UNIVERSITY OF SCIENCE,

HO CHI MINH

DIJKSTRA' s APPLICATION IN BUSMAP

Build a Graph of Vertices and Edges based on Stops in HCM's bus map
Query Big Data and Optimizing the handling process using OOP
Find the shortest path using Dijkstra's Algorithm for all pairs

TECHNICAL REPORT
W07

submitted for the Solo Project in Semester 2 – First Year

IT DEPARTMENT

Computer Science

by

Phan Tuan Kiet

Full name: Phan Tuan Kiet

ID: 23125062

Class: 23APCS2

Tasks achieved: 04/04

Lecturer:
Professor Dinh Ba Tien (PhD)
Teaching Assistants:
Mr. Ho Tuan Thanh (MSc)
Mr. Nguyen Le Hoang Dung (MSc)

2024

Contents

I) Structure

- *There are six files .py in the program:*

+ graph1.py: This has the class Graph1() but Graph1 is not used in the program as it is just an example of another way of reading data to the graph. However, this method of reading graphs is good but has large errors compared to other methods. The detailed method of reading this way will be discussed in the next session.

+ graph2.py: This is the main graph class that we will use to handle our data and run Dijkstra's Algorithm to find the shortest paths of all pairs and count the importance of all vertices. Also, optimization will be deployed in this graph2.

+ main.py: Where the user interface will be handled and all of the tasks/queries from users will be done here.

+ path.py: Class Path that we implemented in the previous week.

+ stop.py: Class Stop that we implemented in the previous week.

+ var.py: Class Var that we implemented in the previous week.

- *Output file includes all of the files that we extract from our programs:*

+ allShortestPathsGraph2.txt: The shortest path of all pairs in our graph. Start from stop_start_id to stop_end_id with the shortest time.

+ geoJsonGraph2.json: JSON file that contains the shortest paths between a start_stop_id to an end_stop_id that the users input in the main.py. We can put this JSON file on the web to see that all paths/routes related to consider whether that path is suitable in real life.

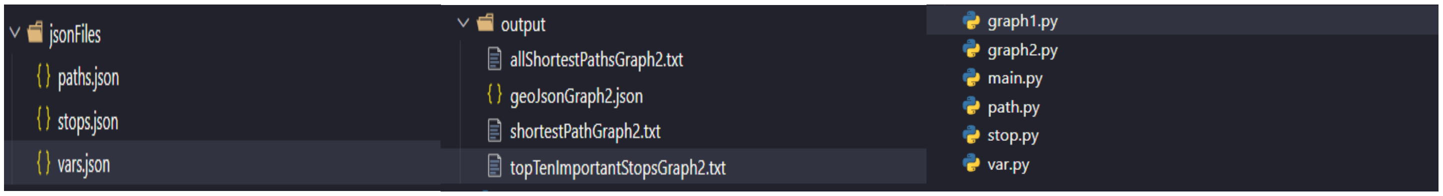
+ shortestPathGraph2.txt: Give the shortest path from a start_stop_id to an end_stop_id with all points it goes through. For example, shortest path from 1 to 10: 1->5->4->7->9->12->6->10. One of the paths from 4 to 7 is:

From 4->7 in RouteId: 131 and RouteVarId: 261: [106.66041565,10.75296211]->[106.66207886,10.753088]

+ topTenImportantStopsGraph2.txt: List out the top 10 stops that have the highest importance including all of its data like id, lat, lng, code,…

- *JSON files:*

+ Including all JSON files needed to read data into our graph: paths.json, stops.json, vars.json.


```
jsonFiles
  paths.json
  stops.json
  vars.json
output
  allShortestPathsGraph2.txt
  geoJsonGraph2.json
  shortestPathGraph2.txt
  topTenImportantStopsGraph2.txt
graph1.py
graph2.py
main.py
path.py
stop.py
var.py
```

II) Comments and Approaches

A) Comments

- In order to build the graph that we want, firstly, we have to have data from vars.json, stops.json, and paths.json to make vertices, edges and weight. Taking advantage of the programs that we built last week to handle var, stop and path, we don't need to do this again. Instead, we just need to push data in three variables: varQuery, stopQuery and pathQuery.

$$varQuery = RouteVarQuery(vars.json)$$

$$stopQuery = StopQuery(stops.json)$$

$$pathQuery = PathQuery(paths.json)$$

- Notes that I have changed the RouteVarQuery, StopQuery and PathQuery to receive the address of the file so that they can find the correct paths to handle data.

```
def __init__(self, fileName):
    self.paths = []
    self.loadJson(fileName)
```

```
def __init__(self, fileName):
    self.StopList = []
    self.loadStopJson(fileName)
```

```
def __init__(self, fileName):
    self.routeVarList = []
    self.loadJson(fileName)
```

- Notice that the paths from one stop to another stop can vary as there are many routes, therefore we have to build the edges with [(time, distance), to_stop_id, (RouteId, RouteVarID)] so that we can distinguish the path from one stop to another. This will be helpful when we apply Dijkstra's Algorithm as we don't have to find out if there are any paths from this StopID to another StopID.

- The Dijkstra algorithm on this graph can be done using a priority queue to optimize the process of finding the shortest paths. Also, we need to have a dict of trace, dist to keep track of our shortest paths so that we can print the paths to the file.

- For finding the shortest paths of all pairs, as the data is quite big I won't print it under JSON type, but a normal Txt file. This will show the shortest paths of all pairs by time.

- As I mentioned above, for each kind of reading data to the graph, the errors will be different, therefore, I will choose the method that has the least error (in my opinion) and give suitable reasons for my choice.

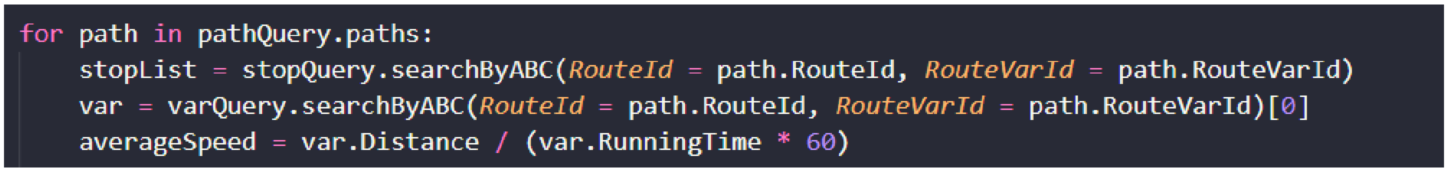
- Another comment is that we have to change our lat-lng to x-y coordinates for better calculation. We will use the pyproj library to do this.

B) Approach

- Let’s dive into two methods we can put data into our graph. For the first method (graph1), I will give the code and explanation here, but for the second method, I will put it later to the detailed explanation so that the flow is maintained.

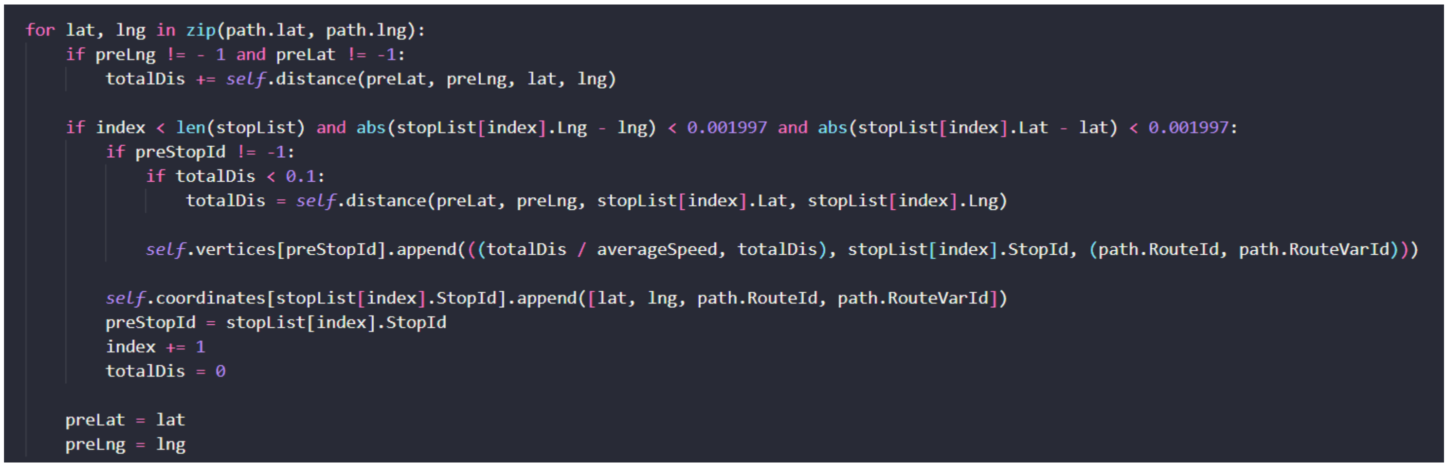
1) Graph1

- We will depend on the paths.json to search for corresponding Stops and Vars. We also find the average speed based on the information on the var.



```python
for path in pathQuery.paths:
    stopList = stopQuery.searchByABC(RouteId = path.RouteId, RouteVarId = path.RouteVarId)
    var = varQuery.searchByABC(RouteId = path.RouteId, RouteVarId = path.RouteVarId)[0]
    averageSpeed = var.Distance / (var.RunningTime * 60)
```

- We will maintain the *preLng, preLat, preStopId, totalDis, index* for this method. We will traverse through the path, if any lat-lng has a difference with the lat-lng of the stop less than 0.002 or 0.001997 (as I have checked for all the stops, this error will counter all of the stops in the data given), we will consider that coordinate or point in the path is a stop and add it to our graph.



```python
for lat, lng in zip(path.lat, path.lng):
    if preLng != - 1 and preLat != -1:
        totalDis += self.distance(preLat, preLng, lat, lng)

    if index < len(stopList) and abs(stopList[index].Lng - lng) < 0.001997 and abs(stopList[index].Lat - lat) < 0.001997:
        if preStopId != -1:
            if totalDis < 0.1:
                totalDis = self.distance(preLat, preLng, stopList[index].Lat, stopList[index].Lng)

            self.vertices[preStopId].append(((totalDis / averageSpeed, totalDis), stopList[index].StopId, (path.RouteId, path.RouteVarId)))

        self.coordinates[stopList[index].StopId].append([lat, lng, path.RouteId, path.RouteVarId])
        preStopId = stopList[index].StopId
        index += 1
        totalDis = 0

    preLat = lat
    preLng = lng
```

- Notes that we will update the totalDis between the stops, the preLat, preLng while we traverse through the paths. Also, there will be a point when the distance between two stops is zero (error), we will consider the distance between two stops the same as the distance between the preLat, preLng and the current stop.lat and stop.lng. In this way, no weight will be zero.

- Moreover, as the coordinates of each stop in different paths are not the same, so we have to save it in the self.coordinates with the RouteId and RouteVarId correspondingly.

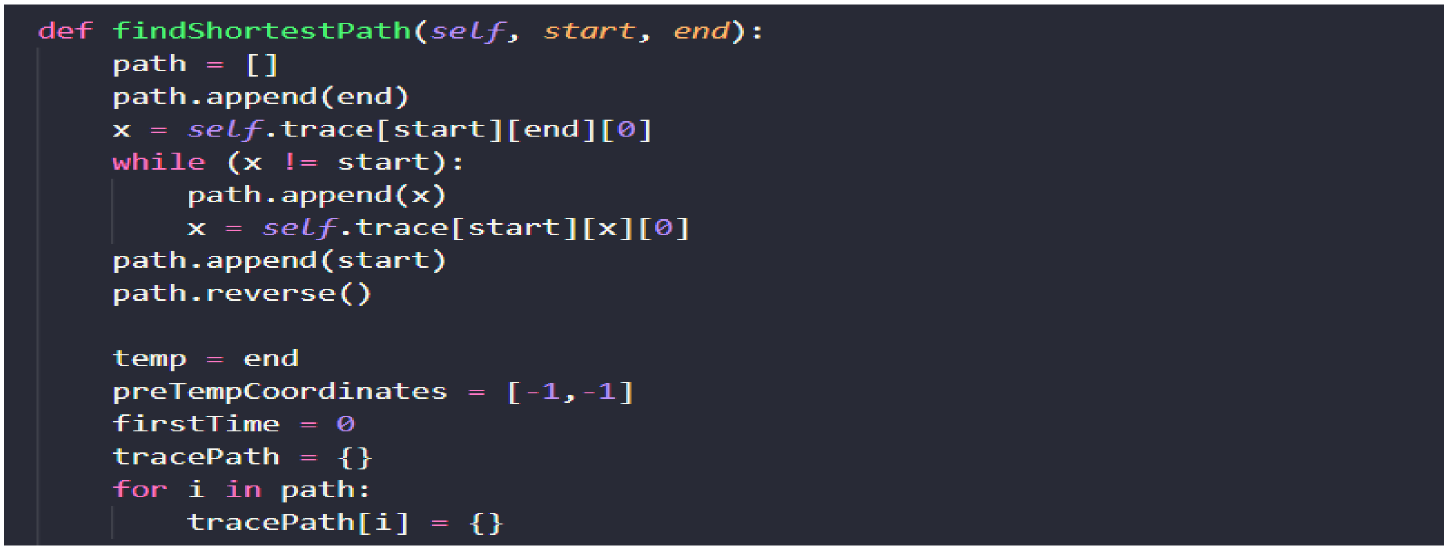
- The self.vertices will save our edges, for example:

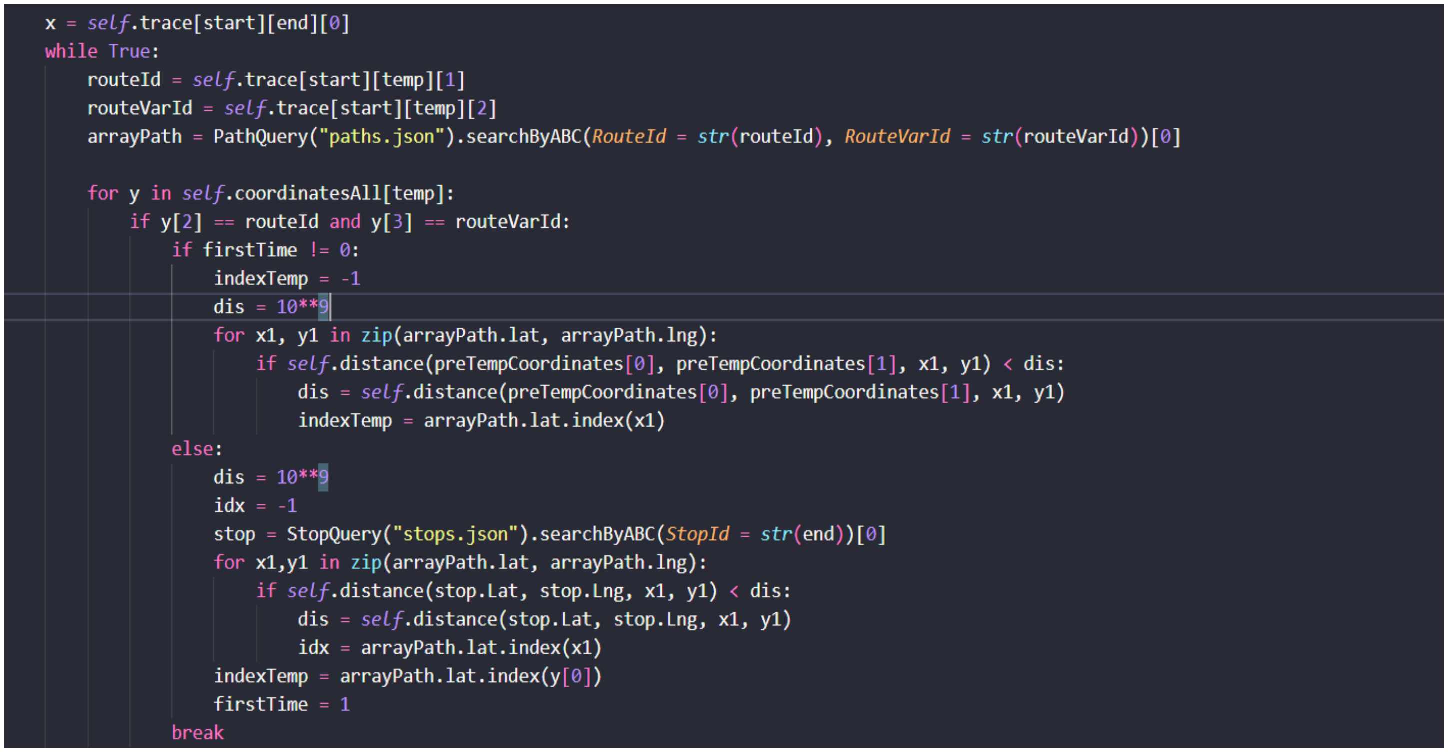
+ *self.vertices[1] = ((time, distance), to_stop_id, (RouteId, RouteVarId)).*

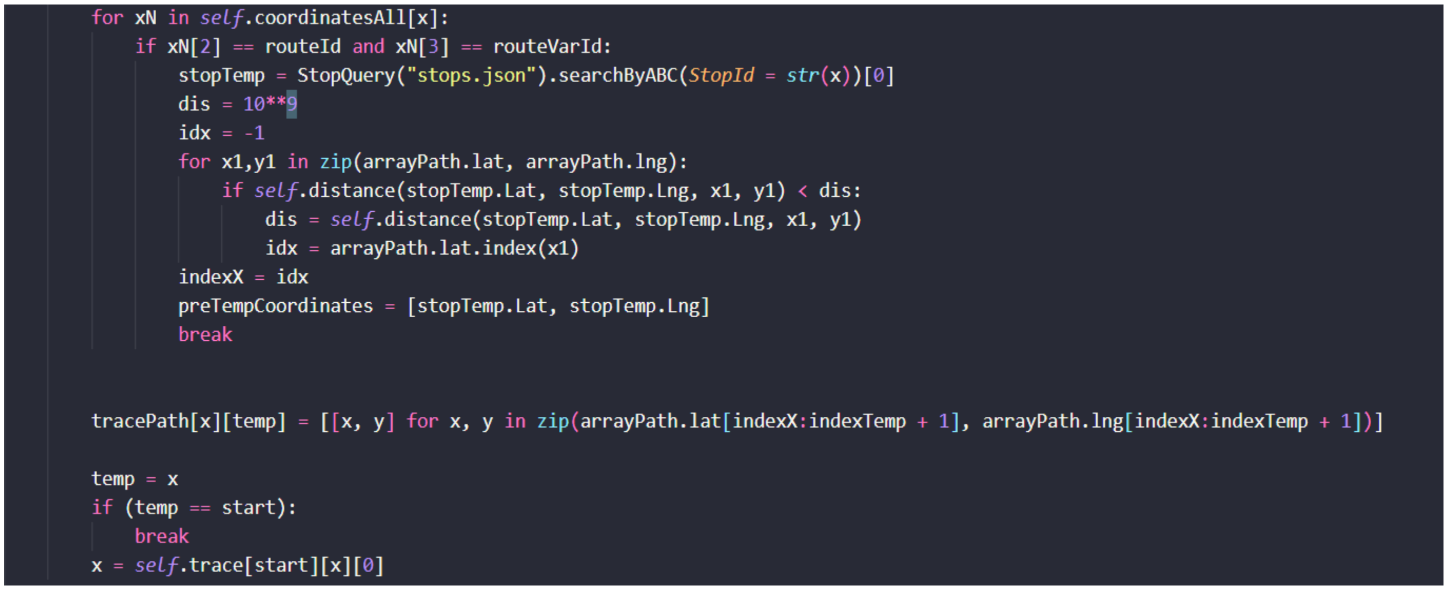
+ *self.coordinates[1] = (lat, lng, RouteId, RouteVarId).*

- In this way, we will connect all of the stops in our data together with the corresponding time, distance, RouteId and RouteVarId. However, this way of reading is quite complex and takes a lot of time to save and implement Dijkstra’s algorithm later on. The trace will be really complex when using this method (graph1).

- Why it is difficult to do this way, let see my codes when implementing Dijkstra’s algorithm using this method:


```python
def findShortestPath(self, start, end):
    path = []
    path.append(end)
    x = self.trace[start][end][0]
    while (x != start):
        path.append(x)
        x = self.trace[start][x][0]
    path.append(start)
    path.reverse()

    temp = end
    preTempCoordinates = [-1,-1]
    firstTime = 0
    tracePath = {}
    for i in path:
        tracePath[i] = {}
```


```python
    x = self.trace[start][end][0]
    while True:
        routeId = self.trace[start][temp][1]
        routeVarId = self.trace[start][temp][2]
        arrayPath = PathQuery("paths.json").searchByABC(RouteId = str(routeId), RouteVarId = str(routeVarId))[0]

        for y in self.coordinatesAll[temp]:
            if y[2] == routeId and y[3] == routeVarId:
                if firstTime != 0:
                    indexTemp = -1
                    dis = 10**9
                    for x1, y1 in zip(arrayPath.lat, arrayPath.lng):
                        if self.distance(preTempCoordinates[0], preTempCoordinates[1], x1, y1) < dis:
                            dis = self.distance(preTempCoordinates[0], preTempCoordinates[1], x1, y1)
                            indexTemp = arrayPath.lat.index(x1)
                else:
                    dis = 10**9
                    idx = -1
                    stop = StopQuery("stops.json").searchByABC(StopId = str(end))[0]
                    for x1,y1 in zip(arrayPath.lat, arrayPath.lng):
                        if self.distance(stop.Lat, stop.Lng, x1, y1) < dis:
                            dis = self.distance(stop.Lat, stop.Lng, x1, y1)
                            idx = arrayPath.lat.index(x1)
                    indexTemp = arrayPath.lat.index(y[0])
                    firstTime = 1
                break
```


```python
        for xN in self.coordinatesAll[x]:
            if xN[2] == routeId and xN[3] == routeVarId:
                stopTemp = StopQuery("stops.json").searchByABC(StopId = str(x))[0]
                dis = 10**9
                idx = -1
                for x1,y1 in zip(arrayPath.lat, arrayPath.lng):
                    if self.distance(stopTemp.Lat, stopTemp.Lng, x1, y1) < dis:
                        dis = self.distance(stopTemp.Lat, stopTemp.Lng, x1, y1)
                        idx = arrayPath.lat.index(x1)
                indexX = idx
                preTempCoordinates = [stopTemp.Lat, stopTemp.Lng]
                break


        tracePath[x][temp] = [[x, y] for x, y in zip(arrayPath.lat[indexX:indexTemp + 1], arrayPath.lng[indexX:indexTemp + 1])]

        temp = x
        if (temp == start):
            break
        x = self.trace[start][x][0]
```

- This is so many codes and It's hard to implement, although this still works, but not a good way of implementing it (in my opinion).

- Therefore, I think of another way to read the data into our graph, which is much more efficient → Graph2.

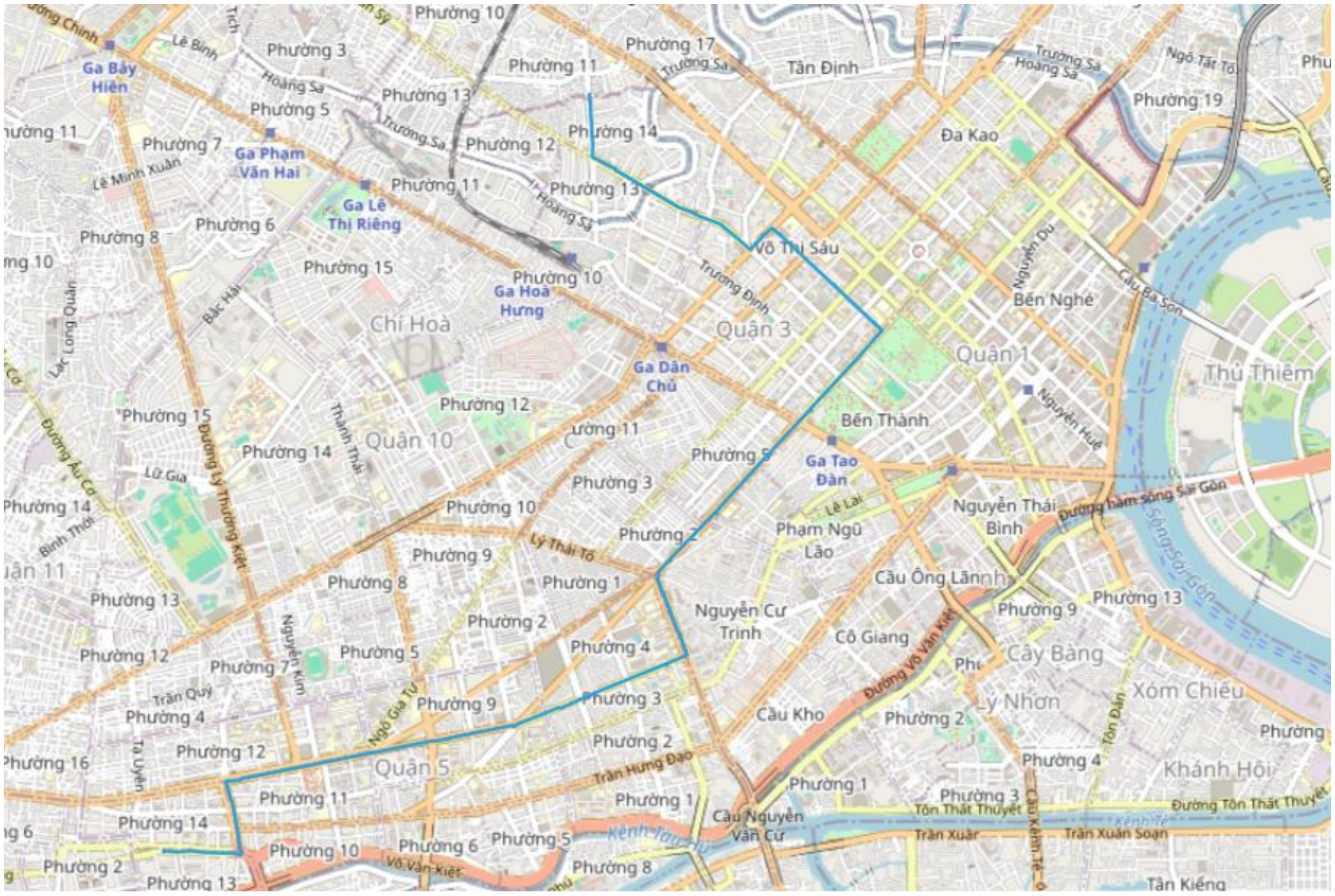
2) Graph2

- With graph 2, there is another way of reading the data into our graph. The idea is to query the Stops and Paths based on the Vars. We will traverse through each stop in a RouteVar and then consider the distance between a stop to all of the points in the paths, then we will choose the minimum to make that point in the path become our stop. Then for the next stop, we will traverse from the next points in that path, we don't have to traverse from the beginning again.

- In this way, our stops will be definitely in the paths and make a good route so that we can see how the bus goes. The code will be shown in the detailed explanation.

- The result of this kind of reading data is quite good when implementing Dijkstra's algorithm:

+ Shortest path from 2 to 450: 2->3->5->437->439->440->465->469->466->472->467->474->471->287->290->289->292->242->7589->2481->2483->2491->2494->2485->449->453->450



+ Notes that when reading data through this method, for better optimization, we should include a self.pathAll and self.timeAll for more convenience when we trace stops or want to see the time between random stops.

```
self.timeAll[startStopId][endStopId].append([time, routeId, routeVarId])
self.pathAll[startStopId][endStopId].append([path, routeId, routeVarId])
```

3) Dijkstra's algorithm for all pairs

- The approach that I will use for better optimisation will include the priority queue. For each stop, we will do Dijkstra's algorithm which results in *O(Elog(v))*. We will do this for all stops in our graph, therefore the total complexity in time is *O(V*E*log(v)) in the G(V, E, w).*

4) Dijkstra's algorithm for one-stop

- Why did I mention Dijkstra's algorithm for one-stop when we have already done it for all pairs, which means we can run it and trace for the stop we want? The reason is this kind of doing will take a lot of time, so I will run Dijkstra's algorithm for one vertice and then take the information we only want, not taking redundant information.

5) Count the importance of a stop in the graph

- To be honest, this is one of the most challenging parts of this week's project as the implementation is quite hard to do.

- For the naive method, we can loop through all stops/ vertices, and then do another two loops for each pair, if the vertice we are considering, the shortest path of that pair goes through that vertice, we will add it to the important[vertice]. But this method, for the worst part, will lead to the terrible complexity $O(n^3)$.

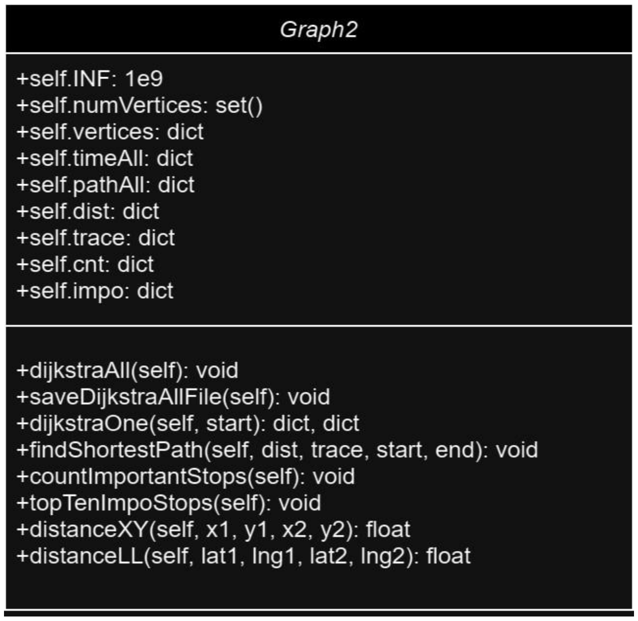
- Therefore my approach is to maintain a self.cnt (dict). Self.cnt will be the multiplication of the shortest path from other vertices to the considering vertice with the dp[u] – the shortest path from u to other vertices.

- Of course, we will exclude all of the edges that are not in the shortest paths and then sort topological to dp.

- Then the importance of a stop u will be: $\sum \mathrm{self.cnt}[v][u]$ for v in $\mathrm{self.numVertices}$.

III) Class diagram for Graph2

- This will be the class diagram for Graph2 including all of its attributes and methods:



```
Graph2

+self.INF: 1e9
+self.numVertices: set()
+self.vertices: dict
+self.timeAll: dict
+self.pathAll: dict
+self.dist: dict
+self.trace: dict
+self.cnt: dict
+self.impo: dict

+dijkstraAll(self): void
+saveDijkstraAllFile(self): void
+dijkstraOne(self, start): dict, dict
+findShortestPath(self, dist, trace, start, end): void
+countImportantStops(self): void
+topTenImpoStops(self): void
+distanceXY(self, x1, y1, x2, y2): float
+distanceLL(self, lat1, lng1, lat2, lng2): float
```

+ The (+) is public. The self.impo will be the importance of all stops, while the self.dist and self.trace is used to keep track of Dijkstra's algorithm. For all other attributes, I have already discussed above.

+ Notes that for the self.numVertices: set(). This will hold the distinct stops as a vertice in our graph.

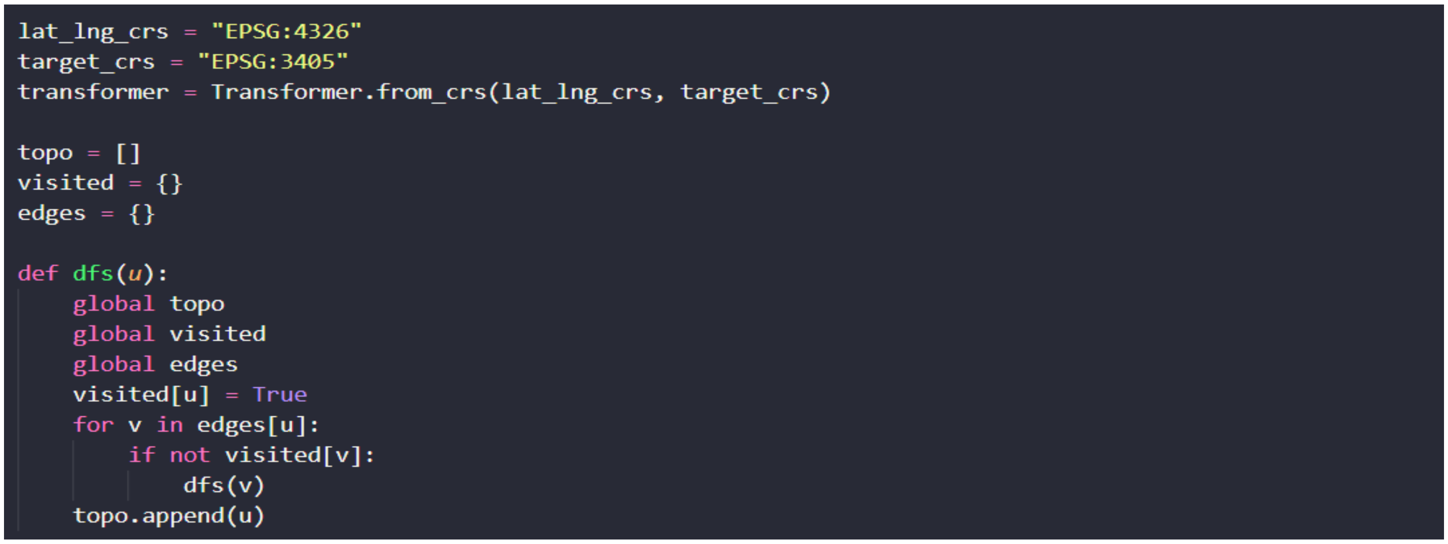
IV) Detailed explanation

- We will dive deeper into graph2.py and see how the reading graph and Dijkstra's algorithm were implemented!

1) Library used in the program

```
from var import *

from stop import *

from path import *

from math import sin, cos, sqrt, atan2, radians

from pyproj import Transformer

from time import sleep

from tqdm import tqdm

import heapq

import math
```

2) Functions and variables outside the Graph



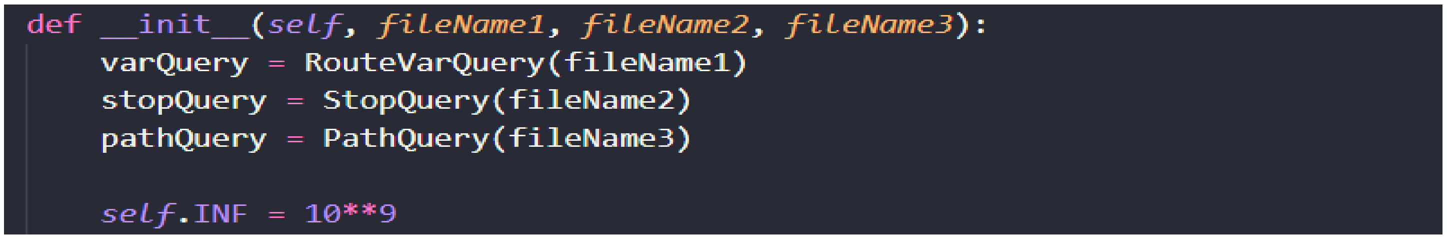
```
lat_lng_crs = "EPSG:4326"
target_crs = "EPSG:3405"
transformer = Transformer.from_crs(lat_lng_crs, target_crs)

topo = []
visited = {}
edges = {}

def dfs(u):
    global topo
    global visited
    global edges
    visited[u] = True
    for v in edges[u]:
        if not visited[v]:
            dfs(v)
    topo.append(u)
```

- For the *lat_lng_crs* and *target_crs*, they are used to transform lat-lng to x-y *(EPSG:4326 to EPSG:3405).*

- The topo, visited and edges are used for counting the importance of a stop by the approach mentioned above.

- The dfs(u) function is used for sorting topologically. Notice that edges[u] will be in the shortest paths.

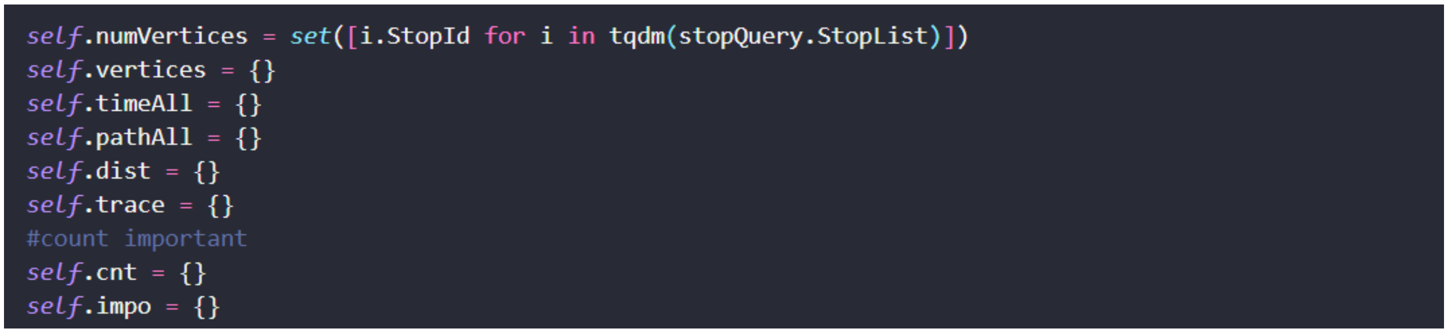
3) Reading Graph2 detailed explanation



```
def __init__(self, fileName1, fileName2, fileName3):
    varQuery = RouteVarQuery(fileName1)
    stopQuery = StopQuery(fileName2)
    pathQuery = PathQuery(fileName3)

    self.INF = 10**9
```

- Of course, first, we have to read data from the vars.json, stops.json and paths.json and init a self.INF = 1e9.

- Some of the initial dictionaries to save the essential information of vertices and edges of the graph *G(V, E, W):*



```
self.numVertices = set([i.StopId for i in tqdm(stopQuery.StopList)])
self.vertices = {}
self.timeAll = {}
self.pathAll = {}
self.dist = {}
self.trace = {}
#count important
self.cnt = {}
self.impo = {}
```

+ Self.numVertices: save distinct stops in our graph.

+ Self.vertices: save edges of our graph.

+ Self.timeAll: save time to go from a start_stop_id to an end_stop_id in a particular RouteVar and RouteVarId.

+ Self.pathAll: save paths to go from a start_stop_id to an end_stop_id in a particular RouteVar and RouteVarId.

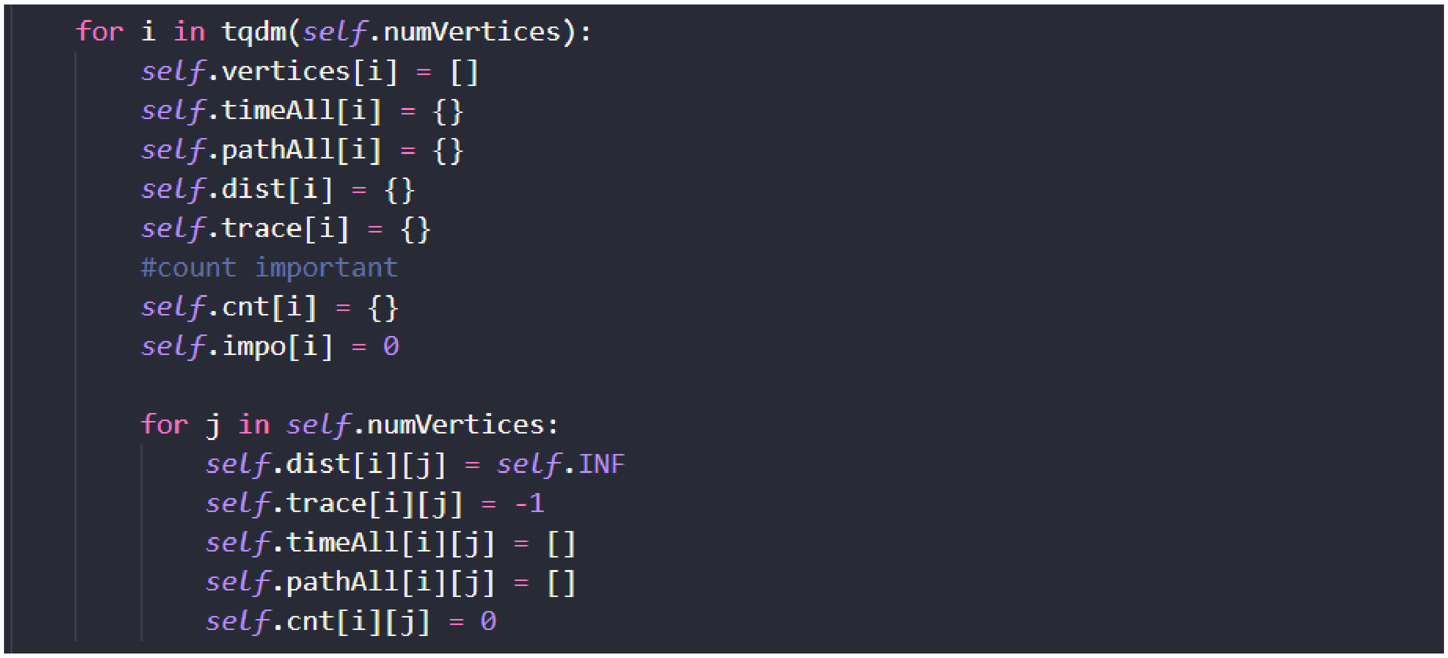
+ Self.dist: save the shortest time from a start_stop_id to an end_stop_id when applying Dijkstra's algorithm.

+ Self.trace: save the shortest paths from a start_stop_id to an end_stop_id when applying Dijkstra's algorithm.

+ Self.cnt: used to count all of the shortest paths from a start_stop_id to vertice u in the graph.

+ Self.imp: the importance of a stop in out graph after counting $\sum \text{self.cnt}[v][u]$ for v in self.numVertices.

- First, we will have to make the first setting for all of the dictionaries we made above:



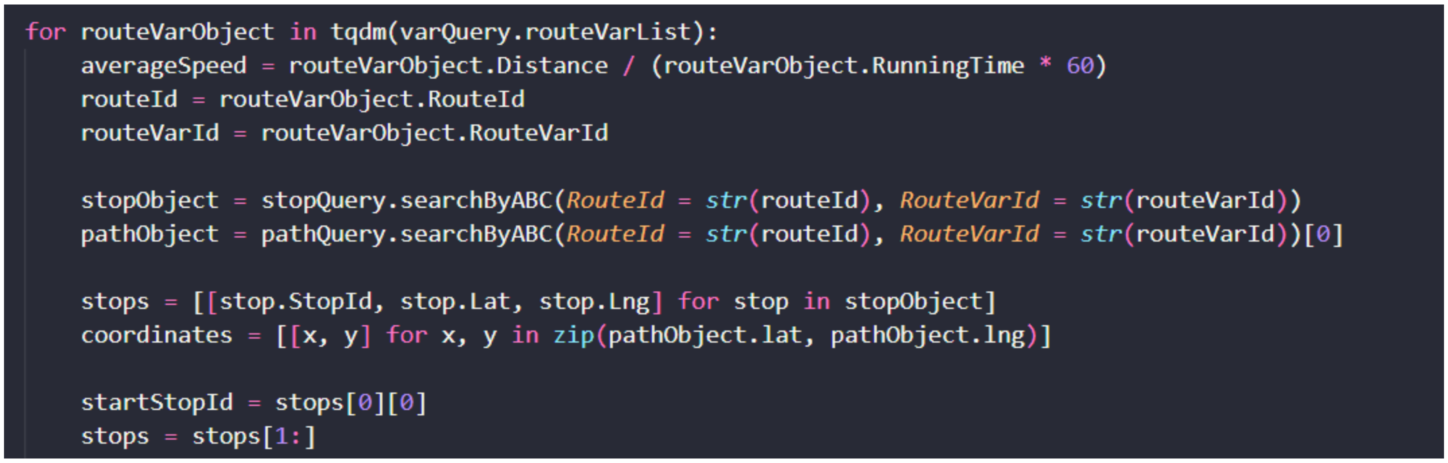
```
    for i in tqdm(self.numVertices):
        self.vertices[i] = []
        self.timeAll[i] = {}
        self.pathAll[i] = {}
        self.dist[i] = {}
        self.trace[i] = {}
        #count important
        self.cnt[i] = {}
        self.impo[i] = 0

        for j in self.numVertices:
            self.dist[i][j] = self.INF
            self.trace[i][j] = -1
            self.timeAll[i][j] = []
            self.pathAll[i][j] = []
            self.cnt[i][j] = 0
```

+ Notice that self.dis[i][j] = self.INF as for Dijkstra's algorithm, firstly, every shortest path/time will be inf.

+ Self.cnt[i][j] at first will be all the shortest paths going from vertice i to vertice j.

+ Self.trace[i][j] at first will be equal to -1 which means that the shortest path going from I to j is none (at first).

- Now, let’s see how I read the graph2:



```
for routeVarObject in tqdm(varQuery.routeVarList):
    averageSpeed = routeVarObject.Distance / (routeVarObject.RunningTime * 60)
    routeId = routeVarObject.RouteId
    routeVarId = routeVarObject.RouteVarId

    stopObject = stopQuery.searchByABC(RouteId = str(routeId), RouteVarId = str(routeVarId))
    pathObject = pathQuery.searchByABC(RouteId = str(routeId), RouteVarId = str(routeVarId))[0]

    stops = [[stop.StopId, stop.Lat, stop.Lng] for stop in stopObject]
    coordinates = [[x, y] for x, y in zip(pathObject.lat, pathObject.lng)]

    startStopId = stops[0][0]
    stops = stops[1:]
```

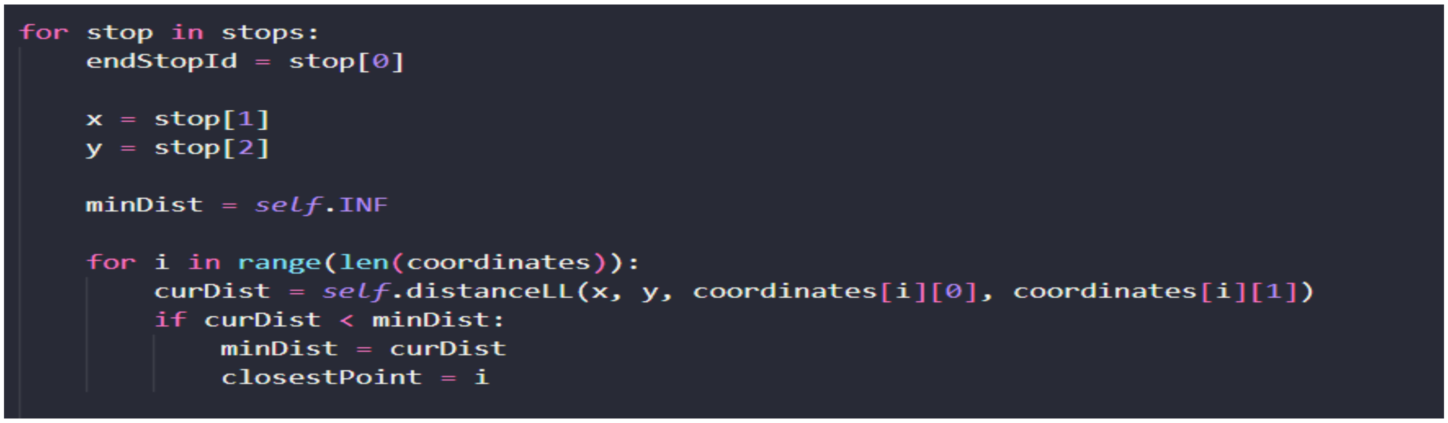
+ We will traverse through the routeVarList in file vars.json and then query the corresponding stopObject and pathObject based on routeId and routeVarId.

+ Then we will make a list of stops including information: StopId, StopLat, StopLng.

+ Similarly with the coordinates in the corresponding path object.

+ Notice that we will make the *startStopId* = *stops[0][0]* and then make *the stops* = *stops[1:]* (exclude the first stop).

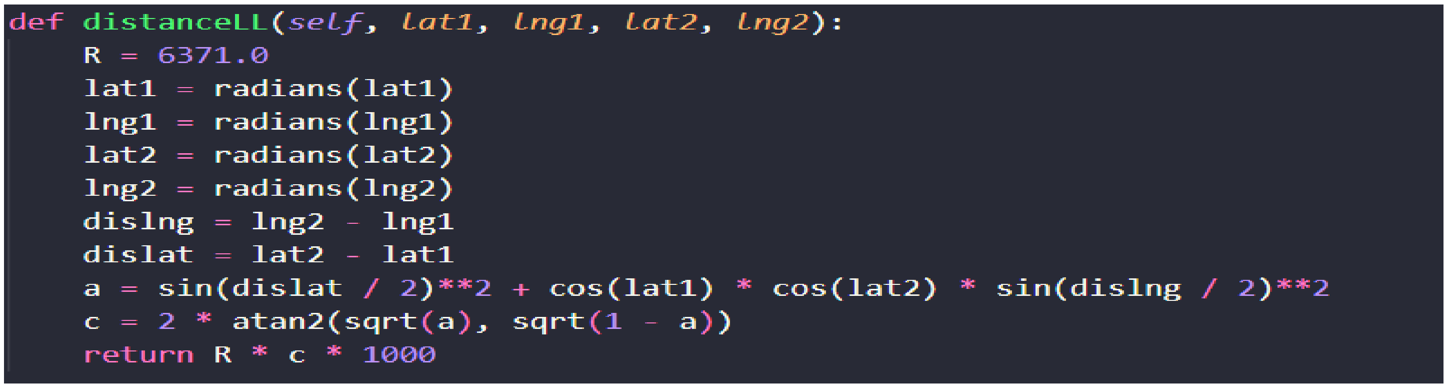
- Then, we will traverse the stop in stops:



```
for stop in stops:
    endStopId = stop[0]

    x = stop[1]
    y = stop[2]

    minDist = self.INF

    for i in range(len(coordinates)):
        curDist = self.distanceLL(x, y, coordinates[i][0], coordinates[i][1])
        if curDist < minDist:
            minDist = curDist
            closestPoint = i
```

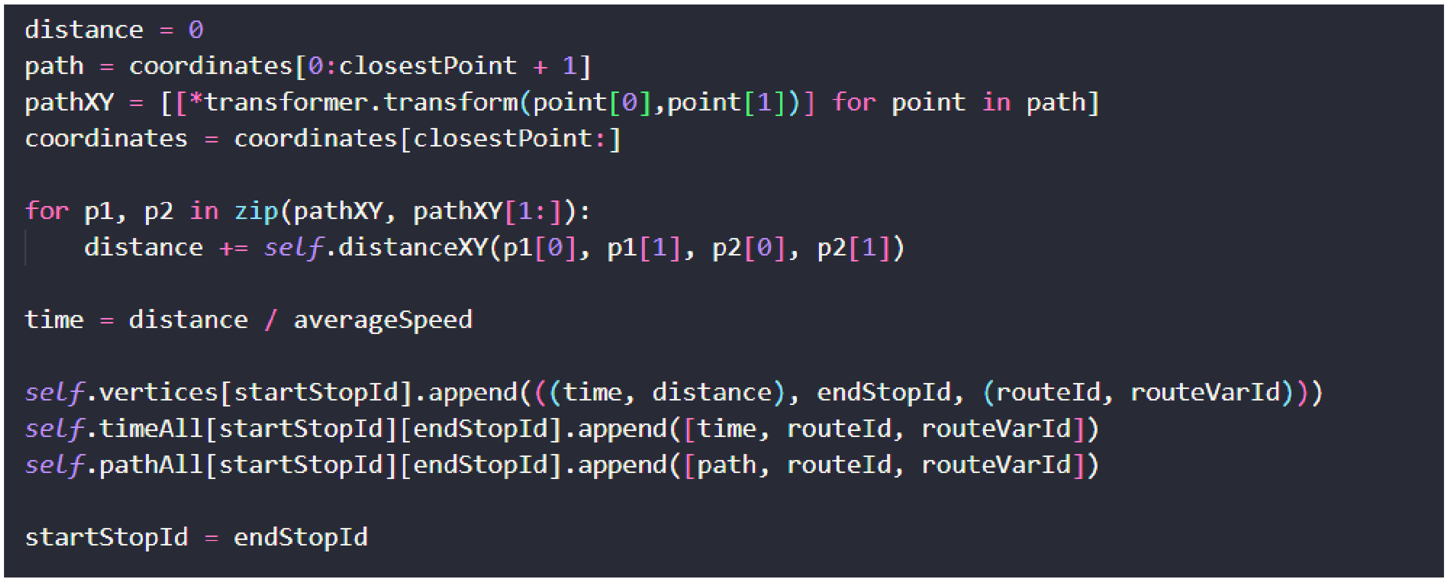
+ As for the method I mentioned above in the *“Approach”*. We will find the most similar point in paths with the StopLat and StopLng. We will do this by finding the minimum distance as the codes show above.

+ Notes that the self.distanceLL will count the distance in terms of lat and lng:



```
def distanceLL(self, lat1, lng1, lat2, lng2):
    R = 6371.0
    lat1 = radians(lat1)
    lng1 = radians(lng1)
    lat2 = radians(lat2)
    lng2 = radians(lng2)
    dislng = lng2 - lng1
    dislat = lat2 - lat1
    a = sin(dislat / 2)**2 + cos(lat1) * cos(lat2) * sin(dislng / 2)**2
    c = 2 * atan2(sqrt(a), sqrt(1 - a))
    return R * c * 1000
```

- After we find the closest one, we will add it to our graph:



```
distance = 0
path = coordinates[0:closestPoint + 1]
pathXY = [[*transformer.transform(point[0],point[1])] for point in path]
coordinates = coordinates[closestPoint:]

for p1, p2 in zip(pathXY, pathXY[1:]):
    distance += self.distanceXY(p1[0], p1[1], p2[0], p2[1])

time = distance / averageSpeed

self.vertices[startStopId].append(((time, distance), endStopId, (routeId, routeVarId)))
self.timeAll[startStopId][endStopId].append([time, routeId, routeVarId])
self.pathAll[startStopId][endStopId].append([path, routeId, routeVarId])

startStopId = endStopId
```

+ path will be all of the lat-lng from the last StopCoordinates to this new Stop. Similarly with pathXY but in terms of x and y.

+ The loop is to find the distance between two stops and then find the time.

+ Then we will push that information to the vertices, timeAll and pathAll as the codes above.

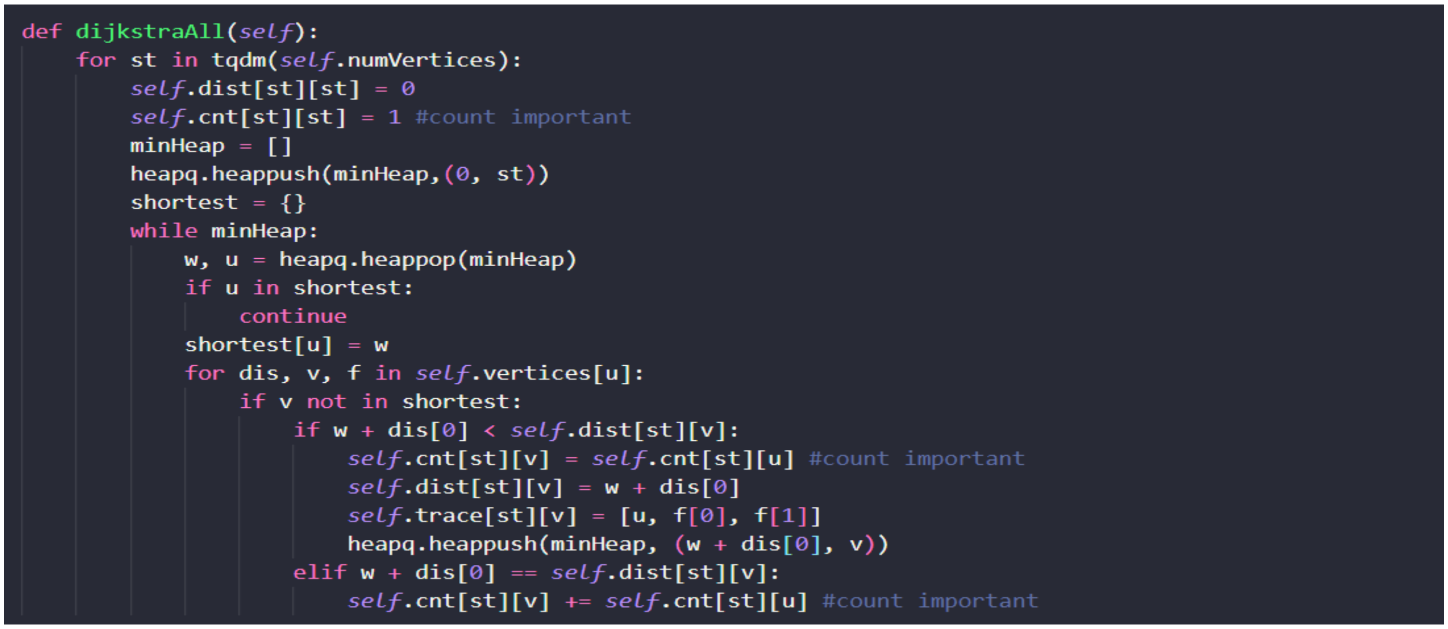
+ Notes that after implementation, we have to make the startStopId = endStopId as I mentioned in the approach, we will exclude all the points in paths before the newStop which has been added to our graph.

→ We can see that the implementation is quite clean and not too long to do. In this way, we can read our graph successfully and make it clear when we push data to the web to see the paths of the bus in Ho Chi Minh.

Notes: *tqdm is used to see the progress bar while handling the loop, in the user interface, I will explain more.*

4) Dijkstra's algorithm all pairs

- As I mentioned in the approach about the complexity, in this explanation, I will dive deeper into the codes and how we can save or trace the path after implementing Dijkstra's algorithm!



```
def dijkstraAll(self):
    for st in tqdm(self.numVertices):
        self.dist[st][st] = 0
        self.cnt[st][st] = 1 #count important
        minHeap = []
        heapq.heappush(minHeap,(0, st))
        shortest = {}
        while minHeap:
            w, u = heapq.heappop(minHeap)
            if u in shortest:
                continue
            shortest[u] = w
            for dis, v, f in self.vertices[u]:
                if v not in shortest:
                    if w + dis[0] < self.dist[st][v]:
                        self.cnt[st][v] = self.cnt[st][u] #count important
                        self.dist[st][v] = w + dis[0]
                        self.trace[st][v] = [u, f[0], f[1]]
                        heapq.heappush(minHeap, (w + dis[0], v))
                    elif w + dis[0] == self.dist[st][v]:
                        self.cnt[st][v] += self.cnt[st][u] #count important
```

- As this is Dijkstra for all pairs, so we have to loop for st in self.numVertices (distinct stops).

- First, the dist (time) from st to st is 0 (obviously).

**Notes: For the counting importance, I will explain later on.*

- Then we will create a priority queue and do the algorithm. Notice that the shortest is used to save the stop that we have already gone through so that we don't have to go again.

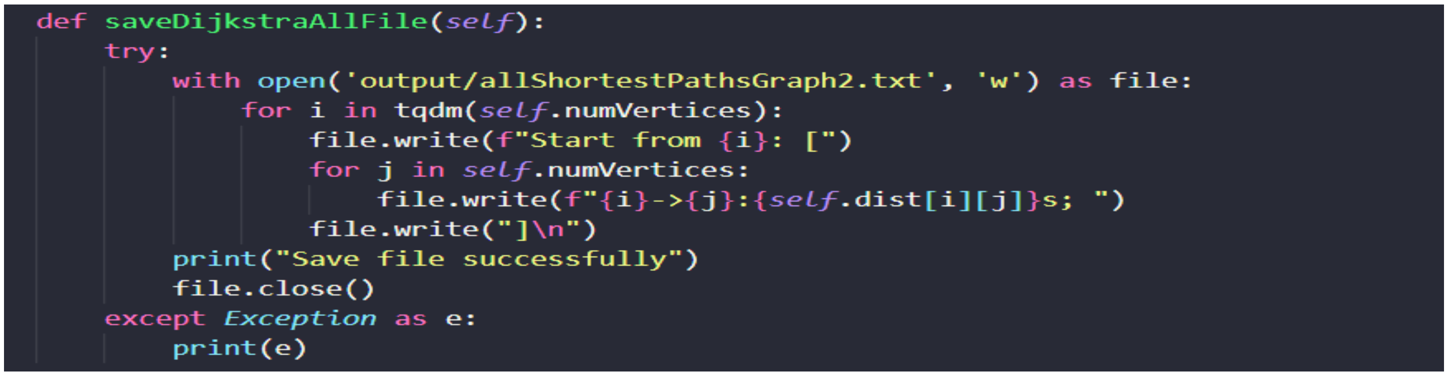
- For the looping, dis is the (time, distance); v is the to_stop_id (from u) and f is the (RouteId, RouteVarId) in self.vertices I have mentioned above.

- If the w + dis[0] is less than the path from start to u, we will make that the shortest path and then save it to the self.dist and self.trace as the codes above.

- Finally, we push the new data to the priority queue and continue doing like above until the queue is empty.

- That is the implementation of Dijkstra's algorithm for all pairs. Notice that this is optimized thanks to the method of reading data into our graph.

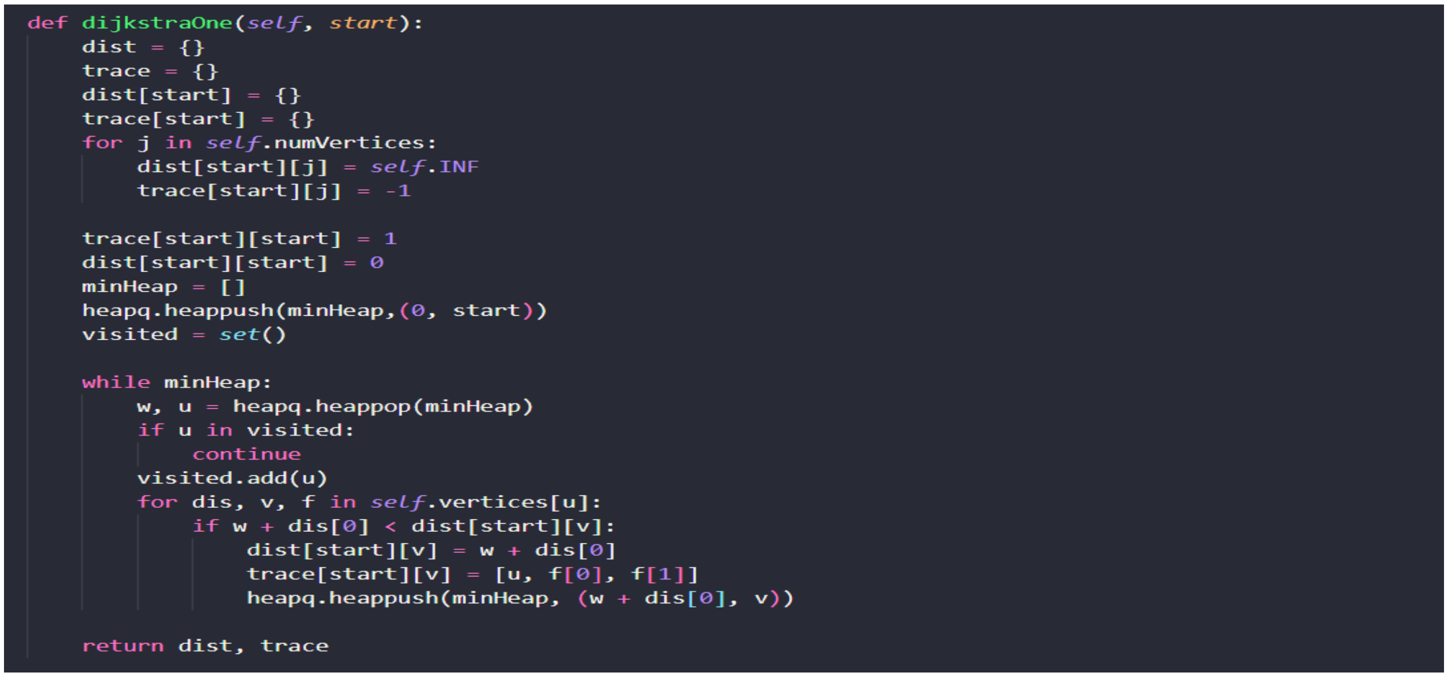
- Finally, we will save all shortest paths to a normal text file thanks to this function:



```python
def saveDijkstraAllFile(self):
    try:
        with open('output/allShortestPathsGraph2.txt', 'w') as file:
            for i in tqdm(self.numVertices):
                file.write(f"Start from {i}: [")
                for j in self.numVertices:
                    file.write(f"{i}->{j}:{self.dist[i][j]}s; ")
                file.write("]\n")
        print("Save file successfully")
        file.close()
    except Exception as e:
        print(e)
```

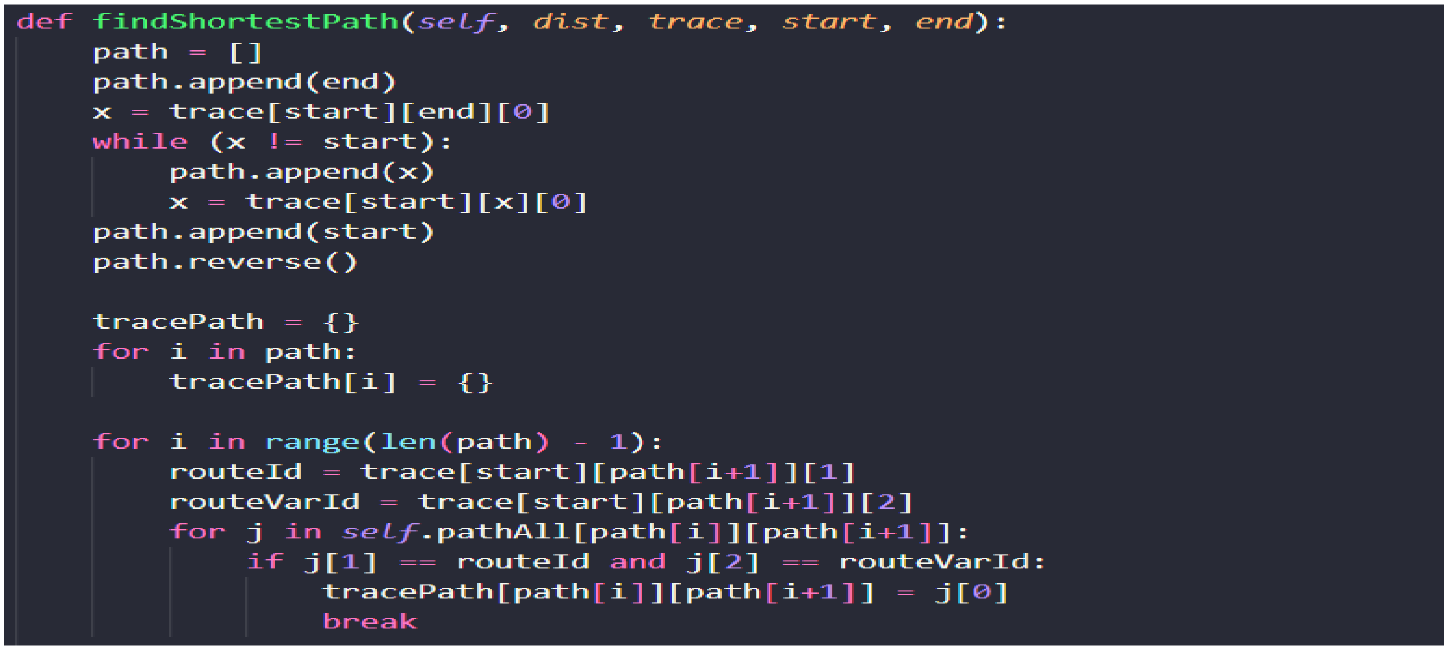
5) Dijkstra's algorithm for one stop and then save it to files

- Similar to the implementation of Dijkstra's algorithm for all pairs, but this time we will do with one stop and also handle the reading one into a JSON file and a normal text file. We will return the dist and trace of this stop.



```python
def dijkstraOne(self, start):
    dist = {}
    trace = {}
    dist[start] = {}
    trace[start] = {}
    for j in self.numVertices:
        dist[start][j] = self.INF
        trace[start][j] = -1

    trace[start][start] = 1
    dist[start][start] = 0
    minHeap = []
    heapq.heappush(minHeap,(0, start))
    visited = set()

    while minHeap:
        w, u = heapq.heappop(minHeap)
        if u in visited:
            continue
        visited.add(u)
        for dis, v, f in self.vertices[u]:
            if w + dis[0] < dist[start][v]:
                dist[start][v] = w + dis[0]
                trace[start][v] = [u, f[0], f[1]]
                heapq.heappush(minHeap, (w + dis[0], v))

    return dist, trace
```

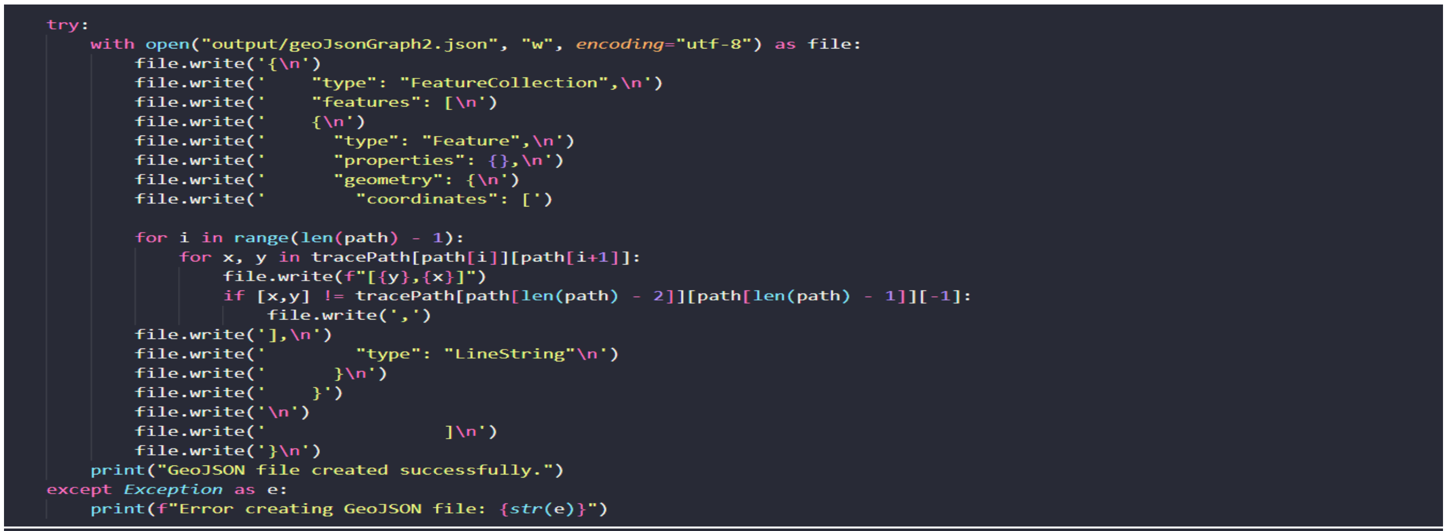
- After Dijkstra for one-stop, we can trace back the shortest path from that stop to any other stops easily through the dictionary trace that it returns.



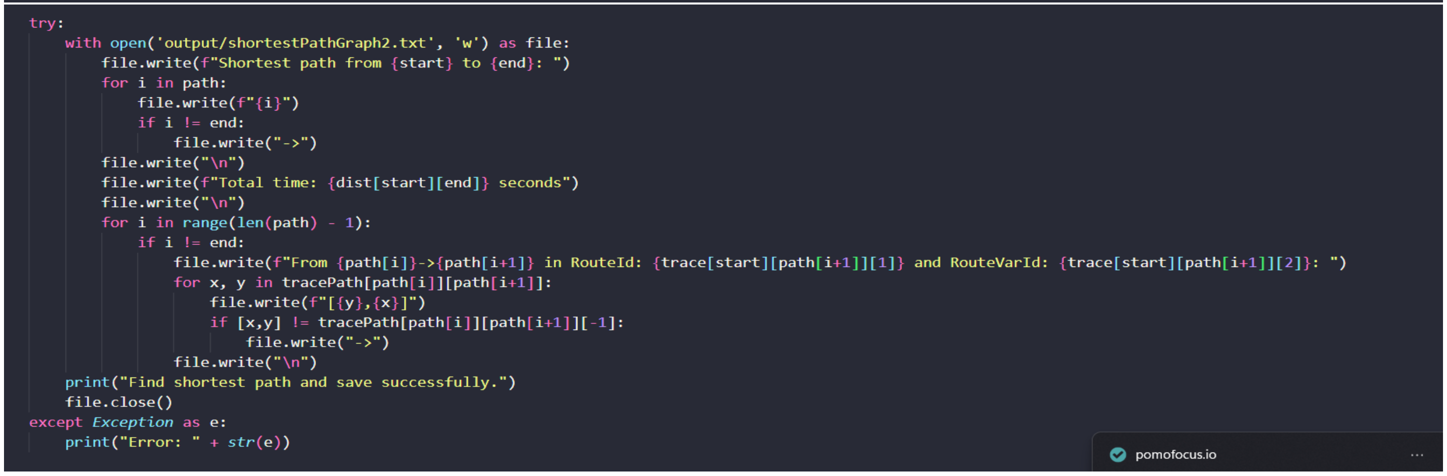
```python
def findShortestPath(self, dist, trace, start, end):
    path = []
    path.append(end)
    x = trace[start][end][0]
    while (x != start):
        path.append(x)
        x = trace[start][x][0]
    path.append(start)
    path.reverse()

    tracePath = {}
    for i in path:
        tracePath[i] = {}

    for i in range(len(path) - 1):
        routeId = trace[start][path[i+1]][1]
        routeVarId = trace[start][path[i+1]][2]
        for j in self.pathAll[path[i]][path[i+1]]:
            if j[1] == routeId and j[2] == routeVarId:
                tracePath[path[i]][path[i+1]] = j[0]
                break
```

- The loop before is to find the corresponding path in the correct routeId and routeVarId from start_stop_id to end_stop_id.

- Finally, our job is to save the paths to the JSON file and text file:



```python
try:
    with open("output/geoJsonGraph2.json", "w", encoding="utf-8") as file:
        file.write('{\n')
        file.write('    "type": "FeatureCollection",\n')
        file.write('    "features": [\n')
        file.write('    {\n')
        file.write('      "type": "Feature",\n')
        file.write('      "properties": {},\n')
        file.write('      "geometry": {\n')
        file.write('        "coordinates": [')

        for i in range(len(path) - 1):
            for x, y in tracePath[path[i]][path[i+1]]:
                file.write(f"[{y},{x}]")
                if [x,y] != tracePath[path[len(path) - 2]][path[len(path) - 1]][-1]:
                    file.write(',')
        file.write('],\n')
        file.write('        "type": "LineString"\n')
        file.write('      }\n')
        file.write('    }')
        file.write('\n')
        file.write('                  ]\n')
        file.write('}\n')
    print("GeoJSON file created successfully.")
except Exception as e:
    print(f"Error creating GeoJSON file: {str(e)}")
```



```python
try:
    with open('output/shortestPathGraph2.txt', 'w') as file:
        file.write(f"Shortest path from {start} to {end}: ")
        for i in path:
            file.write(f"{i}")
            if i != end:
                file.write("->")
        file.write("\n")
        file.write(f"Total time: {dist[start][end]} seconds")
        file.write("\n")
        for i in range(len(path) - 1):
            if i != end:
                file.write(f"From {path[i]}->{path[i+1]} in RouteId: {trace[start][path[i+1]][1]} and RouteVarId: {trace[start][path[i+1]][2]}: ")
                for x, y in tracePath[path[i]][path[i+1]]:
                    file.write(f"[{y},{x}]")
                    if [x,y] != tracePath[path[i]][path[i+1]][-1]:
                        file.write("->")
                file.write("\n")
    print("Find shortest path and save successfully.")
    file.close()
except Exception as e:
    print("Error: " + str(e))
```

6) Counting the importance of a stop

- As I mentioned in the approach, we have to count the self.cnt[u][v] first (which is the shortest path from u to v). We will do this while handling Dijkstra's algorithm for all pairs.

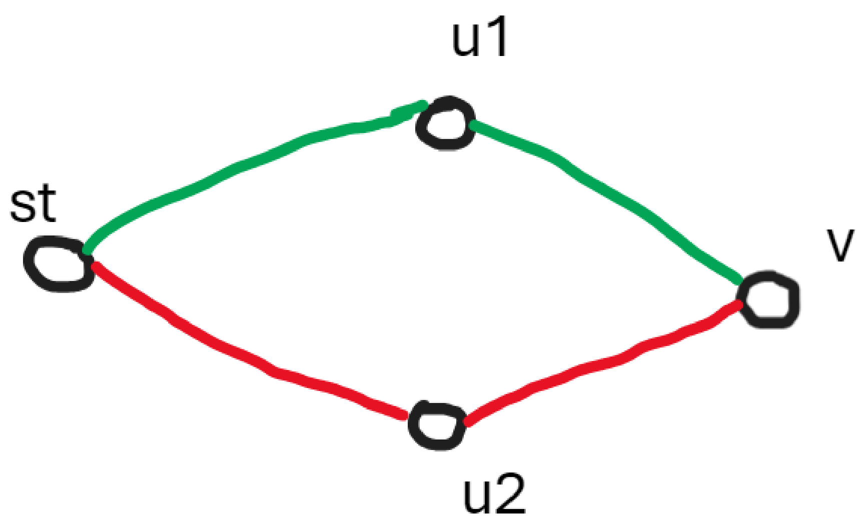
- Easy to see that for a stop, the shortest path from it to itself is 1, so: ***self.cnt[st][st] = 1***

- Then for when we update the shortest path from st to v, *self.cnt[st][v]* will be equal to *self.cnt[st][u]* as if the shortest path from st to v is to go through u, then all shortest paths from st to v will be equal to all shortest paths from st to u.

- However, there is a situation when there are two shortest paths from start to v going through u, then we have to add to the *self.cnt[st][v]:*

```
elif w + dis[0] == self.dist[st][v]:
    self.cnt[st][v] += self.cnt[st][u]
```

- Example:



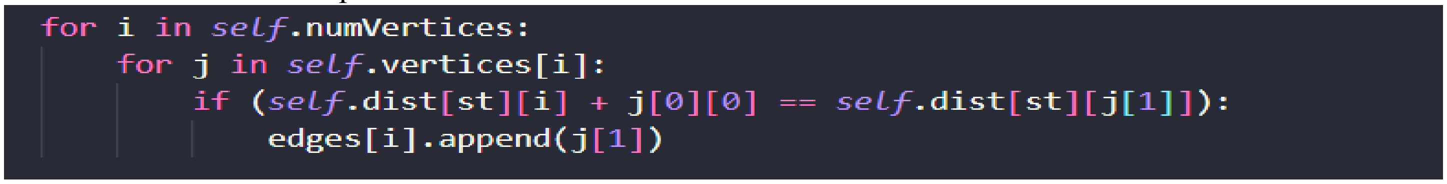
self.cnt[st][v] = self.cnt[st][u1] + self.cnt[st][u2]

- After we count all of the self.cnt[st][v]. We will find out the shortest path from v to other vertices with start_stop_id is u. We will exclude all not shortest paths then sort topologically by dfs(u) then multiply with the self.cnt[st][v]:

*self.cnt[st][v] *= dp[v];*

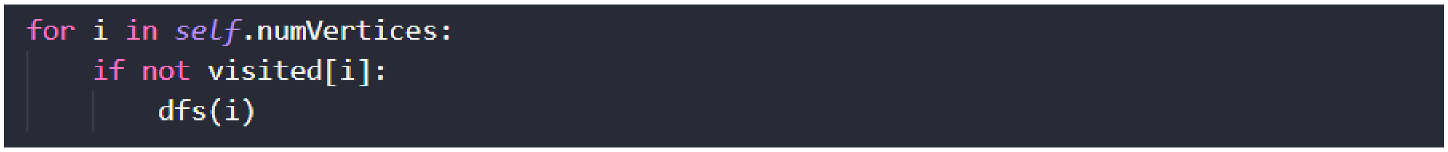
- This is the implementation:

+ Exclude all not shortest paths:

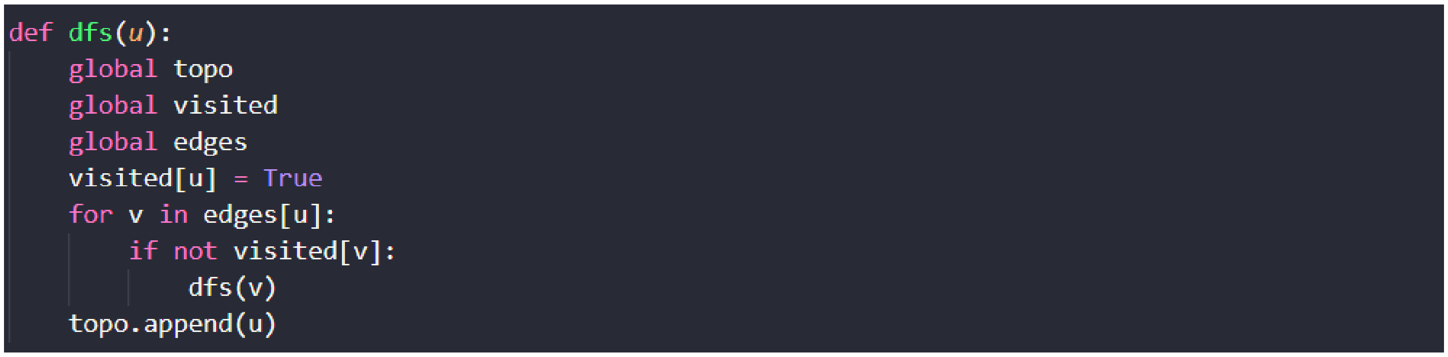


```
for i in self.numVertices:
    for j in self.vertices[i]:
        if (self.dist[st][i] + j[0][0] == self.dist[st][j[1]]):
            edges[i].append(j[1])
```

+ Sort topologically:

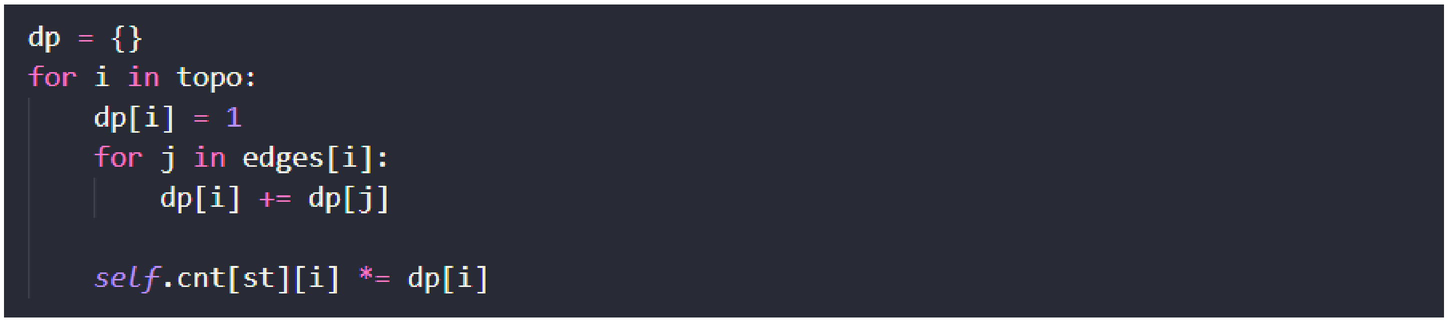


```
for i in self.numVertices:
    if not visited[i]:
        dfs(i)
```



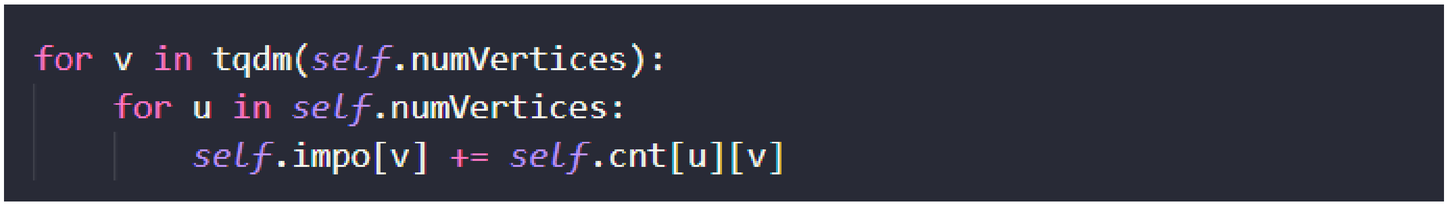
```
def dfs(u):
    global topo
    global visited
    global edges
    visited[u] = True
    for v in edges[u]:
        if not visited[v]:
            dfs(v)
    topo.append(u)
```

+ dp to count the final self.cnt:



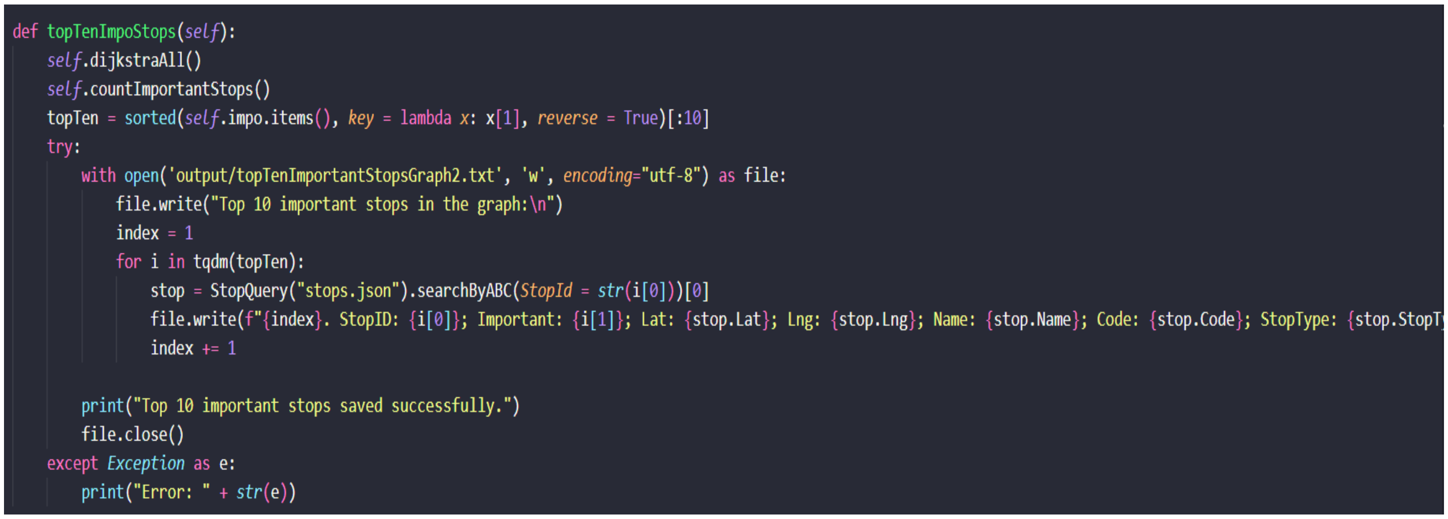
```
dp = {}
for i in topo:
    dp[i] = 1
    for j in edges[i]:
        dp[i] += dp[j]

    self.cnt[st][i] *= dp[i]
```

+ After all of the above implementation, our job is to add all of the self.cnt[u][v] to find the importance of v.



```
for v in tqdm(self.numVertices):
    for u in self.numVertices:
        self.impo[v] += self.cnt[u][v]
```

- Finally, our job is to sort the importance list and then save the top 10 most important stops in a file with their data:



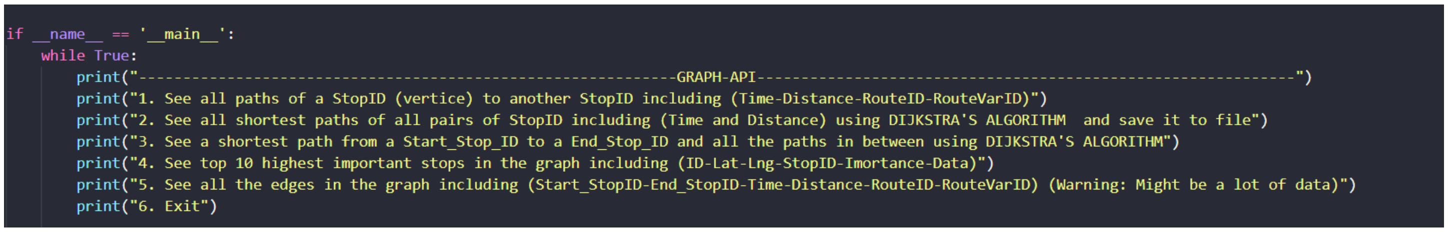
```
def topTenImpoStops(self):
    self.dijkstraAll()
    self.countImportantStops()
    topTen = sorted(self.impo.items(), key = lambda x: x[1], reverse = True)[:10]
    try:
        with open('output/topTenImportantStopsGraph2.txt', 'w', encoding="utf-8") as file:
            file.write("Top 10 important stops in the graph:\n")
            index = 1
            for i in tqdm(topTen):
                stop = StopQuery("stops.json").searchByABC(StopId = str(i[0]))[0]
                file.write(f"{index}. StopID: {i[0]}; Important: {i[1]}; Lat: {stop.Lat}; Lng: {stop.Lng}; Name: {stop.Name}; Code: {stop.Code}; StopType: {stop.StopT
                index += 1

        print("Top 10 important stops saved successfully.")
        file.close()
    except Exception as e:
        print("Error: " + str(e))
```

V) User interface in main.py

1) Codes

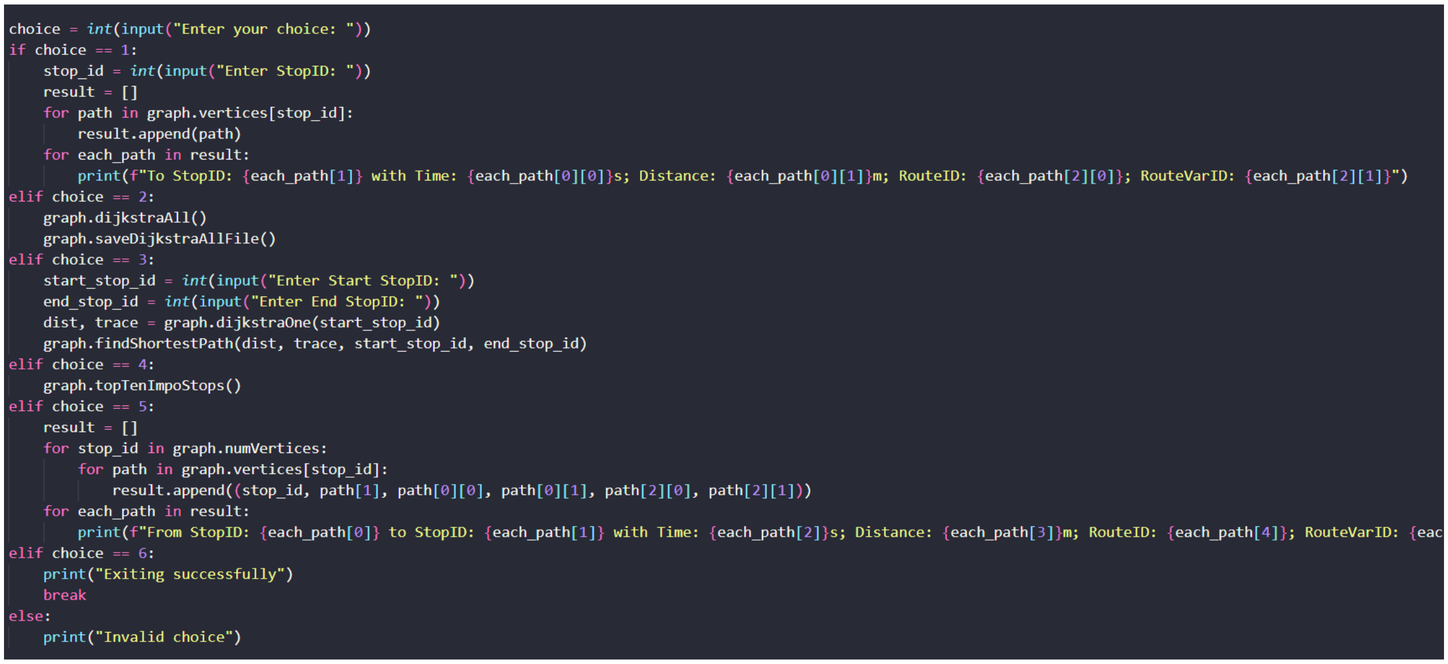
- First, we have to load our graph: *graph = Graph2("vars.json", "stops.json", "paths.json")*

- The user interface:



```python
if __name__ == '__main__':
    while True:
        print("------------------------------------------------------------GRAPH-API------------------------------------------------------------")
        print("1. See all paths of a StopID (vertice) to another StopID including (Time-Distance-RouteID-RouteVarID)")
        print("2. See all shortest paths of all pairs of StopID including (Time and Distance) using DIJKSTRA'S ALGORITHM  and save it to file")
        print("3. See a shortest path from a Start_Stop_ID to a End_Stop_ID and all the paths in between using DIJKSTRA'S ALGORITHM")
        print("4. See top 10 highest important stops in the graph including (ID-Lat-Lng-StopID-Imortance-Data)")
        print("5. See all the edges in the graph including (Start_StopID-End_StopID-Time-Distance-RouteID-RouteVarID) (Warning: Might be a lot of data)")
        print("6. Exit")
```

- Handling queries:



```python
choice = int(input("Enter your choice: "))
if choice == 1:
    stop_id = int(input("Enter StopID: "))
    result = []
    for path in graph.vertices[stop_id]:
        result.append(path)
    for each_path in result:
        print(f"To StopID: {each_path[1]} with Time: {each_path[0][0]}s; Distance: {each_path[0][1]}m; RouteID: {each_path[2][0]}; RouteVarID: {each_path[2][1]}")
elif choice == 2:
    graph.dijkstraAll()
    graph.saveDijkstraAllFile()
elif choice == 3:
    start_stop_id = int(input("Enter Start StopID: "))
    end_stop_id = int(input("Enter End StopID: "))
    dist, trace = graph.dijkstraOne(start_stop_id)
    graph.findShortestPath(dist, trace, start_stop_id, end_stop_id)
elif choice == 4:
    graph.topTenImpoStops()
elif choice == 5:
    result = []
    for stop_id in graph.numVertices:
        for path in graph.vertices[stop_id]:
            result.append((stop_id, path[1], path[0][0], path[0][1], path[2][0], path[2][1]))
    for each_path in result:
        print(f"From StopID: {each_path[0]} to StopID: {each_path[1]} with Time: {each_path[2]}s; Distance: {each_path[3]}m; RouteID: {each_path[4]}; RouteVarID: {eac
elif choice == 6:
    print("Exiting successfully")
    break
else:
    print("Invalid choice")
```

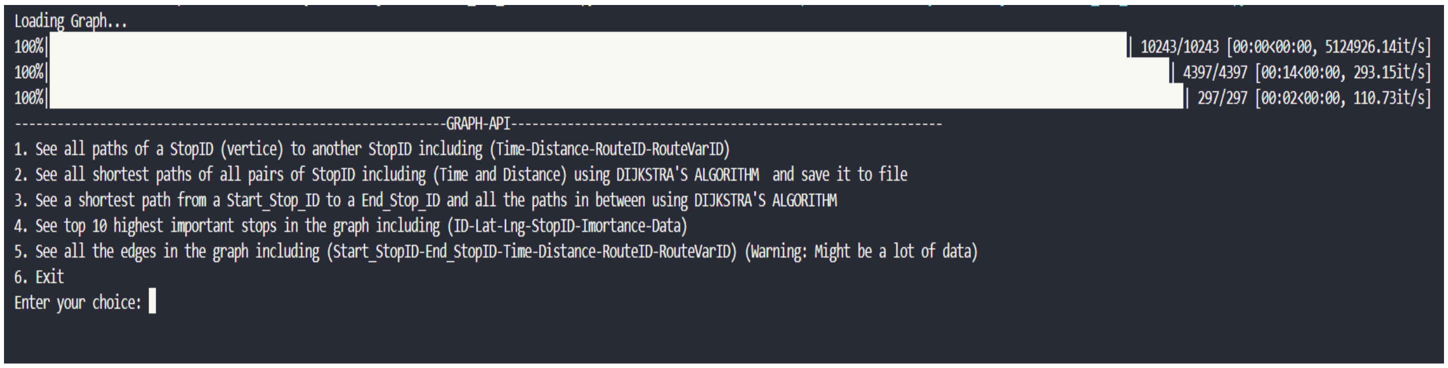
+ Notes that for the first task, we will show the graph.vertices, which are the edges of the graph.

+ For task 3, we don't have to Dijkstra for all, only one is enough.

+ Task 5 will print all of the edges of the graph, which may cause data overload as it's quite big.

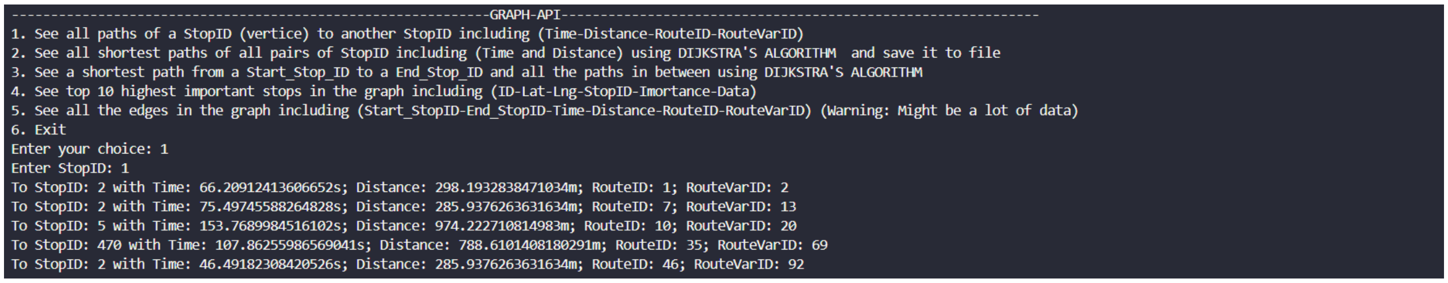
2) Examples of giving queries

+ Notice that we will have time to load our graph, the time will be shown in the terminal



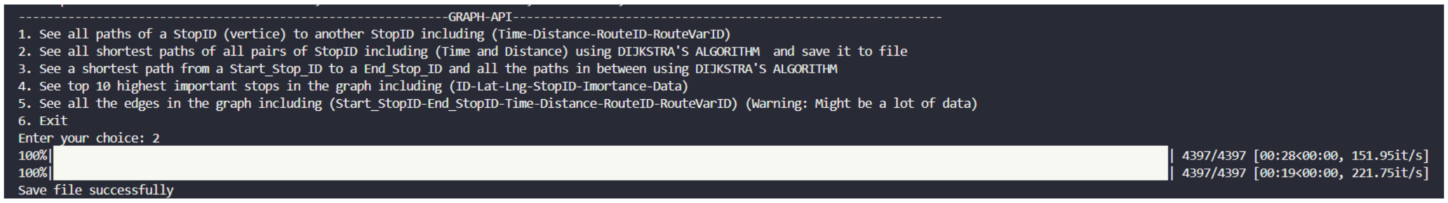
```
Loading Graph...
100%|██████████████████████████████████████████████████| 10243/10243 [00:00<00:00, 5124926.14it/s]
100%|██████████████████████████████████████████████████| 4397/4397 [00:14<00:00, 293.15it/s]
100%|██████████████████████████████████████████████████| 297/297 [00:02<00:00, 110.73it/s]
-----------------------------------------------------------GRAPH-API-----------------------------------------------------------
1. See all paths of a StopID (vertice) to another StopID including (Time-Distance-RouteID-RouteVarID)
2. See all shortest paths of all pairs of StopID including (Time and Distance) using DIJKSTRA'S ALGORITHM  and save it to file
3. See a shortest path from a Start_Stop_ID to a End_Stop_ID and all the paths in between using DIJKSTRA'S ALGORITHM
4. See top 10 highest important stops in the graph including (ID-Lat-Lng-StopID-Imortance-Data)
5. See all the edges in the graph including (Start_StopID-End_StopID-Time-Distance-RouteID-RouteVarID) (Warning: Might be a lot of data)
6. Exit
Enter your choice:
```

+ *Task1:*

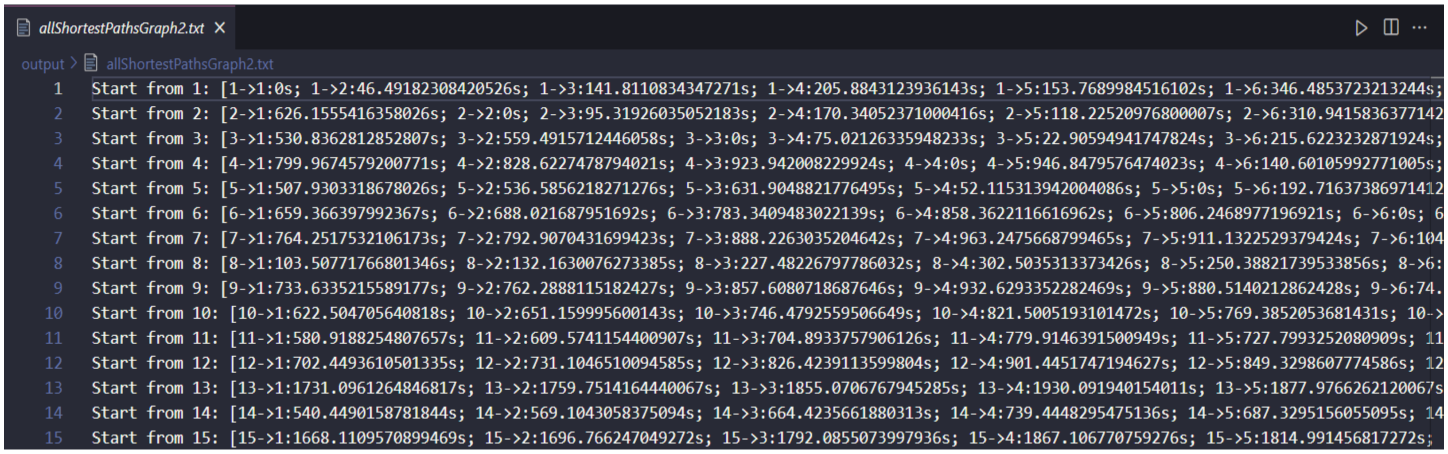


```
-------------------------------------------------------------GRAPH-API-------------------------------------------------------------
1. See all paths of a StopID (vertice) to another StopID including (Time-Distance-RouteID-RouteVarID)
2. See all shortest paths of all pairs of StopID including (Time and Distance) using DIJKSTRA'S ALGORITHM  and save it to file
3. See a shortest path from a Start_Stop_ID to a End_Stop_ID and all the paths in between using DIJKSTRA'S ALGORITHM
4. See top 10 highest important stops in the graph including (ID-Lat-Lng-StopID-Imortance-Data)
5. See all the edges in the graph including (Start_StopID-End_StopID-Time-Distance-RouteID-RouteVarID) (Warning: Might be a lot of data)
6. Exit
Enter your choice: 1
Enter StopID: 1
To StopID: 2 with Time: 66.20912413606652s; Distance: 298.1932838471034m; RouteID: 1; RouteVarID: 2
To StopID: 2 with Time: 75.49745588264828s; Distance: 285.9376263631634m; RouteID: 7; RouteVarID: 13
To StopID: 5 with Time: 153.7689984516102s; Distance: 974.222710814983m; RouteID: 10; RouteVarID: 20
To StopID: 470 with Time: 107.86255986569041s; Distance: 788.6101408180291m; RouteID: 35; RouteVarID: 69
To StopID: 2 with Time: 46.49182308420526s; Distance: 285.9376263631634m; RouteID: 46; RouteVarID: 92
```

+ *Task2:*



```
-------------------------------------------------------------GRAPH-API-------------------------------------------------------------
1. See all paths of a StopID (vertice) to another StopID including (Time-Distance-RouteID-RouteVarID)
2. See all shortest paths of all pairs of StopID including (Time and Distance) using DIJKSTRA'S ALGORITHM  and save it to file
3. See a shortest path from a Start_Stop_ID to a End_Stop_ID and all the paths in between using DIJKSTRA'S ALGORITHM
4. See top 10 highest important stops in the graph including (ID-Lat-Lng-StopID-Imortance-Data)
5. See all the edges in the graph including (Start_StopID-End_StopID-Time-Distance-RouteID-RouteVarID) (Warning: Might be a lot of data)
6. Exit
Enter your choice: 2
100%|████████████████████████████████████████████████████████████████████| 4397/4397 [00:28<00:00, 151.95it/s]
100%|████████████████████████████████████████████████████████████████████| 4397/4397 [00:19<00:00, 221.75it/s]
Save file successfully
```

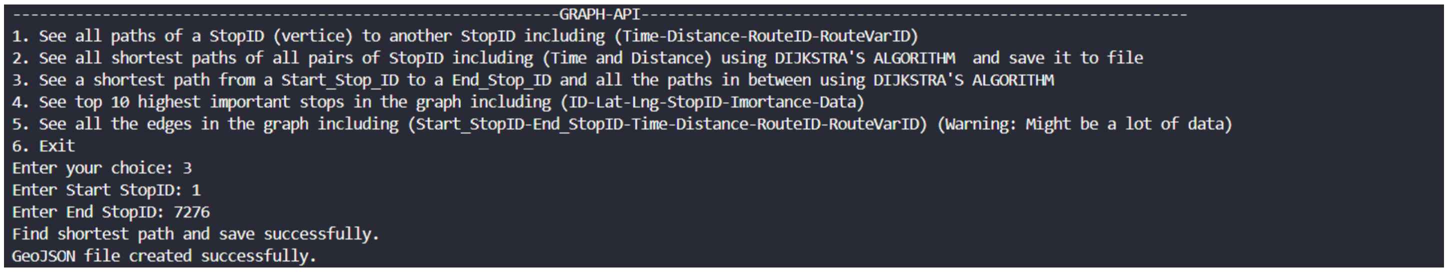


allShortestPathsGraph2.txt

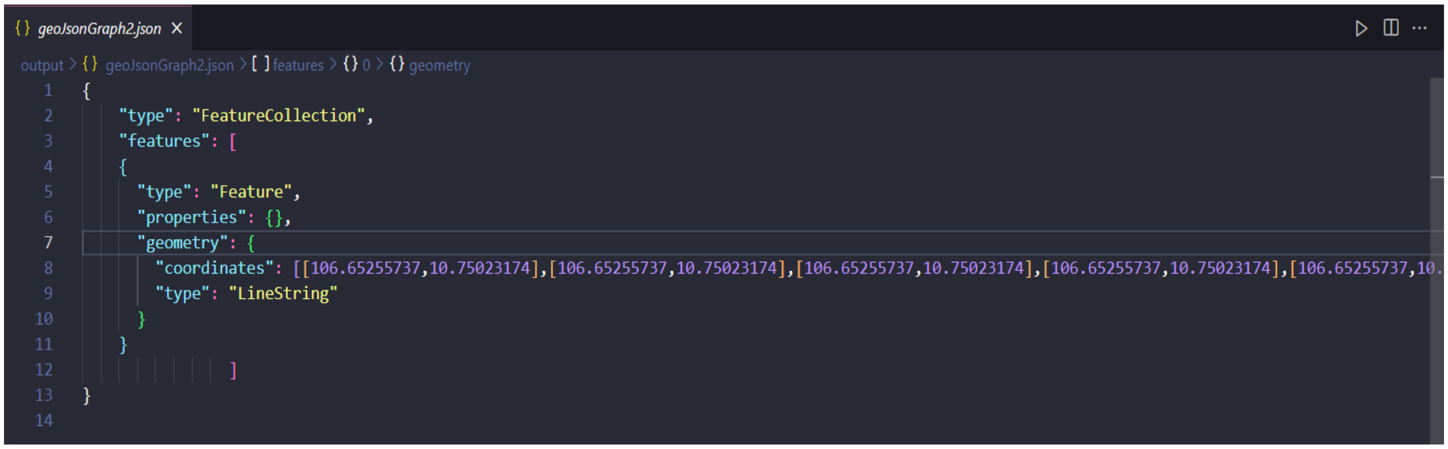
output > allShortestPathsGraph2.txt

```
Start from 1: [1->1:0s; 1->2:46.49182308420526s; 1->3:141.8110834347271s; 1->4:205.8843123936143s; 1->5:153.7689984516102s; 1->6:346.4853723213244s;
Start from 2: [2->1:626.1555416358026s; 2->2:0s; 2->3:95.31926035052183s; 2->4:170.34052371000416s; 2->5:118.22520976800007s; 2->6:310.941583637714
Start from 3: [3->1:530.8362812852807s; 3->2:559.4915712446058s; 3->3:0s; 3->4:75.02126335948233s; 3->5:22.90594941747824s; 3->6:215.622323287192 4s;
Start from 4: [4->1:799.9674579200771s; 4->2:828.6227478794021s; 4->3:923.942008229924s; 4->4:0s; 4->5:946.8479576474023s; 4->6:140.6010599277100 5s;
Start from 5: [5->1:507.9303318678026s; 5->2:536.5856218271276s; 5->3:631.9048821776495s; 5->4:52.115313942004086s; 5->5:0s; 5->6:192.7163738697141
Start from 6: [6->1:659.366397992367s; 6->2:688.021687951692s; 6->3:783.340948302213 9s; 6->4:858.3622116616962s; 6->5:806.2468977196921s; 6->6:0s; 6
Start from 7: [7->1:764.2517532106173s; 7->2:792.9070431699423s; 7->3:888.2263035204642s; 7->4:963.2475668799465s; 7->5:911.1322529379424s; 7->6:104
Start from 8: [8->1:103.50771766801346s; 8->2:132.1630076273385s; 8->3:227.48226797786032s; 8->4:302.5035313373426s; 8->5:250.38821739533856s; 8->6:
Start from 9: [9->1:733.6335215589177s; 9->2:762.2888115182427s; 9->3:857.6080718687646s; 9->4:932.6293352282469s; 9->5:880.5140212862428s; 9->6:74.
Start from 10: [10->1:622.504705640818s; 10->2:651.159995600143s; 10->3:746.4792559506649s; 10->4:821.5005193101472s; 10->5:769.3852053681431s; 10->
Start from 11: [11->1:580.9188254807657s; 11->2:609.5741154400907s; 11->3:704.8933757906126s; 11->4:779.9146391500949s; 11->5:727.7993252080909s; 11
Start from 12: [12->1:702.4493610501335s; 12->2:731.1046510094585s; 12->3:826.4239113599804s; 12->4:901.4451747194627s; 12->5:849.3298607774586s; 12
Start from 13: [13->1:1731.0961264846817s; 13->2:1759.7514164440067s; 13->3:1855.0706767945285s; 13->4:1930.091940154011s; 13->5:1877.976626212006 7s
Start from 14: [14->1:540.4490158781844s; 14->2:569.1043058375094s; 14->3:664.4235661880313s; 14->4:739.4448295475136s; 14->5:687.3295156055095s; 14
Start from 15: [15->1:1668.1109570899469s; 15->2:1696.766247049272s; 15->3:1792.0855073997936s; 15->4:1867.10677075927 6s; 15->5:1814.991456817272s;
```

+ *Task3:*



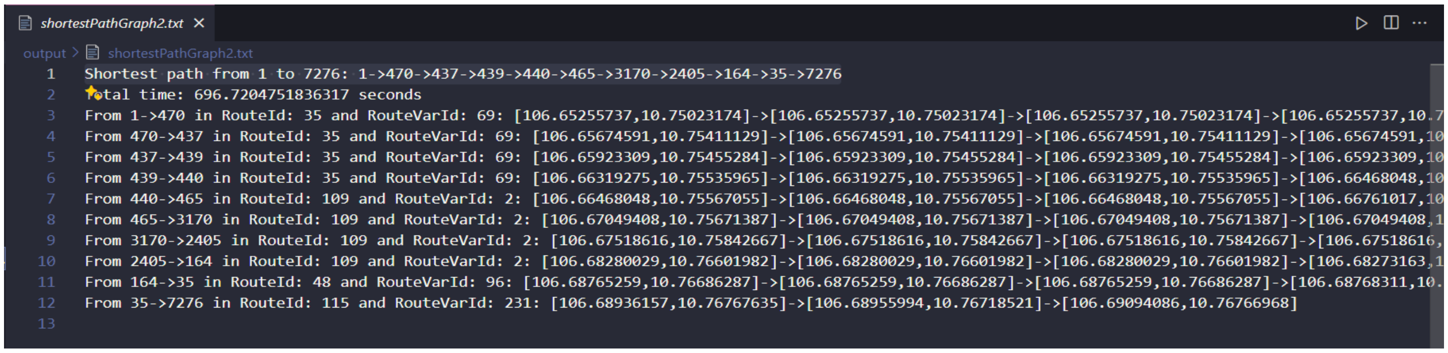
```
-------------------------------------------------------------GRAPH-API-------------------------------------------------------------
1. See all paths of a StopID (vertice) to another StopID including (Time-Distance-RouteID-RouteVarID)
2. See all shortest paths of all pairs of StopID including (Time and Distance) using DIJKSTRA'S ALGORITHM  and save it to file
3. See a shortest path from a Start_Stop_ID to a End_Stop_ID and all the paths in between using DIJKSTRA'S ALGORITHM
4. See top 10 highest important stops in the graph including (ID-Lat-Lng-StopID-Imortance-Data)
5. See all the edges in the graph including (Start_StopID-End_StopID-Time-Distance-RouteID-RouteVarID) (Warning: Might be a lot of data)
6. Exit
Enter your choice: 3
Enter Start StopID: 1
Enter End StopID: 7276
Find shortest path and save successfully.
GeoJSON file created successfully.
```



geoJsonGraph2.json

output > {} geoJsonGraph2.json > [ ] features > {} 0 > {} geometry

```
{
    "type": "FeatureCollection",
    "features": [
    {
      "type": "Feature",
      "properties": {},
      "geometry": {
        "coordinates": [[106.65255737,10.75023174],[106.65255737,10.75023174],[106.65255737,10.75023174],[106.65255737,10.75023174],[106.65255737,10.
        "type": "LineString"
      }
    }
                ]
}
```

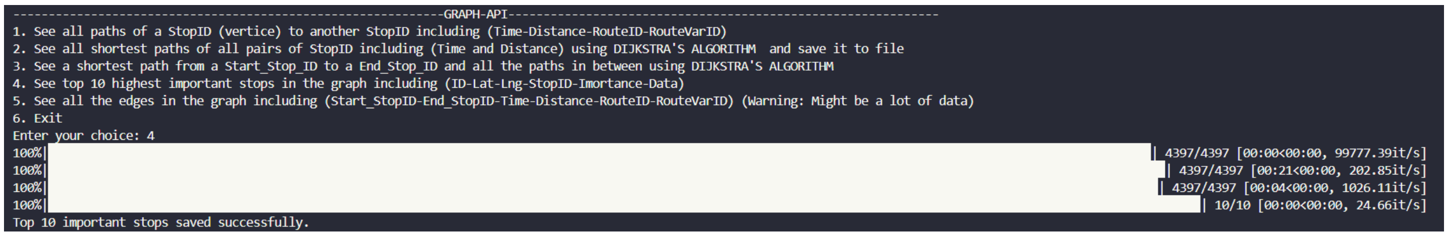


shortestPathGraph2.txt

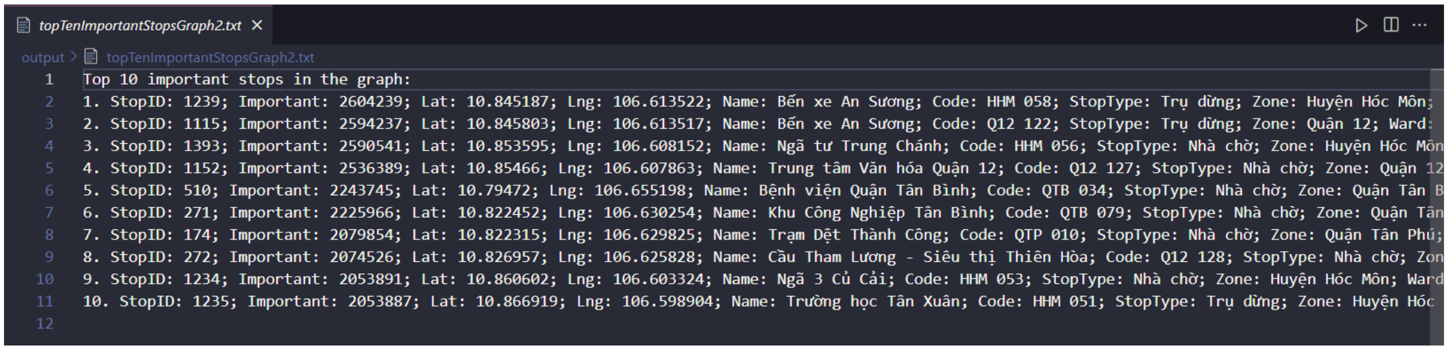
output > shortestPathGraph2.txt

```
Shortest path from 1 to 7276: 1->470->437->439->440->465->3170->2405->164->35->7276
Total time: 696.7204751836317 seconds
From 1->470 in RouteId: 35 and RouteVarId: 69: [106.65255737,10.75023174]->[106.65255737,10.75023174]->[106.65255737,10.75023174]->[106.65255737,10.7
From 470->437 in RouteId: 35 and RouteVarId: 69: [106.65674591,10.75411129]->[106.65674591,10.75411129]->[106.65674591,10.75411129]->[106.65674591,10
From 437->439 in RouteId: 35 and RouteVarId: 69: [106.65923309,10.75455284]->[106.65923309,10.75455284]->[106.65923309,10.75455284]->[106.65923309,10
From 439->440 in RouteId: 35 and RouteVarId: 69: [106.66319275,10.75535965]->[106.66319275,10.75535965]->[106.66319275,10.75535965]->[106.66468048,10
From 440->465 in RouteId: 109 and RouteVarId: 2: [106.66468048,10.75567055]->[106.66468048,10.75567055]->[106.66468048,10.75567055]->[106.66761017,10
From 465->3170 in RouteId: 109 and RouteVarId: 2: [106.67049408,10.75671387]->[106.67049408,10.75671387]->[106.67049408,10.75671387]->[106.67049408,1
From 3170->2405 in RouteId: 109 and RouteVarId: 2: [106.67518616,10.75842667]->[106.67518616,10.75842667]->[106.67518616,10.75842667]->[106.67518616,
From 2405->164 in RouteId: 109 and RouteVarId: 2: [106.68280029,10.76601982]->[106.68280029,10.76601982]->[106.68280029,10.76601982]->[106.68273163,1
From 164->35 in RouteId: 48 and RouteVarId: 96: [106.68765259,10.76686287]->[106.68765259,10.76686287]->[106.68765259,10.76686287]->[106.68768311,10.
From 35->7276 in RouteId: 115 and RouteVarId: 231: [106.68936157,10.76767635]->[106.68955994,10.76718521]->[106.69094086,10.76766968]
```

+ *Task4:*



```
-----------------------------------------------------------GRAPH-API-----------------------------------------------------------
1. See all paths of a StopID (vertice) to another StopID including (Time-Distance-RouteID-RouteVarID)
2. See all shortest paths of all pairs of StopID including (Time and Distance) using DIJKSTRA'S ALGORITHM  and save it to file
3. See a shortest path from a Start_Stop_ID to a End_Stop_ID and all the paths in between using DIJKSTRA'S ALGORITHM
4. See top 10 highest important stops in the graph including (ID-Lat-Lng-StopID-Imortance-Data)
5. See all the edges in the graph including (Start_StopID-End_StopID-Time-Distance-RouteID-RouteVarID) (Warning: Might be a lot of data)
6. Exit
Enter your choice: 4
100%|                                                                         | 4397/4397 [00:00<00:00, 99777.39it/s]
100%|                                                                           | 4397/4397 [00:21<00:00, 202.85it/s]
100%|                                                                          | 4397/4397 [00:04<00:00, 1026.11it/s]
100%|                                                                                  | 10/10 [00:00<00:00, 24.66it/s]
Top 10 important stops saved successfully.
```

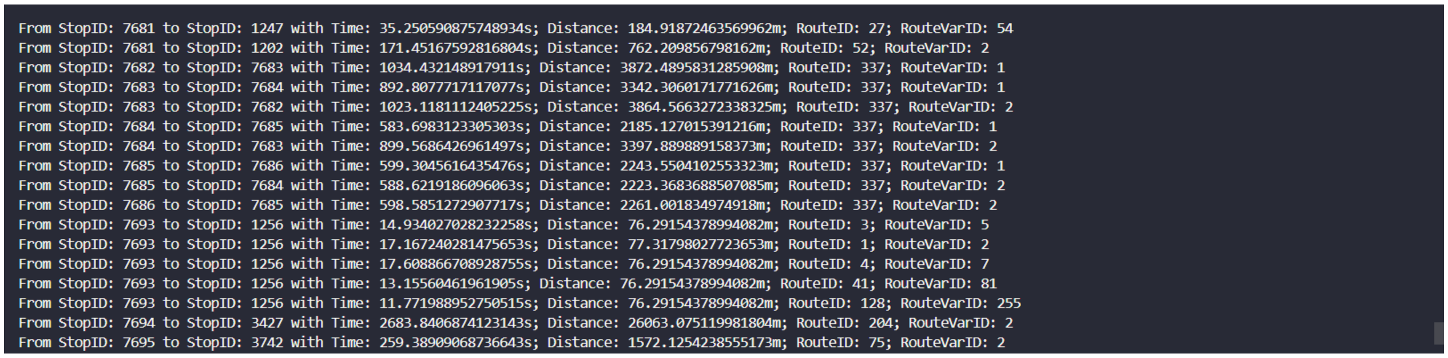


topTenImportantStopsGraph2.txt

output > topTenImportantStopsGraph2.txt

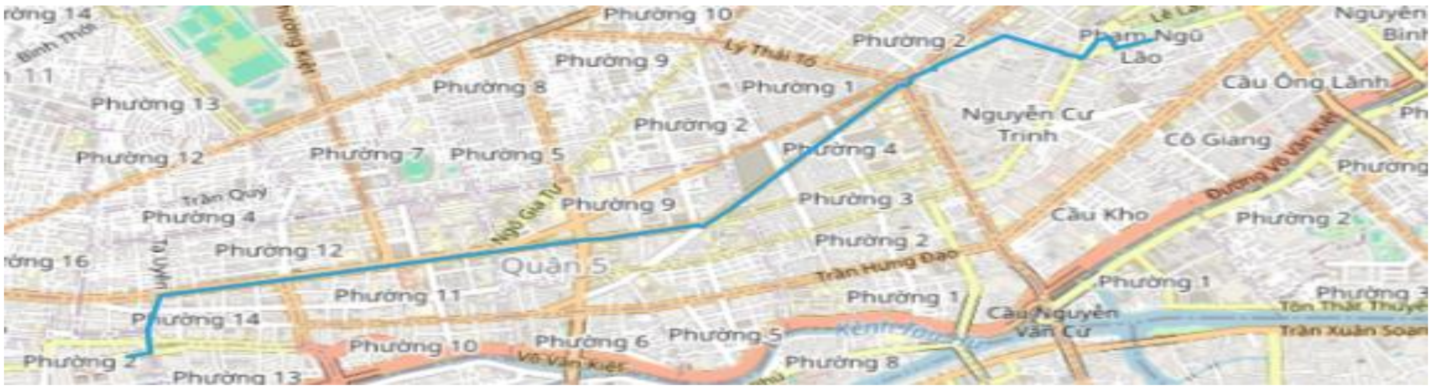
```
Top 10 important stops in the graph:
1. StopID: 1239; Important: 2604239; Lat: 10.845187; Lng: 106.613522; Name: Bến xe An Sương; Code: HHM 058; StopType: Trụ dừng; Zone: Huyện Hóc Môn;
2. StopID: 1115; Important: 2594237; Lat: 10.845803; Lng: 106.613517; Name: Bến xe An Sương; Code: Q12 122; StopType: Trụ dừng; Zone: Quận 12; Ward:
3. StopID: 1393; Important: 2590541; Lat: 10.853595; Lng: 106.608152; Name: Ngã tư Trung Chánh; Code: HHM 056; StopType: Nhà chờ; Zone: Huyện Hóc Môn
4. StopID: 1152; Important: 2536389; Lat: 10.85466; Lng: 106.607863; Name: Trung tâm Văn hóa Quận 12; Code: Q12 127; StopType: Nhà chờ; Zone: Quận 12
5. StopID: 510; Important: 2243745; Lat: 10.79472; Lng: 106.655198; Name: Bệnh viện Quận Tân Bình; Code: QTB 034; StopType: Nhà chờ; Zone: Quận Tân B
6. StopID: 271; Important: 2225966; Lat: 10.822452; Lng: 106.630254; Name: Khu Công Nghiệp Tân Bình; Code: QTB 079; StopType: Nhà chờ; Zone: Quận Tân
7. StopID: 174; Important: 2079854; Lat: 10.822315; Lng: 106.629825; Name: Trạm Dệt Thành Công; Code: QTP 010; StopType: Nhà chờ; Zone: Quận Tân Phú;
8. StopID: 272; Important: 2074526; Lat: 10.826957; Lng: 106.625828; Name: Cầu Tham Lương - Siêu thị Thiên Hòa; Code: Q12 128; StopType: Nhà chờ; Zon
9. StopID: 1234; Important: 2053891; Lat: 10.860602; Lng: 106.603324; Name: Ngã 3 Củ Cải; Code: HHM 053; StopType: Nhà chờ; Zone: Huyện Hóc Môn; Ward
10. StopID: 1235; Important: 2053887; Lat: 10.866919; Lng: 106.598904; Name: Trường học Tân Xuân; Code: HHM 051; StopType: Trụ dừng; Zone: Huyện Hóc
```

+ *Task5:*



```
From StopID: 7681 to StopID: 1247 with Time: 35.250590875748934s; Distance: 184.91872463569962m; RouteID: 27; RouteVarID: 54
From StopID: 7681 to StopID: 1202 with Time: 171.45167592816804s; Distance: 762.209856798162m; RouteID: 52; RouteVarID: 2
From StopID: 7682 to StopID: 7683 with Time: 1034.432148917911s; Distance: 3872.4895831285908m; RouteID: 337; RouteVarID: 1
From StopID: 7683 to StopID: 7684 with Time: 892.8077717117077s; Distance: 3342.3060171771626m; RouteID: 337; RouteVarID: 1
From StopID: 7683 to StopID: 7682 with Time: 1023.1181112405225s; Distance: 3864.5663272338325m; RouteID: 337; RouteVarID: 2
From StopID: 7684 to StopID: 7685 with Time: 583.6983123305303s; Distance: 2185.127015391216m; RouteID: 337; RouteVarID: 1
From StopID: 7684 to StopID: 7683 with Time: 899.5686426961497s; Distance: 3397.889889158373m; RouteID: 337; RouteVarID: 2
From StopID: 7685 to StopID: 7686 with Time: 599.3045616435476s; Distance: 2243.5504102553323m; RouteID: 337; RouteVarID: 1
From StopID: 7685 to StopID: 7684 with Time: 588.6219186096063s; Distance: 2223.3683688507085m; RouteID: 337; RouteVarID: 2
From StopID: 7686 to StopID: 7685 with Time: 598.5851272907717s; Distance: 2261.001834974918m; RouteID: 337; RouteVarID: 2
From StopID: 7693 to StopID: 1256 with Time: 14.934027028232258s; Distance: 76.29154378994082m; RouteID: 3; RouteVarID: 5
From StopID: 7693 to StopID: 1256 with Time: 17.167240281475653s; Distance: 77.31798027723653m; RouteID: 1; RouteVarID: 2
From StopID: 7693 to StopID: 1256 with Time: 17.608866708928755s; Distance: 76.29154378994082m; RouteID: 4; RouteVarID: 7
From StopID: 7693 to StopID: 1256 with Time: 13.15560461961905s; Distance: 76.29154378994082m; RouteID: 41; RouteVarID: 81
From StopID: 7693 to StopID: 1256 with Time: 11.771988952750515s; Distance: 76.29154378994082m; RouteID: 128; RouteVarID: 255
From StopID: 7694 to StopID: 3427 with Time: 2683.8406874123143s; Distance: 26063.075119981804m; RouteID: 204; RouteVarID: 2
From StopID: 7695 to StopID: 3742 with Time: 259.38909068736643s; Distance: 1572.1254238555173m; RouteID: 75; RouteVarID: 2
```

***GEOJSON – MAP VIEW OF SHORTEST PATHS FROM STOP 1 TO STOP 7276:**



VI) Conclusion

- This report details the development of a system that utilizes Dijkstra's algorithm to find the shortest paths between bus stops in Ho Chi Minh City. The system leverages data from three JSON files: "vars.json," "stops.json," and "paths.json," containing information on bus routes, stops, and travel times.

- The report is structured into six sections:

+ Structure: This section outlines the organization of the program's codebase, including the various Python files responsible for building the graph, handling user queries, and processing data.

+ Comments and Approaches: This section delves into the implementation details, explaining the rationale behind specific coding choices. It explores two methods for constructing the graph from the JSON data and justifies the selection of the more efficient approach (Graph2). It also discusses the optimization techniques employed for implementing Dijkstra's algorithm, including priority queues and trace dictionaries.

+ Class Diagram for Graph2: This section visually represents the Graph2 class, showcasing its attributes and methods used for managing the graph data structure.

+ Detailed Explanation: This section provides a step-by-step breakdown of the key functionalities within Graph2.py. It explains how the program reads data from the JSON files, constructs the graph with vertices and edges representing bus stops and connections, and implements Dijkstra's algorithm for both all pairs of stops and a single user-specified stop. Additionally, it details the approach for calculating the importance of each stop within the network.

+ User Interface in main.py: This section describes the user interface implemented in the "main.py" script. It explains how users can interact with the system to perform various tasks, such as displaying graph information, finding the shortest paths between specific stops, and retrieving the top 10 most important stops.

- Key Findings and Successes:

+ The report demonstrates the successful application of Dijkstra's algorithm to a real-world transportation network, enabling efficient route planning for bus travel in Ho Chi Minh City.

+ The chosen approach (Graph2) for constructing the graph from the JSON data offers a balance between accuracy and computational efficiency.

+ The implementation of Dijkstra's algorithm leverages optimization techniques like priority queues to handle large datasets efficiently.

+ The system calculates the importance of each stop within the network, providing valuable insights into bus traffic patterns and potentially informing route optimization strategies.

- Future Enhancements:

+ The system could be extended to incorporate real-time traffic data to provide more dynamic route suggestions.

+ The user interface could be further developed to offer a more interactive and visually appealing experience.

+ Additional functionalities could be implemented, such as multi-modal journey planning (combining buses with other transportation options).

+ The system's scalability could be improved to handle even larger datasets and more complex transportation networks.

→ Overall, this project demonstrates the effectiveness of Dijkstra's algorithm in optimizing route planning for public transportation systems. The report provides a comprehensive explanation of the system's development and paves the way for further enhancements towards a more user-friendly and adaptable solution for navigating Ho Chi Minh City's bus network.