08/2024

BIG GRAPH PROCESSING

Data Structures - CS163

Task 03: Estimate buses' position

Prepared By
Le Tien Dat

Student ID
23125028

Class
23APCS2

www.facebook.com/ltd.sword
ltdat23@apcs.fitus.edu.vn
University of Science, VNU-HCM

Contents

Cover Page 1
Contents 3
1. Problem Statement and File Description 4
1.1. Problem Statement 4
1.2. Descirption of two input files 4
1.2.1. HoChiMinh.osm 4
1.2.2. bus-history.json 6
2. Organize the .osm file to graph structure 7
2.1. What is .osm file? 7
2.2. Read HoChiMinh.osm file 7
2.3. Structurize the data 8
3. Edge Matrix creation 10
3.1. Load the edges 10
3.2. Map the pair of nodes in bus-history.json 10
3.3. Processing the most occuring edges 12
3.4. Result 15
3.5. Query and Testing 16
4. Conclusion 18
5. Reference 19

1. Problem Statement and File Description

1.1. Problem Statement

The data includes:
- City data are stored in the file HoChiMinh.osm in OpenStreetMap format (https://osmcode.org/file-formats-manual/#file-formats)
- Historical bus trip data are stored in the file bus-history.json, which includes a list of buses, each with a list of trips consisting of the sequence of edges the bus passed through (each edge consists of 2 point IDs - taken from the map data).

The requirements are as follows:
- Read the Ho Chi Minh City map data and organize it into a graph structure.
- Create an edge matrix to identify the most frequently occurring edge between any two edges based on historical data. Save the matrix to a file such that, given a list of edges, the corresponding rows of the matrix can be retrieved.

Based on the map data and the first requirement, propose a way to store the edge matrix as efficiently as possible and an algorithm to find the most frequently occurring edge between any two given edges.

1.2. Descirption of two input files

1.2.1. HoChiMinh.osm

This is an *OpenStreetMap* file which store the preliminary map of Ho Chi Minh City, which consists of nodes, ways and relations.

The structure of the file as the picture follows:

InputFiles > HoChiMinh.osm

```
<?xml version='1.0' encoding='UTF-8'?>
<osm version="0.6" generator="osmconvert 0.8.11" timestamp="2023-09-01T00:00:00Z">
    <bounds minlat="10.66" minlon="106.54" maxlat="10.89" maxlon="106.85"/>
    <node id="366224284" lat="10.6600871" lon="106.7792241" version="1"/>
    <node id="366224291" lat="10.6628748" lon="106.7773527" version="1"/>
    <node id="366367223" lat="10.8042433" lon="106.6290559" version="1"/>
    <node id="366367233" lat="10.7711099" lon="106.7097014" version="1"/>
    <node id="366367242" lat="10.7093371" lon="106.7371895" version="1"/>
    <node id="366367274" lat="10.854489" lon="106.7600808" version="1"/>
    <node id="366367285" lat="10.8049939" lon="106.7211626" version="1"/>
    <node id="366367308" lat="10.8891263" lon="106.7664372" version="1"/>
    <node id="366367319" lat="10.790438" lon="106.6185524" version="1"/>
    <node id="366367322" lat="10.7991487" lon="106.6571439" version="1"/>
    <node id="366367330" lat="10.8469507" lon="106.564943" version="1"/>
    <node id="366367338" lat="10.7237534" lon="106.6325754" version="1"/>
    <node id="366367340" lat="10.8031111" lon="106.7142612" version="1"/>
    <node id="366367342" lat="10.7647028" lon="106.7062937" version="1"/>
    <node id="366367348" lat="10.6678407" lon="106.5889887" version="1"/>
    <node id="366367364" lat="10.7868079" lon="106.5523388" version="1"/>
    <node id="366367367" lat="10.8138085" lon="106.732437" version="1"/>
    <node id="366367369" lat="10.8597279" lon="106.6378377" version="1"/>
```

A *node* includes id, version and the coordinate (latitude and longtitude denotes as *lat* and *lon*). Sometime it can have some *tags,* which is the further information about the node. (highway, gate, barrier, …)

```
2498610     <tag k="service" v="alley"/>
2498611   </way>
2498612   <way id="802338945" version="1">
2498613     <nd ref="7505048076"/>
2498614     <nd ref="366408509"/>
2498615     <nd ref="366479294"/>
2498616     <nd ref="366449145"/>
2498617     <nd ref="2987193110"/>
2498618     <nd ref="5755653748"/>
2498619     <nd ref="2024972876"/>
2498620     <tag k="highway" v="secondary"/>
2498621     <tag k="lanes" v="2"/>
2498622     <tag k="name" v="Đường Võ Văn Ngân"/>
2498623     <tag k="name:en" v="Vo Van Ngan Street"/>
2498624     <tag k="name:zh" v="武文银路"/>
2498625     <tag k="surface" v="asphalt"/>
2498626   </way>
2498627   <way id="802766869" version="1">
2498628     <nd ref="4976692690"/>
```

A *way* contains *id, version,* some *tags* (if exists) and a list of *nodes* (denoted as *nd* and *ref* – the id of the node). These nodes are connected to each other by this way. Therefore, we can create a graph structure based on the ways in this file by listing the connections between nodes.

```
3219882   <relation id="2531662" version="1">
3219883     <member type="way" ref="39514796" role="outer"/>
3219884     <member type="way" ref="165491096" role="inner"/>
3219885     <member type="way" ref="165491088" role="inner"/>
3219886     <tag k="addr:housename" v="Tòa án Nhân dân TP.HCM"/>
3219887     <tag k="addr:housenumber" v="131"/>
3219888     <tag k="addr:street" v="Nam Kỳ Khởi Nghĩa"/>
3219889     <tag k="amenity" v="courthouse"/>
3219890     <tag k="building" v="yes"/>
3219891     <tag k="int_name" v="Ho Chi Minh City People&#39;s Court"/>
3219892     <tag k="name" v="Tòa án Nhân dân TP.HCM"/>
3219893     <tag k="name:en" v="Ho Chi Minh City People&#39;s Court"/>
3219894     <tag k="name:id" v="Pengadilan Rakyat Kota Ho Chi Minh"/>
3219895     <tag k="name:vi" v="Tòa án Nhân dân TP.HCM"/>
3219896     <tag k="name:zh" v="胡志明市人民法庭"/>
3219897     <tag k="old_name" v="Saigon Courthouse"/>
3219898     <tag k="type" v="multipolygon"/>
3219899   </relation>
```

A *relation* involves *id, version*, some *tags* (if exist) and a list of *member.* The members in the relation can be nodes, ways, or also *other relations* that is the child of this relation. To be simple, the relation is something similar to an area having the names in tags and the nodes, ways, and subareas (child relations).

A *member* has some typical components: *type* (the type of member, can be nodes, ways or other relations), *ref* (the id of the member), and *role* (the role of the member in that relation)

1.2.2. bus-history.json

This file demonstrates the historical travelling path of some route in Ho Chi Minh City.

The structure of the file is as the below picture:

Each line in this file contains the vehicle number, route ID, route var ID, and a trip list. Inside of a trip list is a list of path (which is again a list of edges created by combining two nodes) and a timestamp of this path.

By using these path list (noting that each path is *independent* to other), we can output a edge matrix which store the **most frequently occur** edge between two edges.

And be mentioned that an edge is the connection of two head nodes in a highway, which is displayed in `HoChiMinh.osm` file.

2. Organize the .osm file to graph structure

2.1. What is .osm file?

OpenStreetMap (OSM) is a huge collection of volunteered geographic information stores in different types of files, using different encoding schemes to convert this data into bits and bytes. OSM is a collaborative effort toward the creation of a free editable map of the world. The primary output of this collaborative effort is geographic data rather than the map itself. The constraints on the use or availability of geographic information across much of the world triggers the need to create an OSM.

The data available from OSM is ready to replace Google Maps for classical applications (Facebook, Craigslist etc.) and default data for GPS receiver's applications.Although data quality is diverse across the world yet OpenStreetMap data can be conveniently compared with patent data sources.

An OSM file is a street map saved in the OpenStreetMap (OSM) format. It contains **XML-formatted** data in the form of "nodes" (points), "ways" (connections), and "relations" (street and object properties, such as tags).

2.2. Read HoChiMinh.osm file

To read an OSM file in Python, we need to install osmium from `pip` – a library for processing OSM files.

To get the osmium, open the terminal and type in:

```
pip install osmium
```

And wait for the pip to install the library.

Once we get the osmium, we can implement the Python class `HCMGraph` to get the nodes, ways and relations as below code:

```
class HCMGraph(osm.SimpleHandler):
    def __init__(self):
        osm.SimpleHandler.__init__(self)
        self.nodes = {}
        self.ways = {}
        self.relations = {}
        self.intersections = []
        self.adjNode = defaultdict(dict)

    def node(self, n):
        # initialize the dictionary with the id of the node
        self.nodes[n.id] = {
            'location': (n.location.lat, n.location.lon),
```

```
            'tags': dict(n.tags)
        }

    def way(self, w):
        self.ways[w.id] = {
            'nodes': [str(n.ref) for n in w.nodes],
            'tags': dict(w.tags)
        }

    def relation(self, r):
        self.relations[r.id] = {
            # ref = id of type of the member
            # role = role of the member in the relation (city, subarea, etc.)
            # type = type of the member (relation...)
            'members': [(str(m.ref), str(m.role), str(m.type)) for m in r.members],
            'tags': dict(r.tags)
        }
```

Then, we can simply run this code and we will get the data for the file:

```
handler = HCMGraph()
handler.apply_file("InputFiles/HoChiMinh.osm")
```

2.3. Structurize the data

Now we'd got the data for structurizing, we will construct the graph based on the nodes and ways.

Noting that an edge is the connection between two head of a highway in this file, so we will create an adjacency list consisting of a list of nodes that create an edge with the node we are considering.

Details of the implementation:

```
def structurize(self):

        # structurize the data as graph
        # the adjacency list of the nodes comes from the ways
        # adjNode stucture = {node_id: {
        #   node_id1: way1_id,
        #   node_id2: way2_id,
        #   ...
        # }}
        #-----------------------------------------------------------

        for way_id, way in self.ways.items():
            tag = way['tags']
            if (tag.get('highway') != None):
                self.adjNode[way['nodes'][0]][way['nodes'][-1]] = {'id': way_id, 'nodes':
way['nodes'][1:-1]}
```

```
                if (tag.get('oneway') != 'yes'):
                    self.adjNode[way['nodes'][-1]][way['nodes'][0]] = {'id': way_id,
'nodes': way['nodes'][-2:0]}
        print(len(self.adjNode))
        length = 0
        for key in self.adjNode:
            length += len(self.adjNode[key])
        print(length)
```

After creating the graph, we should save it, along with the datails of the nodes, ways and relations to the file so that we can load it from file without computing it again when processing with `bus-history.json` file.

```
    def outputAsJSON(self):
        #output to 3 files in UTF-8
        with open('Result/nodes.json', 'w', encoding= 'utf8') as f:
            json.dump(self.nodes, f, ensure_ascii=False)
        with open('Result/ways.json', 'w', encoding='utf8') as f:
            json.dump(self.ways, f, ensure_ascii=False)
        with open('Result/relations.json', 'w', encoding = 'utf8') as f:
            json.dump(self.relations, f, ensure_ascii=False)
        with open('Result/adjNode.json', 'w', encoding = 'utf8') as f:
            json.dump(self.adjNode, f, ensure_ascii=False)
```

3. Edge Matrix creation

3.1. Load the edges

Firstly, we have to load the pair of nodes which are the two heads of the ways in `HoChiMinh.osm`. For easiness in the mapping section, I introduced a dictionary called *interDict* which stores the ways that a node is on along with the index of this node on the ways.

For example,

`interDict['373543511'] = [['5758104203','373543511',2], ['373543511','378247304',0]]`

which means the node that has an id of '373543511' is on the way `['5758104203','373543511']` with index 2 and way `['373543511','378247304']` with index 0.

Implementation of interDict:

```
def loadWay(self, filename):
        self.interDict = defaultdict(list)
        for wayid in self.ways:
            way = self.ways[wayid]
            tag = way['tags']
            nodes = way['nodes']
            if (tag.get('highway') != None):
                self.listEdges.append([nodes[0], nodes[-1]])
                if (tag.get('oneway') == None or tag['oneway'] == 'no'):
                    self.listEdges.append([nodes[-1], nodes[0]])
                for i in range(0, len(nodes)):
                    self.interDict[nodes[i]].append([nodes[0], nodes[-1], i])

        # map the listEdge for easier access
        self.listEdges = list(map(tuple, self.listEdges))
        self.totalEdges = self.listEdges
        print(f"Length of listEdges: {len(self.listEdges)}")
        self.mapEdges = {}
        for i in range(len(self.listEdges)):
            self.mapEdges[self.listEdges[i]] = i
        print(f"Length of mapEdges: {len(self.mapEdges)}")
```

Beside with `self.interDict`, I also introduce a new dictionary called `self.mapEdges`. This dictionary helps me to shorten the id of the node so that the following operations will be easier to define and better in performance.

3.2. Map the pair of nodes in bus-history.json

The next thing we should do is to map each pair of nodes in bus-history.json file to the corresponding edges, which we have defined above.

This is how the mapping process is conducted:

For example, we consider a pair of node having id: `["5755653748","2987193110"]`

We will look for the way that contains this two nodes in HoChiMinh.osm:

```
<way id="802338945" version="1">
        <nd ref="7505048076"/>
        <nd ref="366408509"/>
        <nd ref="366479294"/>
        <nd ref="366449145"/>
        <nd ref="2987193110"/>
        <nd ref="5755653748"/>
        <nd ref="2024972876"/>
        <tag k="highway" v="secondary"/>
        <tag k="lanes" v="2"/>
        <tag k="name" v="Đường Võ Văn Ngân"/>
        <tag k="name:en" v="Vo Van Ngan Street"/>
        <tag k="name:zh" v="武文银路"/>
        <tag k="surface" v="asphalt"/>
    </way>
```

Eventually, we find that this pair appears in a way having id = '802338945', and the edge corresponds to this way is ['7505048076', '2024972876'] or ['2024972876', '7505048076'] depending on the direction on the pair of nodes. In this instance, the direction of this pair is from '2024972876' to '7505048076', so the mapping edge we are finding is ['2024972876', '7505048076']

So, how can we express this way into our code?

This is when our `interDict` takes into play. It consists of the edge that a node is on and also the index of the node at this edge, so it will be easy for us to trace the edge from a pair of nodes.

The implemantation is just straight-forward: we will take the common edge of the two nodes and consider the index of them. If the index of the first node is smaller than that of the second node, we can just return the edge we found; otherwise, we reverse the edge before returning it.

```
def findEdge(self, u):

        # find the common intersection of the edge
        # u is a tuple of 2 node (a, b)
        a = self.interDict[u[0]]
        b = self.interDict[u[1]]
        for i in range(len(a)):
            for j in range(len(b)):
                if (a[i][0] == b[j][0] and a[i][1] == b[j][1]):
                    if (a[i][2] < b[j][2]):
                        if (self.mapEdges.get((a[i][0], a[i][1])) != None):
                            return (a[i][0], a[i][1])
                    else:
                        if (self.mapEdges.get((a[i][1], a[i][0])) != None):
                            return (a[i][1], a[i][0])
```

Then, we will utilize this function for all pairs of nodes in the file:

```
        self.edges = []
        self.listEdges = set()
        for trip in self.tripArr:
            temp = []
            for i in range (len(trip)):
                inter = self.mapEdges[self.findWay(trip[i])]
                self.listEdges.add(inter)
                if (len(temp) != 0):
                    if (temp[-1] != inter):
                        temp.append(inter)
                else:
                    temp.append(inter)
            self.edges.append(temp)
        self.listEdges = list(self.listEdges)
        self.listEdges.sort()
        print(f"Length of listEdges: {len(self.listEdges)}")
```

In my code, `self.tripArr` is the list containing all the `edgeOfPath2` listed in the JSON file. This mapping can create many duplicating edges in the list, so I am trying to remove the repeated edges and save the result into an other list: `self.edges`.

The result when printing the length of `self.listEdges`:

```
Length of listEdges: 17443
```

3.3. Processing the most occuring edges

After mapping the pairs in the `bus-history.json` file to edges, we can start the main process of this task: compute the most frequently occurring edge between two edges.

My first approach to this problem is to loop all the edges in each trip list, and with all pair of edges in the list, I will increase the freqency of all edges appearing between this pair and update the maximum frequency in each imcrement. In the end, I will loop all the pair of edges to take the result and store it in a final dictionary, called `self.matrix`.

This approach takes the time complexity as: $O(K.M^3)$, where K is the number of lists in the file and M is the average length of each list in the file. As $K = 15455$ and $M_{avg} = 70$, this approach will take approximately 3000 seconds in my estimation, which is a bit long. Moreover, the memory it takes is also a huge problem. The estimated auxiliary space of this *brute-force-like* algorithm is $O(N^2.M)$ (N is the number of edges in the file after mapping), which is more than 70GB of memory. Apparently, this approach cannot be processed on my 8GB-RAM laptop, which is too small compared to that expected space needed to store that data.

Therefore, I eventually came up with an alternative way to calculate the most frequently occurring edges, which allows me to process the function with lower memory. First, I will introduce a new dictionary called `self.tripDict`, which contains the trips that go through the edge we consider.

Implementation of the `tripDict` dictionary:

```
        self.tripDict = defaultdict(dict)
        # tripDict processing
        # tripDict structure: tripDict[node][path] = index
        for i in range(len(self.edges)):
            for j in range(len(self.edges[i])):
                self.tripDict[self.edges[i][j]][i] = j

        # process...
        self.process()
        print(f"Done processing in {times.time() - cur} seconds")
```

I also map each path to an index that is the index of the node in that path for easier to access and trace in the `process()` function.

This is how the function `process()` works:

- Loop through all pairs of edges.
- In each pair, we will pick out the common paths that have both edges, then trace for the index thanks to `self.tripDict`.
- Loop for all the common paths and increase the frequency. Record the frequency after each increment in a dictionary name `freq`.
- After loop for all the common paths, we calculate for the most frequently occurring edge between this two edges and store it to `self.matrix`.

The complexity in time for this approach is $O(N^2.C.M)$, where C is the average common paths for every two edges, M is the average length of each list in the file and N is the number of *distinct* edges in the file. Through a few calculations, it seems that $C_{avg} = 10$; along with the already-known values $M_{avg} = 70$ and $N = 17443$, I can estimate the running time as 4000 seconds. However, the time complexity in real-life performance is much faster than that expected figure since the loop in the common path can be much less than 70.

But the significance of this algorithm is the auxiliary space complexity. In approximate calculations, the space complexity is $O(N^2)$, which is less than 4GB in memory because we do not need to store the frequency of each edge in the overall matrix since we have a temporary dictionary instead. Moreover, each operations in considering a pair of edge is *an independent query* that we can apply in case of the time complexity goes too high, and we can divide the range to record the value of the matrix.

In addition, we can apply to the algorithm an optimization to reduce the time complexity. In an independent query of two edges, we just record the most frequently occurring edge that appears on the path from the first edge to the second edge. Now, we can use the resource that we have to perform the same loop – but with the instance that we consider the path from the second edge to the first edge.

For example, we take two edges named i and j into instance. When we are performing the query from i to j, we can also perform the query from j to i since the common paths in two queries are the same.

Below is the code for `process()` function. This funtion will loop for all edges and compute for the most frequently occuring edges and output the matrix to the JSON formatted file. You can see more details in my code in my GitHub link at [6].

```
def process(self):
        n = len(self.listEdges)
        for li in range(n):
            sub = times.time()
            for lj in range(li+1, n):
                i = self.listEdges[li]
                j = self.listEdges[lj]
                a = -1
                b = -1
                freq = defaultdict(lambda: 0)
                freq2 = defaultdict(lambda: 0)
                for key in self.tripDict[j]:
                    if (self.tripDict[i].get(key) != None):
                        a = self.tripDict[i][key]
                        b = self.tripDict[j][key]
                        if (a < b):
                            for k in range(a+1, b):
                                freq[self.edges[key][k]] += 1
                        else:
                            for k in range(b+1, a):
                                freq2[self.edges[key][k]] += 1
                # compute the max frequency
                if (a != -1 and b != -1):
                    maxFreq = [-1, 0]
                    for key in freq:
                        if (freq[key] > maxFreq[1]):
                            maxFreq = [key, freq[key]]
                    if (maxFreq[1] != 0):
                        self.matrix[i][j] = maxFreq
                    maxFreq = [-1, 0]
                    for key in freq2:
                        if (freq2[key] > maxFreq[1]):
                            maxFreq = [key, freq2[key]]
                    if (maxFreq[1] != 0):
                        self.matrix[j][i] = maxFreq
            print(f"Done with edge {i} ({li}) in {times.time() - sub} seconds")
        self.outputMatrix()
```

We can also save the matrix to a file for easier to output a corresponding row and load it again for later use.

```
    def outputMatrix(self):
        mapEdges = {}
        for i in self.mapEdges:
            mapEdges[self.combine(i)] = self.mapEdges[i]
        dict = {
            "mapEdges": mapEdges,
            "listEdges": self.listEdges,
            "matrix": self.matrix
        }
        with open('Result/busMatrix.json', 'w', encoding='utf8') as outfile:
            json.dump(dict, outfile, ensure_ascii=False)

    def loadMatrix(self, filename):
        with open(filename, 'r', encoding='utf8') as outfile:
            data = json.load(outfile)
            self.listEdges = data['listEdges']
            mapEdges = data['mapEdges']
            matrix = data['matrix']

        self.matrix = {}
        for i in matrix:
            self.matrix[int(i)] = {}
            for j in matrix[i]:
                self.matrix[int(i)][int(j)] = matrix[i][j]
        self.mapEdges = {}
        for i in mapEdges:
            self.mapEdges[self.decryption(i)] = mapEdges[i]
        print(f"Length of total: {len(self.totalEdges)}")
```

3.4. Result

Through 4 times running the `process()` function, I conclude that the average running time for this function is 1237.885 seconds and the avarage auxiliary space for this function is 2.17GB.

The result of the matrix is a JSON file with 70MB size. The below picture shows a part of the file (`busMatrix.json`):

206485, 206491, 206493, 206495, 206497, 206500, 206501, 206504, 206507, 206508, 206520, 206567, 206568, 206569, 206570, 206588, 206590,
206591, 206592, 206594, 206595, 206596, 206597, 206607, 206620, 206629, 206630, 206714, 206715, 206751, 206756, 206780, 206781, 206784,
206790, 206840, 206855, 206856, 206875, 206876, 206909, 206910, 206912, 206913, 206914, 206915, 206916, 206917, 206918, 206919, 206920,
206921, 206968, 206969, 206970, 206971, 206972, 207009, 207010, 207020, 207023, 207024], "matrix": {"0": {"508": [196965, 2], "2167": [93836,
9], "2521": [188472, 8], "3969": [188472, 23], "4029": [188472, 33], "4043": [188472, 39], "4044": [188472, 9], "4045": [188472, 5], "4046":
[188472, 24], "4047": [169204, 5], "4138": [188472, 7], "4139": [188472, 5], "6296": [93840, 4], "6304": [33538, 4], "6486": [188472, 39],
"6487": [188472, 32], "6488": [188472, 28], "7935": [188472, 6], "7938": [188472, 7], "8161": [188472, 27], "10059": [188472, 8], "10249":
[188472, 21], "10250": [188472, 39], "10251": [188472, 34], "10252": [188472, 10], "13163": [169204, 5], "15414": [188472, 4], "15475":
[196965, 11], "15684": [188472, 5], "15953": [188472, 29], "15955": [188472, 35], "15958": [188472, 33], "15961": [188472, 15], "16038":
[188472, 34], "16039": [188472, 39], "16044": [188472, 39], "17184": [188472, 33], "17186": [188472, 13], "17191": [188472, 33], "18097":
[188472, 22], "18859": [93840, 1], "18878": [188472, 8], "18879": [188472, 18], "18880": [33538, 4], "22997": [188472, 5], "23110": [188472,
16], "23850": [188472, 13], "26806": [187928, 11], "28417": [188472, 28], "29623": [188472, 39], "29657": [188472, 29], "30888": [188472,
39], "33286": [93836, 2], "33328": [188472, 7], "33538": [188472, 39], "33543": [188472, 39], "33547": [188472, 31], "33554": [93836, 19],
"33556": [196965, 10], "33561": [196965, 4], "33563": [93836, 2], "35745": [93836, 3], "37023": [188472, 34], "39598": [188472, 24], "50625":
[188472, 4], "50898": [188472, 32], "52343": [188472, 5], "52344": [188472, 16], "53139": [188472, 13], "57460": [188472, 6], "57488":
[188472, 39], "57490": [33554, 14], "59371": [33554, 10], "59372": [33554, 20], "63942": [188472, 10], "63943": [188472, 18], "65504":
[188472, 25], "66753": [188472, 13], "67310": [188472, 7], "67313": [188472, 39], "72602": [188472, 5], "77486": [169204, 5], "93831":

This matrix just store the edges that are available. If the result of the two edges is nothing, I will not store it into the matrix.
One more thing is that the number of edges in this matrix is just the mapping numbers that are displayed in `self.mapEdges`, which I also save it in this file:

busMatrix.json


```
{"mapEdges": {"5758104203-373543511": 0, "366459052-5764885888": 1, "5764885888-366459052": 2, "366459052-366409867": 3,
"366409867-366459052": 4, "366454835-366405484": 5, "366405484-366454835": 6, "5738009490-366451260": 7, "366451260-5738009490": 8,
"366442479-5735403820": 9, "5735403820-366442479": 10, "366369613-5752843085": 11, "5752843085-366369613": 12, "366379023-3110867029": 13,
"3110867029-366379023": 14, "5811673555-7393930262": 15, "7393930262-5811673555": 16, "366391224-366418696": 17, "366418696-366391224": 18,
"366414641-366404504": 19, "366404504-366414641": 20, "366452436-1516645253": 21, "1516645253-366452436": 22, "370818177-696860119": 23,
"696860119-370818177": 24, "366452436-8556388353": 25, "8556388353-366452436": 26, "366369507-366430275": 27, "366430275-366369507": 28,
"366450125-366450784": 29, "366450784-366450125": 30, "2434647841-5765289876": 31, "5717194968-5719079067": 32, "5719079067-5717194968": 33,
"366422719-366471322": 34, "366471322-366422719": 35, "366469460-366403668": 36, "366403668-366469460": 37, "1996655233-6770923492": 38,
"366441747-5738009486": 39, "5738009486-366441747": 40, "6015565866-2042473306": 41, "2042473306-6015565866": 42, "366465467-366472951": 43,
"366472951-366465467": 44, "5092125047-5772194286": 45, "5772194286-5092125047": 46, "2289950073-366463412": 47, "366463412-2289950073": 48,
"366378323-6757924454": 49, "6757924454-366378323": 50, "2304882882-366381772": 51, "366381772-2304882882": 52, "366463894-366419182": 53,
"366419182-366463894": 54, "366426378-366439942": 55, "366439942-366426378": 56, "366439942-366403237": 57, "366403237-366439942": 58,
"366444636-366402428": 59, "366402428-366444636": 60, "366402428-366426378": 61, "366426378-366402428": 62, "366402428-366439942": 63,
"366439942-366402428": 64, "366397693-2247591674": 65, "2247591674-366397693": 66, "366470281-366381010": 67, "366381010-366470281": 68,
"6793454651-5738447167": 69, "5738447167-6793454651": 70, "3134555292-5785399174": 71, "5785399174-3134555292": 72, "1894608446-1767821197":
73, "366422378-366397092": 74, "366397092-366422378": 75, "1499680420-366401789": 76, "366401789-1499680420": 77, "366451860-366477183": 78,
"366477183-366451860": 79, "366372095-5811673555": 80, "5811673555-366372095": 81, "366368194-2036083396": 82, "2036083396-366368194": 83,
"571360922-366445426": 84, "366445426-571360922": 85, "4595642998-6251878000": 86, "6251878000-4595642998": 87, "6736191037-366452673": 88,
"366452673-6736191037": 89, "7190189892-366410223": 90, "366410223-7190189892": 91, "1996655726-1996655766": 92, "1996655766-1996655726": 93,
"4585919063-366454009": 94, "366454009-4585919063": 95, "5811481199-6725173487": 96, "366423209-5794916694": 97, "5794916694-366423209": 98,
"5763233415-366392636": 99, "366392636-5763233415": 100, "366411839-5765244961": 101, "5765244961-366411839": 102, "11082694019-745747032":
```


Since JSON file does not allow us to save the keys of the dictionary in the format of a *tuple*, so I had to convert that to a *string* by:

```
    def combine(self, u):

        # combine the edges of the node
        # u is a tuple of 2 node (a, b)
        return f"{u[0]}-{u[1]}"

    def decryption(self, u):
        # decryption the edges of the node u
        # u is a string of 2 node "a-b"
        return tuple(map(str, u.split('-')))
```

3.5. Query and Testing

Since we have the matrix saved in a file, we can easily output the corresponding row if we query an edge, then output the result to a file.

The steps in the `query()` function:

- Map the edge by `self.mapEdges`.
- Then loop for all the edges and append the value obtained by `self.matrix` to a result list.

Below is the code for `query()` function:

```
def query(self, listEdges):
        # input = a list of edges [[node1, node2], [node3, node4], ...]
        # output = the corresponding row in the matrix
```

```
        cur = times.time()
        for i in range(len(listEdges)):
            listEdges[i] = (listEdges[i][0], listEdges[i][1])
        result = []
        for i in range(len(listEdges)):
            index = self.mapEdges[self.findWay(listEdges[i])]
            temp = []
            for j in range(len(self.listEdges)):
                if (self.matrix[index].get(self.listEdges[j]) == None):
                    temp.append(None)
                else:
                    #print(self.matrix[index][self.listEdges[j]][0])
                    temp.append(self.totalEdges[self.matrix[index][self.listEdges[j]][0]])
            result.append(temp)

        # output the result
        with open('Result/query.json', 'w', encoding='utf8') as outfile:
            for i in range(len(result)):
                json.dump({'edges': listEdges[i] ,'row': result[i]}, outfile,
ensure_ascii=False)
                outfile.write('\n')

        print(f"Done query in {times.time() - cur} seconds")
```

The time complexity of this function is $O(N.L)$, where L is the length of the query list (`listEdges`).

Testing for this query in driver code:

```
# obj = BusGraph('InputFiles/bus-history.json', mode = 1)
obj = BusGraph('Result/busMatrix.json', mode = 0)
obj.query([['5738158912', '373543511'],["696860148","366479091"],["366415838","366426086"]])
```

We got this file in `0.04635334014892578` seconds. In this file, each query is a seperated line which shows the row in the edge matrix. You can easily get access to this file (`query.json`) via GitHub in [6] or my Drive link in [7].


```
null, null, null, null, null, null, null, null, null, null, null, null, null, null, null, null, null, null, null, null, null, null,
null, null]}
{"edges": ["696860148", "366479091"], "row": [null, null, ["696860153", "724227317"], ["2030639513", "676546410"], null, null, null, null,
null, null, null, ["2659114286", "2361894737"], null, null, null, null, null, null, null, null, null, null, null, null, null, null,
null, null, ["366404527", "366440827"], null, null, null, null, null, null, null, ["696860153", "724227317"], null, null, null, null, null,
null, null, null, null, null, null, ["6770516199", "2353078006"], null, null, null, null, null, null, null, null, null, null, null, null,
["5737979169", "366476912"], null, null, null, null, null, null, null, null, ["6770516199", "2353078006"], null, null, ["1497255633",
"3700550789"], null, null, null, null, null, null, null, null, null, null, null, null, null, null, ["366404527", "366440827"], null,
null, null, null, null, null, null, null, null, null, null, null, null, null, null, null, null, null, null, null, ["5737980762",
"366389394"], null, null, null, null, null, null, null, null, null, null, null, null, null, null, null, null, null, null, null, null, null,
null, null, null, null, null, null, null, null, null, null, null, null, null, null, null, null, null, null, null, null, null, null, null,
null, null, null, null, null, null, null, null, null, null, null, null, null, null, null, null, null, null, null, null, null, null, null,
null, null, null, ["6770516199", "2353078006"], null, ["2659114287", "2361894737"], null, null, null, null, null, null, null, null, null,
["1497255633", "3700550789"], null, null, null, null, null, null, null, null, null, ["1497255633", "3700550789"], null, null, null, null,
null, null, null, null, null, null, null, null, null, null, ["696860153", "1271148299"], ["2351201590", "366435365"], null, null, null, null,
null, null, null, null, null, null, null, null, null, null, ["6794489382", "6639607755"], null, null, ["2351201566", "2351201590"], null, null,
null, null, null, null, null, null, null, ["1497255633", "3700550789"], null, null, null, ["2024973315", "9415478909"], null, null, null,
["6332563146", "2351201562"], ["2351201566", "2351201590"], null, null, null, null, ["4631890834", "2434653577"], null, null, null, null, null,
null, null, null, null, null, null, null, null, null, null, null, null, null, null, null, null, null, ["696860153", "724227317"], null, null, null, null,
null, null, null, null, null, null, null, null, null, null, null, null, null, null, null, null, null, null, ["6770516199", "2353078006"], null, null, null,
```

4. Conclusion

Through enormous time and dedication for this task, I achieved some significant results based on the data, which is the Ho Chi Minh City map and the historical path of bus routes in Ho Chi Minh City.

Firstly, the .osm file has been read and sucessfully converted to a graph with 172945 nodes and 205802 edges in section 2.3. Below is a part of the graph that is completely saved to a JSON file (`adjNode.json`).

Result > {} adjNode.json > {} 366454835

```
{"5758104203": {"373543511": {"id": 31096786, "nodes": ["5738158912"]}, "5758104180": {"id": 607624747, "nodes": ["5758104202", "5758104174",
"5758104201", "5758104176", "5758104177", "5758104179", "5758104200"]}}, "366459052": {"5764885888": {"id": 32575737, "nodes":
["10941379759", "6627102193"]}, "366409867": {"id": 32575738, "nodes": ["2763159054", "2763159055"]}, "366464981": {"id": 32578144, "nodes":
[]}, "366419914": {"id": 32579757, "nodes": []}}, "5764885888": {"366459052": {"id": 32575737, "nodes": []}, "366409525": {"id": 32583517,
"nodes": ["366426162", "366405771", "366385674", "8345921466", "5767654818", "366471067", "5767653023", "8345281374"]}}, "366409867":
{"366459052": {"id": 32575738, "nodes": []}, "5753164989": {"id": 606922367, "nodes": []}, "366384739": {"id": 610123569, "nodes": []}},
"366454835": {"366405484": {"id": 32575749, "nodes": ["5778111885", "366427628", "5778134381", "5778132548", "366369758", "5778111884",
"366408990", "5778134380"]}, "5763233406": {"id": 32579105, "nodes": []}, "366371763": {"id": 1178245700, "nodes": ["5778134375"]}},
"366405484": {"366454835": {"id": 32575749, "nodes": []}, "5778132544": {"id": 610067391, "nodes": []}, "366465954": {"id": 695758697,
"nodes": []}}, "5738009490": {"366451260": {"id": 32575751, "nodes": ["5738009489", "3132376460", "3132376488", "3132376442", "5738009759",
"3132376495", "3132376450", "5738009503", "366379008", "366384631", "366427334", "5738009505", "366447957", "366454002"]}}, "366451260":
{"5738009490": {"id": 32575751, "nodes": []}}, "366442479": {"5735403820": {"id": 32575754, "nodes": ["2477301439", "366473779",
"3202547145"]}}, "5735403820": {"366442479": {"id": 32575754, "nodes": []}}, "366369613": {"5752843085": {"id": 32575768, "nodes":
["5795144851", "366418963", "366373068", "5753228946", "5795144815", "366378718", "5795144828", "5795144841", "5795144839", "4107389720",
"5752843056", "366442490", "5753023085", "366387538", "5793457602", "5752843088", "5753023079", "366444706", "5752843089", "5795145096"]}},
"5752843085": {"366369613": {"id": 32575768, "nodes": []}, "5753023449": {"id": 606900992, "nodes": ["5753023453", "5753023452",
"5793457594", "5753023451", "5753023450", "5753023098", "5753023094", "5793457590", "5753023099", "5753023100", "5753023101", "5793457578",
"5753023102", "5753023103"]}, "5795145099": {"id": 612171169, "nodes": ["4937800722", "366379700", "4937800723", "5752843104", "4937792239",
"4937792233", "4937792235", "4937792240", "5795145081", "4937792242", "5753059981", "5753059976", "5753059977", "4937792246", "5753059940",
"366421415", "5753059962", "5753059961", "5753059960", "5795145098"]}}, "366379023": {"3110867029": {"id": 32575769, "nodes": ["366407925",
"3110867010", "8360852434"]}}, "3110867029": {"366379023": {"id": 32575769, "nodes": []}, "2024972749": {"id": 306238739, "nodes":
["3110867035", "9131563160", "3110867048", "3110867043", "3110867036", "3110867041"]}, "366474368": {"id": 1024850276, "nodes": []}},
"5811673555": {"7393930262": {"id": 32575784, "nodes": ["366453642", "366419656", "366454788", "7393930257", "7393930258", "7393930259",
```


Secondly, I have taken all of my efforts and eventually created a edge matrix with size N x N, which consists of the most frequently occuring edges between all pairs of edges in the data.

In addition, section 3.3 introduced two ways to compute and store the matrix effectively as well as their benefits and drawbacks. Then, section 3.5 proposed the way to take the corresponding row of an edge when taking a query and save it to a file.

5. Reference

These are the website that I have looked for information about OSM files and algorithms, including:

[1]. XML Definition - What is XML? (techterms.com)

[2]. OSM File - What is an .osm file and how do I open it? (fileinfo.com)

[3]. osmium · PyPI

[4]. pandas - How to extract and visualize data from OSM file in Python - Stack Overflow

[5]. Combinations Using Itertools Doesn't Have To Be Hard - Python Pool

This is my GitHub repository link:

[6]. GitHub Source Code for CS163 Solo Project (github.com)

And this is my Google drive link for the result file:

[7]. CS163-23APCS2-Solo-23125028-ResultFiles