 16880, 65127: 16880, 65133: 81355, 65134: 16706, 65239: 81142, 65341: 101172, 65400: 79989, 68929: 9579, 69164: 64628, 69190: 1816, 69301: 1816, 69327: 18102, 69352: 16880, 69358: 27692, 69364: 81355, 69477: 32838, 69478: 15791, 69483: 71480, 69571: 81357, 69578: 64628, 69773: 15791, 69775: 15791, 69874: 81355, 71374: 81355, 71478: 18042, 71479: 81355, 71480: 81357, 72316: 81355, 72527: 16880, 73149: 81142, 73476: 101172, 75605: 81142, 75812: 81142, 76449: 81142, 76572: 101172, 76583: 101172, 77167: 81355, 77181: 16706, 77199: 16880, 77267: 81355, 77269: 81355, 79562: 8233, 79586: 15791, 79891: 15791, 79989: 16880, 80337: 16880, 80600: 16880, 80673: 15791, 81142: 64628, 81352: 1362, 81355: 16880, 81357: 1816, 81363: 64628, 81384: 15791, 81585: 15791, 87697: 71480, 93507: 16880, 95833: 1816, 95835: 100395, 95879: 12070, 96848: 15791, 96849: 15791, 96867: 15791, 96972: 15791, 96973: 15791, 96975: 15791, 97167: 71480, 100033: 16880, 100176: 29713, 100181: 16880, 100395: 81355, 100427: 101172, 101172: 57485, 101173: 101172, 101174: 101172, 101175: 32838, 101176: 101172, 101970: 1816, 103800: 100395, 103956: 16880, 104097: 81355, 104184: 16880, 104217: 81355, 105591: 15791, 105592: 15791, 105593: 15791, 105594: 9849, 105711: 81355, 105713: 81355, 105714: 81355, 105721: 101172, 106360: 64628, 106374: 81355, 106375: 81355, 106499: 81355, 107205: 81355, 107206: 81355, 107285: 64628, 107423: 16880, 107562: 101172}
```

```
Row 74178:
{1391: 97468, 6587: 18112, 6671: 18112, 6943: 97489, 6944: 18112, 6945: 18112, 9923: 97489, 10312: 18112, 11992: 18112, 12004: 97468, 14813: 18112, 15774: 18112, 15775: 18112, 15777: 97489, 15851: 18112, 17953: 18112, 18112: 97468, 26251: 97468, 26472: 97489, 28256: 97489, 35206: 97489, 37275: 97468, 39107: 97489, 39108: 97489, 44095: 97468, 73216: 18112, 73232: 97489, 81419: 97489, 97400: 18112, 97403: 18112, 97469: 97468, 97486: 37275, 97490: 18112, 105743: 18112}

Row 97441:
{204: 4706, 226: 19749, 233: 78461, 291: 19749, 293: 17208, 370: 9579, 495: 19748, 531: 19748, 615: 19749, 730: 19749, 1293: 19749, 1303: 19749, 1429: 19749, 1593: 9579, 1916: 9579, 1955: 17201, 2013: 17201, 2072: 19749, 2075: 21188, 2090: 10158, 2125: 78459, 2159: 21188, 3303: 8489, 3305: 9976, 3312: 9976, 4052: 9974, 4053: 9974, 4297: 53434, 4706: 19749, 4936: 21188, 5296: 9974, 6091: 19749, 8139: 53434, 8176: 53433, 8177: 78458, 8178: 53433, 8269: 8491, 8486: 8489, 8489: 19749, 8491: 8489, 8495: 53433, 8500: 78563, 8543: 53433, 8546: 53434, 8547: 53433, 8867: 53434, 9207: 78459, 9209: 10158, 9215: 19748, 9218: 19748, 9224: 19748, 9227: 96871, 9229: 19754, 9315: 19749, 9408: 78458, 9411: 4706, 9578: 19749, 9579: 6091, 9796: 9977, 9828: 15796, 9971: 9977, 9974: 19749, 9976: 81291, 9977: 9974, 9978: 8486, 9979: 78458, 9980: 9974, 9981: 19749, 10157: 53433, 10158: 8486, 10159: 53434, 10160: 53433, 10313: 77860, 12068: 9579, 12804: 19749, 12827: 19749, 12842: 19748, 13069: 78458, 13348: 9974, 13366: 19749, 13395: 9974, 13407: 19749, 13410: 9974, 13419: 81291, 13427: 19749, 13964: 81291, 14033: 19749, 14043: 19749, 14075: 9974, 14546: 81291, 14596: 19748, 14598: 19749, 14610: 9974, 14611: 9974, 14614: 19749, 14625: 19749, 14633: 9974, 14639: 78458, 14641: 9974, 14650: 9974, 14660: 9974, 14662: 81291, 15773: 78459, 15774: 53433, 15796: 78458, 16880: 6091, 17201: 19748, 17208: 19748, 17768: 19749, 17849: 9974, 17858: 78458, 17860: 78574, 17888: 53434, 17890: 53433, 17936: 19754, 17937: 19748, 18103: 9579, 19748: 19749, 19749: 78459, 19750: 19748, 19751: 17208, 19753: 19749, 19754: 19748, 21136: 81291, 21660: 19749, 24012: 78459, 26592: 9579, 26938: 78574, 26956: 78458, 26963: 21188, 27325: 78458, 27433: 53433, 27583: 53433, 28756: 21188, 30044: 78458, 31864: 81291, 31865: 9976, 36489: 9974, 37428: 53433, 40931: 81291, 41921: 19749, 41922: 9974, 42065: 19749, 47289: 19749, 47414: 9974, 47417: 78458, 47425: 81291, 47659: 9974, 48927: 78461, 49130: 9974, 50604: 19749, 52638: 19749, 53433: 78458, 53434: 78458, 53982: 19748, 54229: 9974, 55835: 9974, 55924: 19748, 56049: 19754, 56053: 19749, 56059: 17201, 56749: 19749, 56925: 19748, 56943: 9579, 58613: 78458, 71587: 78458, 71605: 78458, 71715: 19749, 71774: 9974, 71867: 19749, 71891: 19748, 72098: 19748, 73165: 19749, 73518: 21188, 77845: 19749, 77846: 81291, 77847: 9976, 77848: 9974, 77855: 9976, 77860: 81291, 77861: 81291, 77865: 81291, 77866: 81291, 77867: 81291, 77868: 81291, 77870: 81291, 77881: 9974, 78023: 78459, 78458: 19749, 78459: 19749, 78460: 78458, 78461: 19749, 78462: 19749, 78561: 8489, 78562: 8486, 78563: 8486, 78572: 8489, 78573: 8486, 78574: 8486, 78591: 78458, 78592: 8491, 79977: 53433, 80530: 78458, 80755: 9974, 80975: 9974, 81291: 9974, 81355: 6091, 87710: 19756, 87711: 17208, 87723: 78458, 87726: 78458, 91292: 19748, 95200: 9977, 95583: 19749, 96810: 53433, 96814: 78458, 96815: 78574, 96845: 19749, 96846: 19749, 96868: 78459, 96871: 19748, 96876: 9974, 96932: 78458, 96971: 9976, 96988: 3305, 96989: 9976, 96991: 78459, 97117: 9974, 97278: 17201, 97279: 19748, 97280: 293, 97282: 19754, 97400: 8489, 97402: 78459, 97490: 78458, 97491: 53433, 98219: 21188, 101172: 6091, 101223: 96871, 103126: 19749, 103489: 8491, 103491: 53434, 103493: 53434, 103494: 53433, 103495: 53433, 103633: 9974, 104367: 78458, 104919: 53433, 105046: 21188, 105224: 81291, 105245: 21188, 105713: 9579, 105714: 9579, 105951: 78592}
```

7) FURTHER PROCESSING

7.1) Preprocessing all edges in HoChiMinh.osm

- In this preprocessing step, the goal is to convert edge data from the historical bus trip records into a format that includes geographical coordinates, which are necessary for spatial analysis and visualization.

- We begin by extracting node coordinates from the OpenStreetMap (OSM) file, *'HoChiMinh.osm'*. Using XML parsing, we traverse the OSM file to gather latitude and longitude information for each node, storing these coordinates in a dictionary with the node ID as the key. This dictionary allows for quick lookup of node coordinates when processing the bus trip data.

- Next, we load the historical bus trip data from the file *'bus_history.json'*. This data includes a list of vehicles, each with multiple trips. For each trip, we extract the edges, represented as pairs of node IDs, from the *'edgesOfPath2'* field. These edges are then mapped to their corresponding geographical coordinates using the previously created node coordinates dictionary.

- The output of this process is a list of edge coordinates, where each edge is represented by the latitude and longitude of its start and end nodes. This set of edge coordinates is crucial for further analysis, such as constructing edge matrices or visualizing the bus network on a map.

- This preprocessing ensures that the edge data is in a spatial format, enabling more advanced geospatial computations and visualizations in subsequent steps of the project.

- Code:

*** Notice: For the VehicleQuery class, I have discussed above (module 3).**

```
osm_file_name = "HoChiMinh.osm"

def get_all_node_coordinates(fileName):
    tree = ET.parse(f"osmFiles/{fileName}")
    root = tree.getroot()

    node_coordinates = {}

    for node in root.findall('node'):
        node_id = node.get('id')
        lat = float(node.get('lat'))
        lon = float(node.get('lon'))
        node_coordinates[node_id] = (lat, lon)

    return node_coordinates

busNetwork = VehicleQuery("bus_history.json")
vehicle_history = busNetwork.vehicleList
node_coordinates = get_all_node_coordinates(osm_file_name)
```

- Code transition to spatial data:

```
total_edges_nodes = []
for vehicle in vehicle_history:
    for trip in vehicle.getTripList:
        total_edges_nodes.append(trip["edgesOfPath2"])

total_edges_coordinates = []
for trip in total_edges_nodes:
    new_path = []
    for edge in trip:
        lat1, lon1 = node_coordinates[edge[0]]
        lat2, lon2 = node_coordinates[edge[1]]
        new_path.append([[lon1, lat1],[lon2, lat2]])
    total_edges_coordinates.append(new_path)
```

- Purpose of this process:

→ The goal of the preprocessing step is to transform raw edge data into a format that is more suitable for optimization tasks. Specifically, by mapping node IDs to their corresponding coordinates and converting these into edge coordinates, the process prepares the data for advanced analysis. This setup allows for efficient querying and manipulation of edges based on their geographic locations, which is crucial for optimizing tasks such as route planning, traffic analysis, or any other geospatial computation that relies on accurate and fast edge operations.

7.2) Visualization HoChiMinh.osm

- Visualizing the map data from *'HoChiMinh.osm'* plays a crucial role in understanding the spatial layout and structure of the city's transportation network. By rendering the map with a subset of nodes and edges, the visualization provides an overview of the urban environment and allows for better insights into how different parts of the city are connected.

- This visualization is not just a visual aid; it serves as a foundational step in verifying the integrity of the data extracted from the OSM file. By plotting nodes and edges, it becomes easier to identify potential discrepancies or gaps in the data that could affect subsequent analysis. Moreover, this step aids in comprehending the geographical distribution of the transportation network.

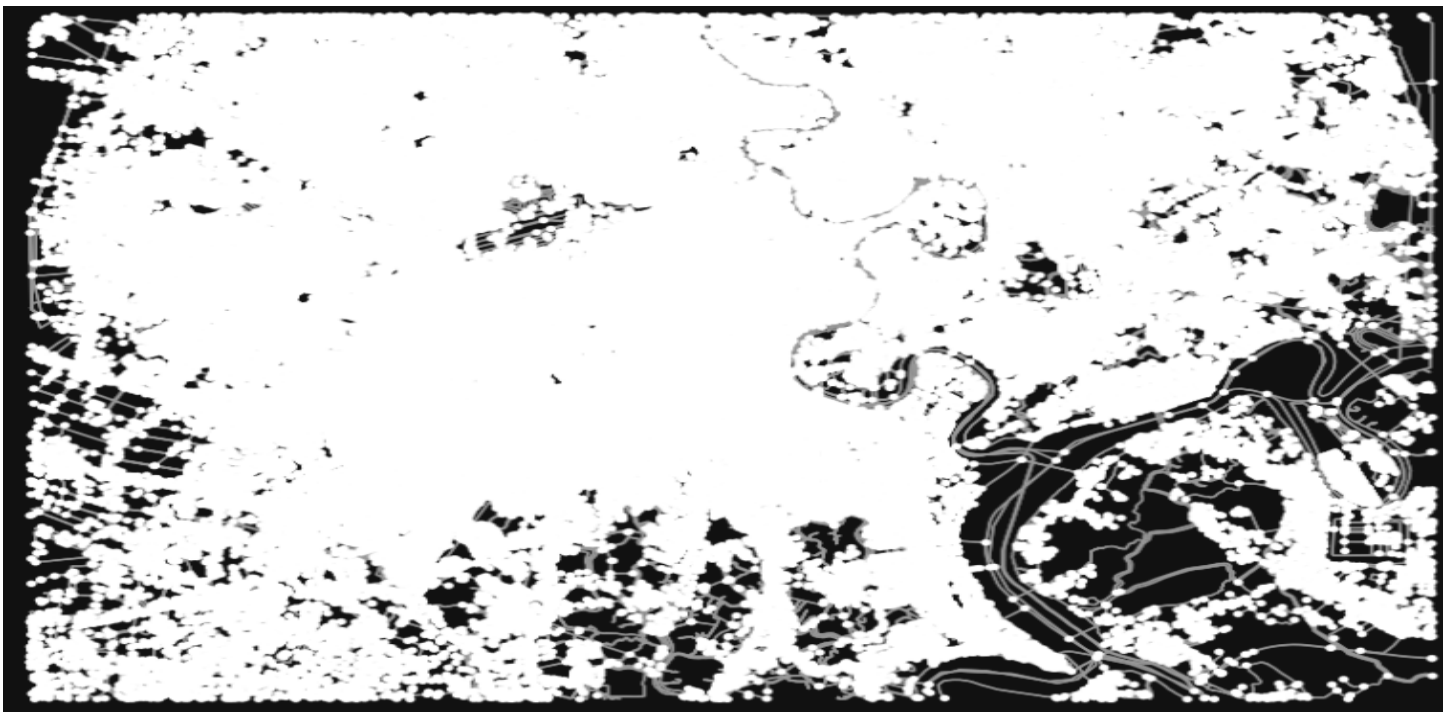
- The use of tools like **Folium** for this visualization further enhances the interactivity, enabling dynamic exploration of the map, which can reveal patterns or anomalies in the data. This step sets the stage for more sophisticated analyses and ensures that the preprocessing of the OSM data aligns with the geographical realities of Ho Chi Minh City.

- Code:

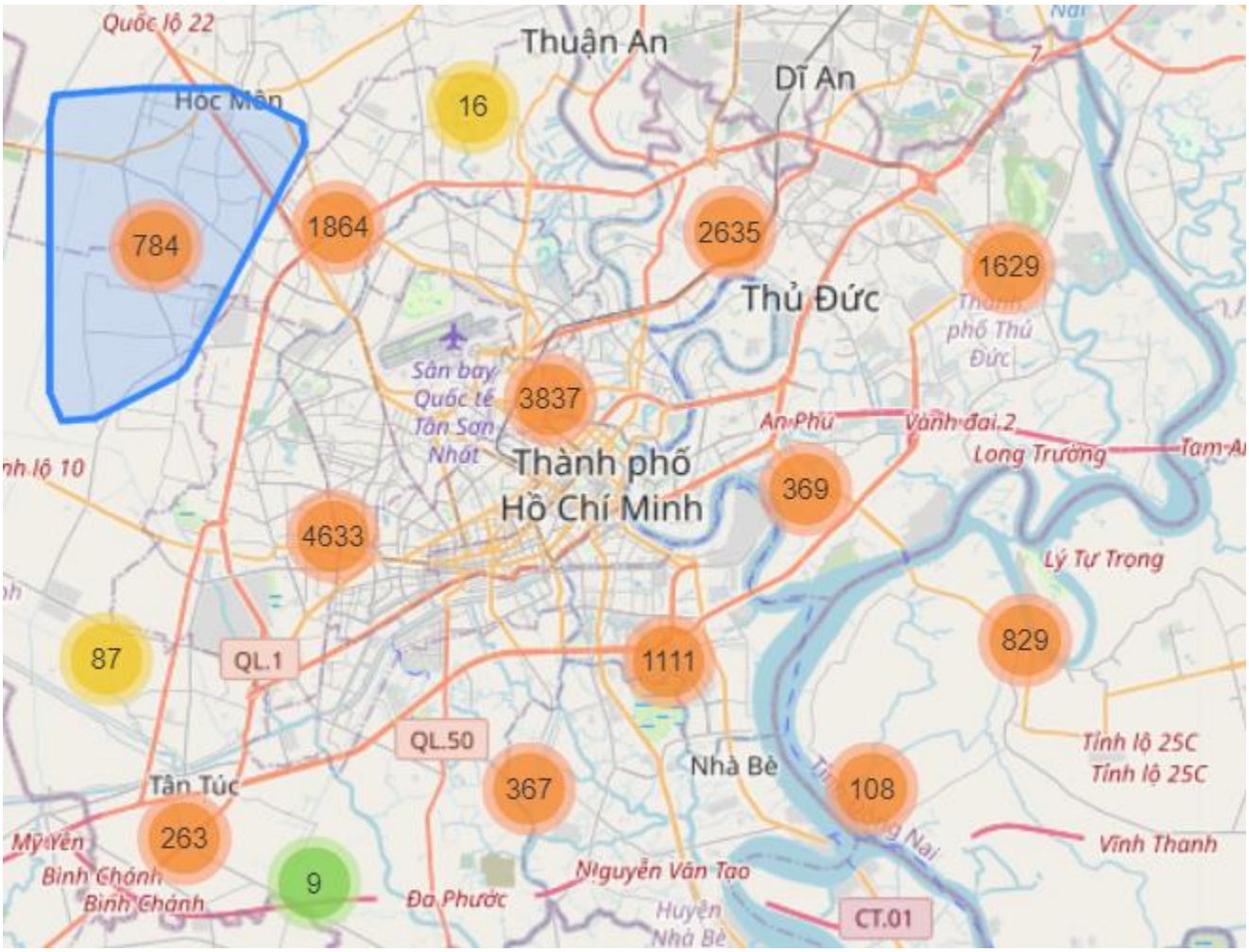
```
G = ox.graph_from_xml('osmFiles/HoChiMinh.osm')

nodes, edges = ox.graph_to_gdfs(G)

sampled_nodes = nodes.sample(frac=0.1)
sampled_edges = edges.sample(frac=0.1)

m = folium.Map(location=[10.8231, 106.6297], zoom_start=11)

mc = MarkerCluster()
for idx, node in sampled_nodes.iterrows():      You, last week • Building Graph (.osm-based) and Edge Matrix (.j…
    folium.CircleMarker([node['y'], node['x']], radius=2).add_to(mc)
mc.add_to(m)

for idx, edge in sampled_edges.iterrows():
    folium.PolyLine(edge['geometry'].coords, weight=2, color='red').add_to(m)

m.save('hcmc_map.html')
```

7.3) Figures and visualization

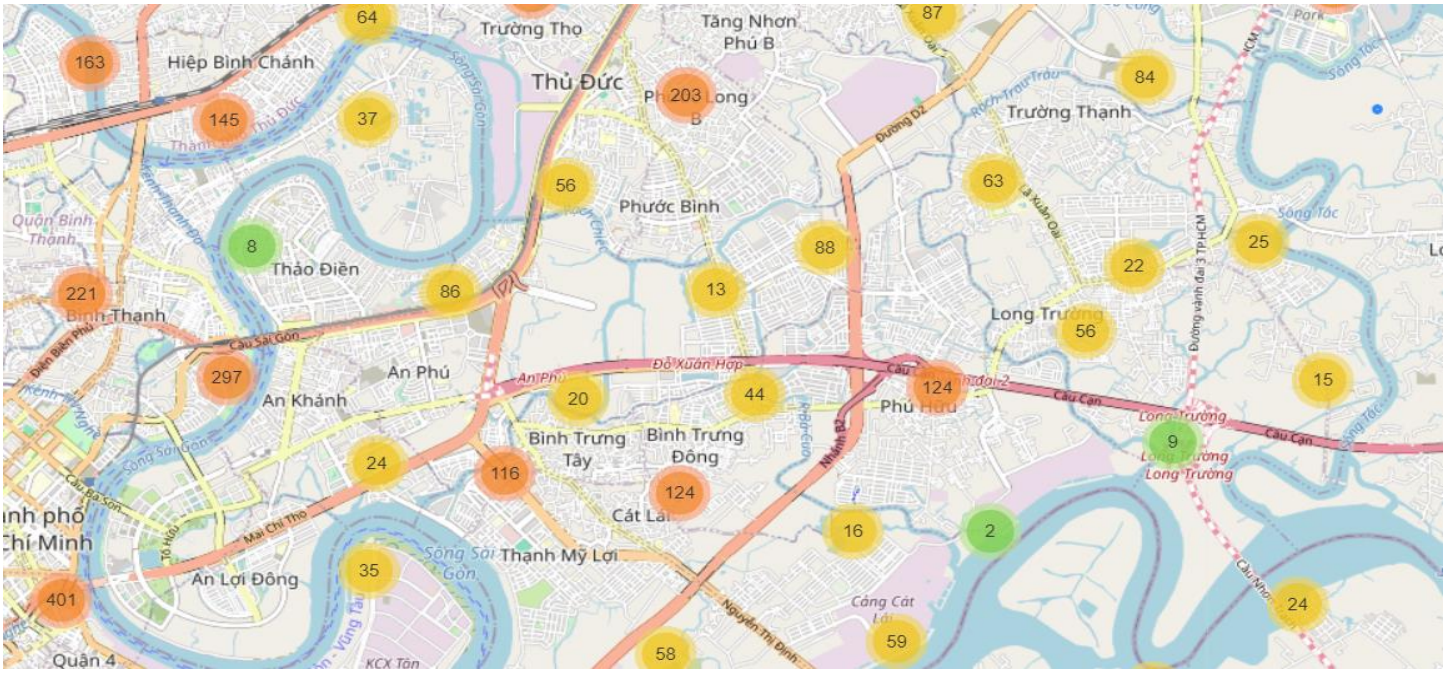
- Figures (nodes in HoChiMinh.osm):

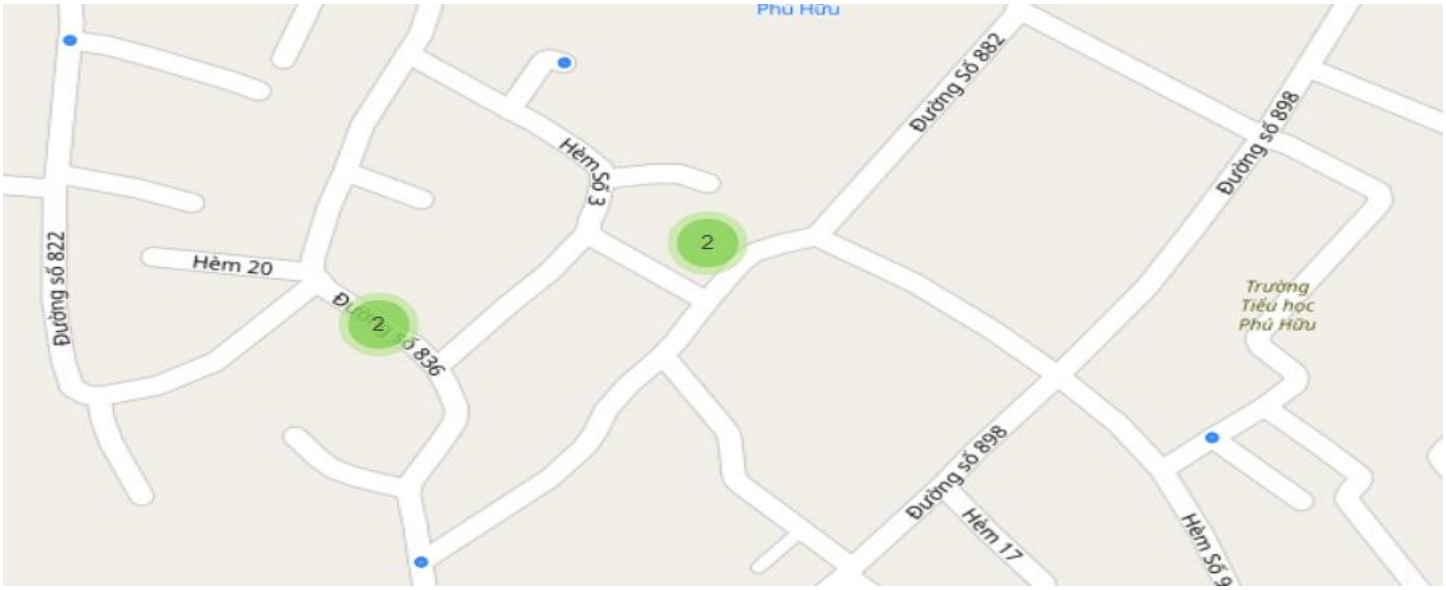


- Visualization of JSON file:



- More details:





8) CONCLUSION

8.1) Summary

- In this technical report, I addressed the challenge of estimating bus positions during data disruptions in Ho Chi Minh City's public transportation system. By leveraging historical data and advanced data processing techniques, I developed a robust framework capable of maintaining accurate bus tracking even in the face of intermittent GPS signals, network connectivity issues, or sensor failures.

8.2) Achievements

- Map Data Organization:

+ I successfully processed and organized the OpenStreetMap (OSM) data of Ho Chi Minh City into a graph structure. This structure, which represents the city’s locations as nodes and connections between these locations as edges, provided the foundation for further analysis and bus tracking.

- Mapping / Hashing Edges:

+ I successfully implemented a robust method for mapping and hashing edges from the bus history data to the actual edges in the OSM graph. This involved creating a comprehensive dictionary that maps sub-edges from bus trips to their corresponding real edges in the graph. By leveraging this mapping, I ensured an accurate representation of the bus routes within the edge matrix.

+ The hashing process was crucial for efficient lookups and integration. I developed a hashing mechanism that maps sub-edges to their real edges using a dictionary-based approach, which significantly improved performance and accuracy when processing large datasets.

- Historical Data Utilization:

+ The project made effective use of historical bus trip data to identify patterns and construct an edge matrix. This matrix captures the frequency of occurrences between edges, serving as a critical reference for estimating bus positions during data disruptions.

- Edge Matrix Development:

+ I developed an edge matrix that not only helps in predicting the most probable path a bus would take during data disruptions but also facilitates quick data retrieval through the use of optimized data structures. The matrix was constructed using a combination of techniques including sliding windows, prefix sum updating, and parallel processing, ensuring both efficiency and accuracy.

- Visualization of Bus Trips:

+ The system includes a capability for visualizing bus trips on a map of Ho Chi Minh City. This visualization integrates real-time and historical data, allowing for a comprehensive view of bus routes and aiding in the identification of potential inefficiencies or disruptions.

**** Impact and Future Work:**

- The implementation of this system is expected to significantly enhance the reliability and efficiency of the city's public transportation network. By providing accurate bus position estimates during disruptions, the system helps mitigate operational inefficiencies, reduces passenger dissatisfaction, and supports more informed decision-making in route management.

- Looking forward, further research and development can focus on refining the predictive models used in the edge matrix by incorporating additional factors such as traffic conditions, weather, and time-of-day variations. Additionally, expanding the system to cover other forms of public transportation and integrating it with broader smart city initiatives could provide even greater benefits to urban mobility in Ho Chi Minh City and beyond.

9) REFERENCES

9.1) NetworkX Documentation.

NetworkX: Network Analysis in Python. Available at:

Software for Complex Networks — NetworkX 3.3 documentation (Accessed: 17 August 2024).

- This reference provides detailed documentation on the NetworkX library, which was used in this project for creating and analyzing graph structures. It covers various functionalities, including the creation, manipulation, and study of complex networks.

9.2) OpenStreetMap Wiki.

Elements - OpenStreetMap Wiki. Available at:

Elements - OpenStreetMap Wiki (Accessed: 17 August 2024).

- The OpenStreetMap (OSM) Wiki was consulted to understand the structure and components of OSM data, including nodes, ways, and relations. This information was crucial for processing and handling the OSM data used in this project.

9.3) Osmium Documentation.

Osmium: Fast and Flexible OpenStreetMap Data Processing. Available at:

Osmium Library - osmcode (Accessed: 17 August 2024).

- This reference provides comprehensive documentation on the Osmium library, which was used for efficient handling and processing of OSM data. The library's performance and scalability were key to managing the large datasets involved in the project.

9.4) Python's Multiprocessing Library.

Multiprocessing — Process-based Parallelism. Available at:
https://docs.python.org/3/library/multiprocessing.html (Accessed: 17 August 2024).

- The Python Multiprocessing library documentation was used as a guide for implementing parallel processing in this project. It provided valuable insights into how to distribute tasks across multiple CPU cores, which helped improve the efficiency of data processing.

9.5) Scipy Sparse Matrix Documentation.

Sparse Matrices (scipy.sparse) — SciPy v1.10.0 Manual. Available at:

Sparse matrices (scipy.sparse) — SciPy v1.14.0 Manual (Accessed: 17 August 2024).

- This reference offers detailed information on the use of sparse matrices in Python. It was particularly useful for implementing the efficient storage of the edge frequency matrix by saving only non-zero elements.

9.6) Geospatial Data Handling with Python.

Geospatial Data Handling using Python by GeoPandas and Shapely. Available at:

https://geopandas.org/ (Accessed: 17 August 2024).

- GeoPandas and Shapely were referenced for their capabilities in handling geospatial data in Python, which provided supplementary methods for managing and visualizing map data.