# A blue and white cover with a map Description automatically generatedCover Page

Contents

[Cover Page 1](#_Toc175192287)

[Contents 3](#_Toc175192288)

[**1.** **Problem Statement and File Description** 4](#_Toc175192289)

[1.1. Problem Statement 4](#_Toc175192290)

[1.2. Descirption of two input files 4](#_Toc175192291)

[*1.2.1. HoChiMinh.osm* 4](#_Toc175192292)

[*1.2.2. bus-history.json* 5](#_Toc175192293)

[**2.** **Organize the .osm file to graph structure** 7](#_Toc175192294)

[2.1. What is .osm file? 7](#_Toc175192295)

[2.2. Read HoChiMinh.osm file 7](#_Toc175192296)

[2.3. Structurize the data 8](#_Toc175192297)

[**3.** **Edge Matrix creation** 9](#_Toc175192298)

[3.1. First Approach 9](#_Toc175192299)

[3.2. Alternative Approach 10](#_Toc175192300)

[3.3. Result and Testing 12](#_Toc175192301)

[**4.** **Conclusion** 14](#_Toc175192302)

[**5.** **Reference** 15](#_Toc175192303)

# **Problem Statement and File Description**

## 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 *osmium* 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:

A screen shot of a computer

Description automatically generated

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, …)

A screen shot of a computer program

Description automatically generated

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.

A screen shot of a computer

Description automatically generated

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:

A screen shot of a computer code

Description automatically generated

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.

# **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**](https://techterms.com/definition/xml) data in the form of "nodes" (points), "ways" (connections), and "relations" (street and object properties, such as tags).

## 2.2. Read HoChiMinh.osm file

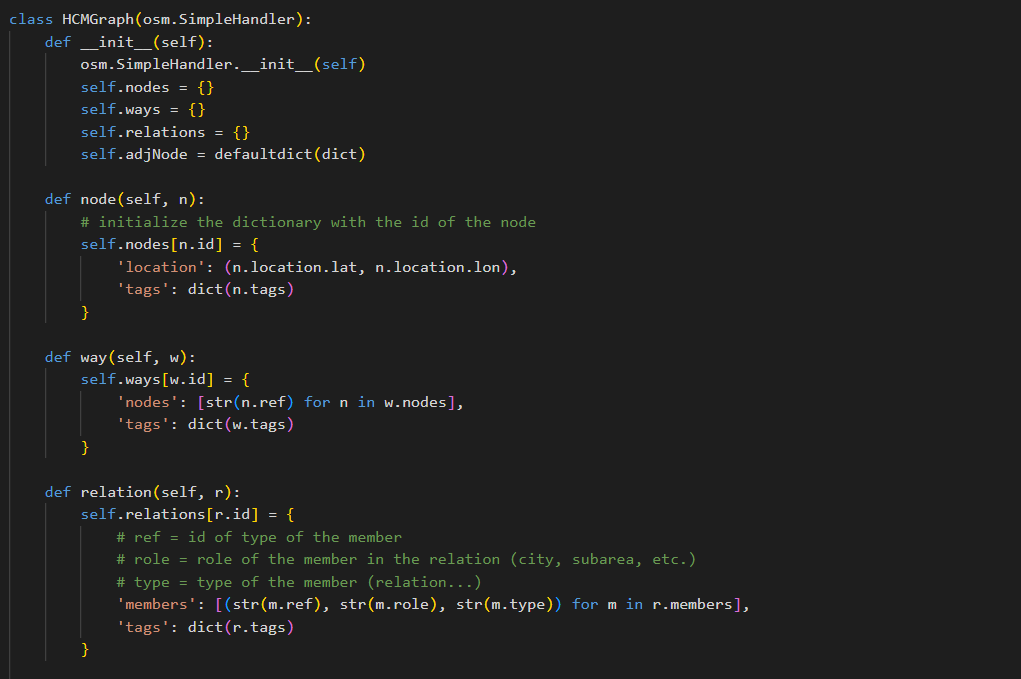
To read an OSM file in Python, we need to install osmium from pip – a library for processing OSM files.

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 picture:



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

A black background with white text

Description automatically generated

## 2.3. Structurize the data

Now we’d got the data for structurizing,

# **Edge Matrix creation**

Path caching is a technique used to enhance the efficiency of algorithms tasked with finding the shortest paths in graphs. This method involves storing previously computed paths so they can be quickly retrieved when the same paths are needed again, rather than recalculating them from scratch. Path caching is particularly advantageous in applications with high-frequency, repetitive path queries, such as in navigation systems, network routing, and various optimization problems.

## 3.1. First Approach

Since the stops in the bus map have the attributes “Zone”, and my observations indicate that the “Zone” in each stops is always a string (the zone is never empty) as well as that string is a district in Ho Chi Minh City. So, what I am trying to do here is to find the common shortest path in all pairs of districts, and these shortest path comes from my all-pair Dijkstra’s algorithm search.

For example, my search is coming to the stops having their stopID is 35 (source) and 3550 (end) and have got the shortest path through Dijsktra’s algorithm.

The algorithm will search for their zones in “stops” dictionary (which is ‘Quận 1’ and ‘Huyện Củ Chi’), then check if the cache has stored the path from these zones or not. If not, simply create a new key to store the list of resulting path as well as the time taken to get to 3550 from 35. Otherwise, I will try to get the common path between the newly created list path and the available path in the cache and store it along with the cost into the cache again.

Therefore, the basic algorithm is:

- For each stop u, run the Dijkstra’s algorithm to get the shortest path from u to all the other stops.

- Then, for each stop, we will get the resulting path and time consuming to get to the destination along with their zones.

- After that, we will look for the key in the cache for checking if the cache has stored the path from these zones or not.

+ If not, create a new key between the zones and store the list path into it.

+ Otherwise, get the common path bet between the newly created list path and the available path in the cache and store it along with the cost into the cache again.

When the searching ends, we store the cache that we have updated into a .JSON file named “cache.json”. So now, we have a file storing the shortest path between two zones and the time taken to go through it as well. In the Query section, we can reuse it by simply read it and apply the combined search to enhance the efficiency and time of the query.

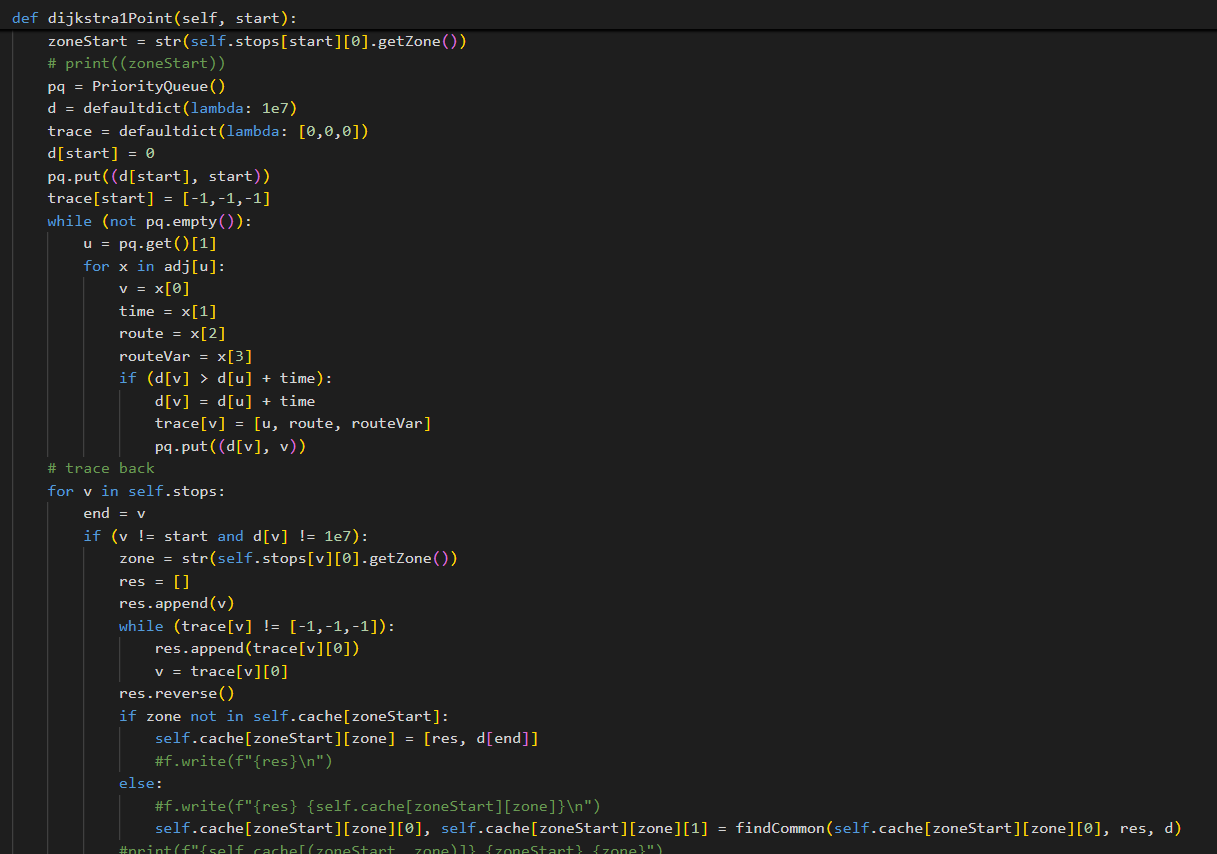
## 3.2. Alternative Approach

Below picture is the implementation of the *caching()* function. It takes in the graph that we have done in CS162 (the *importGraph* function) and output a file with the details of the cache.

A screen shot of a computer program

Description automatically generated

Here is the breakdown of the *dijkstra1Point()* function:



It is simply Dijkstra’s algorithm combined with checking the zones and appending (or modifying) the key in the cache to store the shortest path between the districts.

The caching function has a complexity of , since we have to perform V seaching functions, which have complexity of .

The result of *caching* function is shown below. (due to the size of the file, I will just show a viewport of the file). If no common path in the two zones was found, the resulting cache will be [] and 0.

A screen shot of a computer screen

Description automatically generated

## 3.3. Result and Testing

After creating a cache file for storing the shortest path between the districts in Ho Chi Minh City, we come to the query section for printing out the shortest path between two stops.

To do this, we firstly find the zones of the starting stop u and the ending stop v, then check if the shortest path between two zones is empty.

If this shortest path is empty, we just simply find the regular shortest path between two stops.

If not, we name the shortest path between two zones ‘path’. We will find the shortest path from u to path[0] as well as the shortest path from path[-1] to v, then simply add all the path that we found to the ‘path’ list.

So, what algorithm should we utilize to find the shortest path from u to path[0] and from path[-1] to v?

My answer is, A\* Search Algorithm. Compared with other search algorithms like Dijkstra’s algorithm or Floyd-Warshall algorithm, which explores all directions around the starting node, A\* uses heuristics to estimate the cost from a node to the goal, thereby optimizing the search process and reducing the computational load.

Below figure is the implementation of the query combined with A\* Search Algorithm.

A screen shot of a computer program

Description automatically generated

This is the output when testing 3550 as the starting stop and 3683 as ending stop:

Path caching (combined with A\*):

[3550, 3552, 3553, 3555, 3554, 3556, 3557, 3558, 3559, 3560, 3561, 3563, 3562, 3564, 3565, 3566, 1185, 1206, 1205, 1207, 1208, 1211, 1209, 1212, 1210, 1214, 1216, 1213, 1218, 3232, 7608, 1215, 1222, 1217, 1219, 1224, 1225, 1165, 1221, 1223, 4746, 1227, 4747, 1228, 1230, 1232, 4588, 1231, 1235, 1234, 1393, 1239, 167, 172, 169, 174, 607, 610, 609, 611, 900, 898, 902, 2835, 2837, 2840, 1397, 2842, 2839, 2841, 2844, 466, 472, 467, 1051, 1347, 18, 22, 21, 23, 24, 25, 26, 1344, 1196, 1194, 1198, 1197, 1201, 1199, 1202, 1200, 1204, 835, 7070, 1257, 1260, 1261, 1258, 1356, 1357, 1359, 1361, 1362, 1358, 1360, 1366, 1055, 1058, 1061, 1063, 1060, 1065, 1062, 1066, 1064, 2138, 2137, 2141, 2139, 2144, 3662, 3664, 3663, 3666, 3665, 3667, 3669, 3670, 3671, 3683]

Time: 122.66109437702957

Query time: 0.00281405448913574219

Compared to regular A\* Search algorithm and Dijkstra’s algorithm, the query time has improved significantly since the cache worked effectively on the far points (the two zones are ‘Huyện Củ Chi’ and ‘Huyện Cần Giờ’, which is about 75 kilometers apart).

# **Conclusion**

Utilizing advanced algorithms like A\* Search, Path Caching, and Contraction Hierarchy can significantly enhance the efficiency and performance of solving the shortest path problem in complex graphs, such as those representing urban transportation networks.

The A\* Search algorithm, with its heuristic-driven approach, effectively narrows down the search space, leading to faster pathfinding by focusing on the most promising routes. Path Caching further optimizes performance by storing and reusing previously computed paths, reducing redundant calculations and providing rapid responses to repetitive queries. Contraction Hierarchy, through its preprocessing steps, restructures the graph to allow swift traversal by minimizing the number of nodes and edges that need to be considered during queries.

Each of these techniques offers distinct advantages: A\* Search is particularly effective for real-time pathfinding with its balance of accuracy and speed; Path Caching excels in environments with high query repetition, significantly cutting down on computational load; and Contraction Hierarchy provides unparalleled query performance after an initial preprocessing phase, making it ideal for large-scale, static graphs. When used in conjunction, these algorithms can address the limitations of traditional methods like Dijkstra’s algorithm, providing robust and scalable solutions for modern pathfinding challenges. By leveraging the strengths of A\* Search, Path Caching, and Contraction Hierarchy, systems can achieve a high level of efficiency, accuracy, and responsiveness, essential for applications ranging from GPS navigation to network routing.

# **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)](https://techterms.com/definition/xml)

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

[3]. [osmium · PyPI](https://pypi.org/project/osmium/)

[4]. [pandas - How to extract and visualize data from OSM file in Python - Stack Overflow](https://stackoverflow.com/questions/45771809/how-to-extract-and-visualize-data-from-osm-file-in-python)

[5]. [Combinations Using Itertools Doesn't Have To Be Hard - Python Pool](https://www.pythonpool.com/itertools-combinations/)

This is my GitHub repository link:

[6]. [GitHub Source Code for CS163 Solo Project (github.com)](https://github.com/ltdsword/cs163-apcs23-ltd.sword-solo)