UNIVERSITY OF SCIENCE,

HO CHI MINH

SHORTEST PATH IMPROVEMENTS USING CACHING AND FIXED ROUTES

TECHNICAL REPORT

Task 02

submitted for the Solo Project in Semester 3 – First Year

IT DEPARTMENT

Computer Science

by

Phan Tuan Kiet

Full name: Phan Tuan Kiet

ID: 23125062

Class: 23APCS2

Tasks achieved: 01/01

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

2024

Contents

1) INTRODUCTION

1.1) Algorithms:

- There are many algorithms for finding the shortest path, but additional techniques are required for vast graphs like those used in Google Maps or a world map. Some of the algorithms that are used worldwide and by Google Maps include:

+ Dijkstra: This algorithm finds the shortest paths from a single source vertex to all other vertices in a graph with non-negative edge weights. However, it is not efficient for large-scale graphs due to its worst-case complexity of $O(E + V \log V)$.

+ A*: An extension of Dijkstra's algorithm that uses heuristics to prioritise certain paths, making it more efficient.

+ Contraction Hierarchy: This algorithm preprocesses the graph by contracting nodes and adding shortcut edges, allowing faster query times during the shortest path search.

+ Two-way search / Bidirectional Dijkstra: This technique runs two simultaneous searches from the source and the destination, meeting in the middle to reduce the search space.

- While Dijkstra's algorithm is foundational, it is not practical for vast graphs due to its inefficiency. Therefore, more effective algorithms and techniques have been developed. These include:

+ A*: This algorithm improves on Dijkstra by using a heuristic function to guide the search.

+ Contraction Hierarchy: This involves preprocessing the graph to create a hierarchy of nodes, which allows for much faster shortest-path queries by effectively reducing the graph's complexity.

+ Bidirectional Dijkstra: By running two searches simultaneously (one forward from the source and one backwards from the destination), this method halves the search space, leading to quicker results.

1.2) Techniques:

- In real-life applications, several techniques are used to further enhance the performance of shortest-path algorithms. These include:

+ Dividing the shortest path into smaller beacons: This approach breaks down the path into smaller, more manageable segments or waypoints, simplifying the search process.

+ To the entrance, to the exit, and the destination: This method involves dividing the journey into key stages (entry points, exit points, and the final destination) to streamline the pathfinding process.

+ Pre-computing: By pre-computing certain paths or distances, the algorithm can quickly retrieve results during the actual search, saving significant computation time.

+ Caching fixed routes: Frequently used routes can be cached, allowing the system to bypass redundant calculations and retrieve results instantly.

+ Caching important segments of routes: Similar to caching fixed routes, this technique focuses on storing critical parts of routes that are commonly traversed, improving efficiency.

- These techniques collectively enhance the overall performance of shortest-path algorithms, making them suitable for large-scale, real-world applications. In this report, I will delve into each of these!

2) A STAR SEARCH

2.1) Overview

- A* is considered an improvement over Dijkstra's algorithm when heuristic costs are included, resulting in better performance. Why is this the case? In A* search, each node is assigned a specific heuristic cost that estimates the distance to the destination. This heuristic guides the search, making it more directed and efficient. Unlike Dijkstra's algorithm, which explores all possible paths to find the shortest one, A* uses the heuristic to prioritize nodes that are more likely to lead to the destination quickly. This focused approach significantly reduces the number of nodes that need to be explored, leading to faster search times.

- The heuristic cost I will use is the Euclidean distance from a node to the destination divided by the maximum speed on the entire route. In other words, the heuristic cost h(n) for a node n is calculated as:

$$h(n) = \text{Euclidean\_Distance}(u, v) \, / \, \text{max\_speed}$$

2.2) Code

- [module] : [a_star]

- This shortest path algorithm will be implemented in a separate class, which inherits from the Graph_Optimized class, to facilitate better testing and modularity. By placing the algorithm in a different class, we can isolate the functionality, making it easier to maintain, test, and extend. This approach allows us to leverage the optimized graph structures and methods from Graph_Optimized while focusing specifically on the shortest path computation in the new class:

```
class A_Star(Graph_Optimized):
    def __init__ (self, vars, stops, paths):
        super().__init__(vars, stops, paths)
```

- Heuristic function:

```
def get_heuristic(self, source, destination):
        lat1, lng1 = self.coordinatesAll[source]
        lat2, lng2 = self.coordinatesAll[destination]
        return self.distanceLL(lat1, lng1, lat2, lng2) / self.maxSpeed
```

- For vast graphs or maps (like the Earth), the heuristic function may not be accurate due to the Earth's round shape. Therefore, another heuristic cost used to address this issue is the haversine function, which calculates the shortest distance between two points on the surface of a sphere using their latitudes and longitudes.

- The haversine function is derived from the law of haversines, which relates the sides and angles of spherical triangles. This function is particularly useful in navigation and geospatial calculations, as it provides an accurate distance measurement that takes the Earth's curvature into account.

- The code below takes the latitudes and longitudes of two points as inputs and returns the distance between them in kilometres. It is widely used in mapping software, GPS devices, and various applications requiring accurate distance measurements over the Earth's surface.

```
def haversine(lat1, lon1, lat2, lon2):
        # Radius of the Earth in kilometers
        R = 6371.0

        # Convert latitude and longitude from degrees to radians
        phi1 = math.radians(lat1)
        phi2 = math.radians(lat2)
        delta_phi = math.radians(lat2 - lat1)
        delta_lambda = math.radians(lon2 - lon1)

        # Haversine formula
        a = math.sin(delta_phi / 2.0)**2 + \
            math.cos(phi1) * math.cos(phi2) * \
            math.sin(delta_lambda / 2.0)**2

        c = 2 * math.atan2(math.sqrt(a), math.sqrt(1 - a))

        # Distance in kilometers
        distance = R * c

        return distance
```

- Code for A_Star implementation:

```
def a_star_search(self, source, destination):
    g_score = {v: self.INF for v in self.numVertices}
    f_score = {v: self.INF for v in self.numVertices}
    trace = {v: -1 for v in self.numVertices}

    g_score[source] = 0
    f_score[source] = self.get_heuristic(source, destination)

    minHeap = []
    heapq.heappush(minHeap, (f_score[source], source))
    visited = set()

    while minHeap:
        f_cur, u = heapq.heappop(minHeap)
        if u == destination:
            break
        visited.add(u)

        # Optimization algorithm
        if (f_cur > f_score[u]):
            continue

        for info, neighbor, routeInfo in self.vertices[u]:
            if neighbor in visited:
                continue

            time = info[0]
            distance = info[1]
            routeId = routeInfo[0]
            routeVarId = routeInfo[1]

            inherited_cost = g_score[u] + time
            f = inherited_cost + self.get_heuristic(neighbor, destination)

            if f < f_score[neighbor]:
                g_score[neighbor] = inherited_cost
                f_score[neighbor] = f
                trace[neighbor] = u
                heapq.heappush(minHeap, (f, neighbor))
```

2.3) Time complexity

$$O(E \log V) = O(b^d)$$

- Compared to the Dijkstra algorithm, A* is a much more efficient one, when the time complexity is reduced significantly.

- The time complexity of the A* algorithm is influenced by the heuristic used. In the worst-case scenario involving an unbounded search space, the number of nodes expanded grows exponentially with the depth d of the solution. (i.e., the shortest path). Specifically, it is $O(b^d)$, where b represents the branching factor or the average number of successors per state. This analysis assumes that a goal state exists and is reachable from the start state. If no goal state exists or the state space is infinite, the algorithm will not terminate.

2.4) Space complexity

$$O(V) = O(b^d)$$

- The space complexity of the A* algorithm is generally considered to be $O(b^d)$, where b is the branching factor and d is the depth of the shortest path solution. This is because A* needs to keep all generated nodes in memory, which can lead to high memory usage, especially in cases with a large branching factor and depth. In practice, the space complexity can be a limiting factor for A* on large problems.

- However, if we combine A* with caching, the space complexity will be reduced because it doesn't have to store every node in memory; it can retrieve nodes from the pre-cache instead. Caching involves storing frequently accessed nodes in fast-access storage, allowing the algorithm to quickly retrieve previously computed information, thus reducing the overall memory footprint and improving efficiency. I will dive into caching in the next sections when I combine this technique with the A* algorithm.

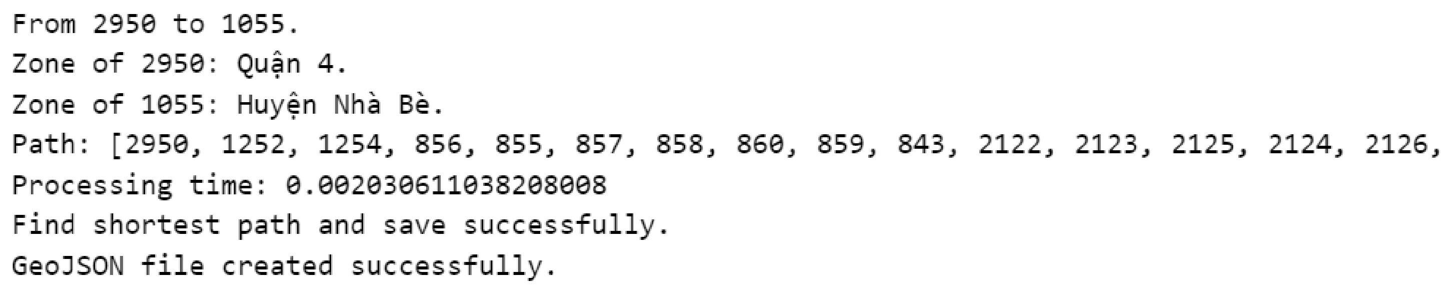
2.5) Results

***Notes:** For the overall performance, I will present after all with other algorithms and techniques.

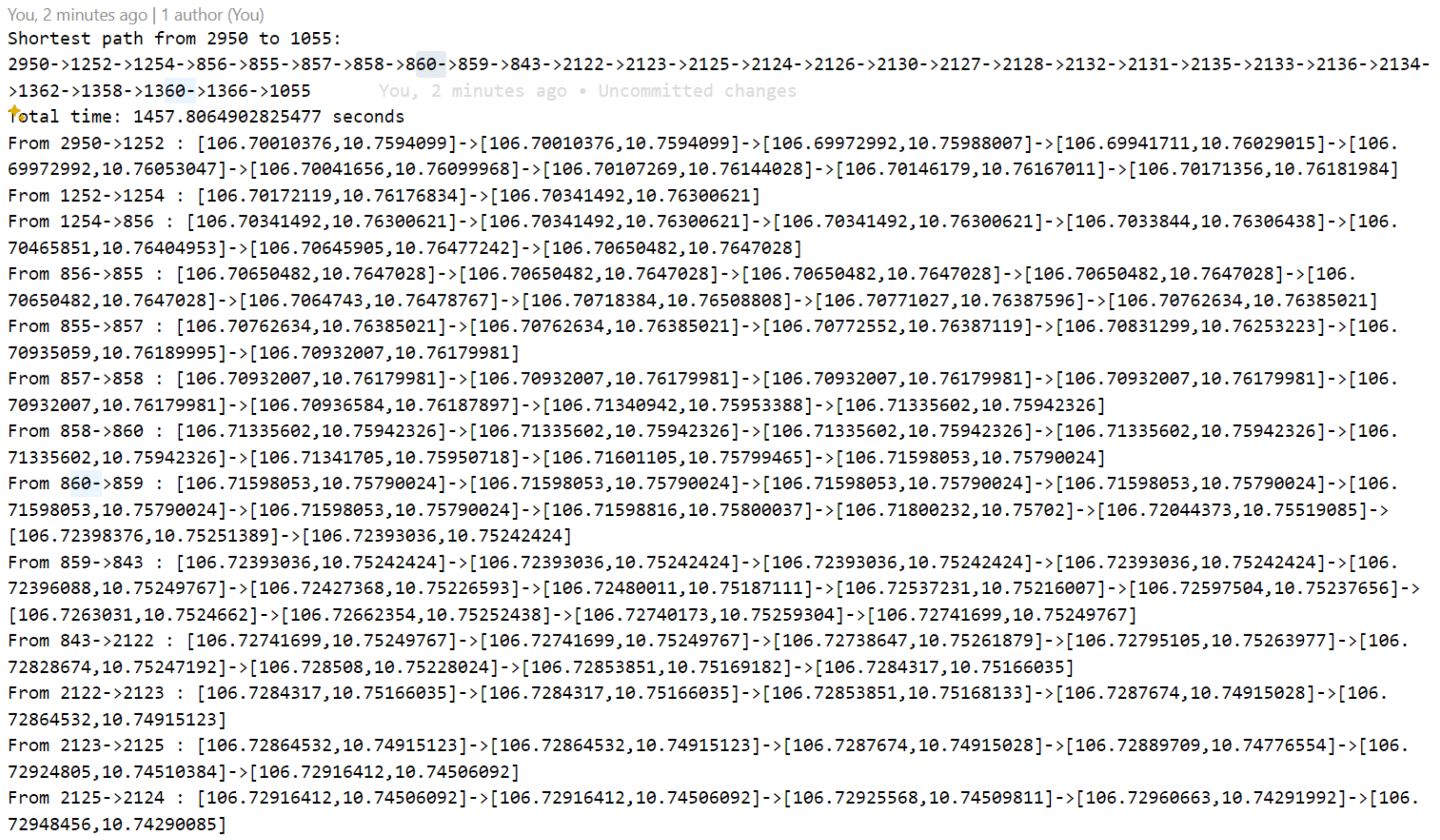
- Code for testing random query(u, v):

```
all_stops = []
for stop in graph_fixed.numVertices:
    all_stops.append(stop)
source = all_stops[random.randint(0, len(all_stops) - 1)]
destination = all_stops[random.randint(0, len(all_stops) - 1)]
print(f"From {source} to {destination}.")
print(f"Zone of {source}: {graph_fixed.zoneAll[source]}.")
print(f"Zone of {destination}: {graph_fixed.zoneAll[destination]}.")
time1 = time.time()
totalTime, path = graph_fixed.a_star_search(source, destination)
time2 = time.time()
print("Path:", path)
print("Processing time:", time2 - time1)
graph_fixed.saveOneSP(totalTime, path, source, destination)
```

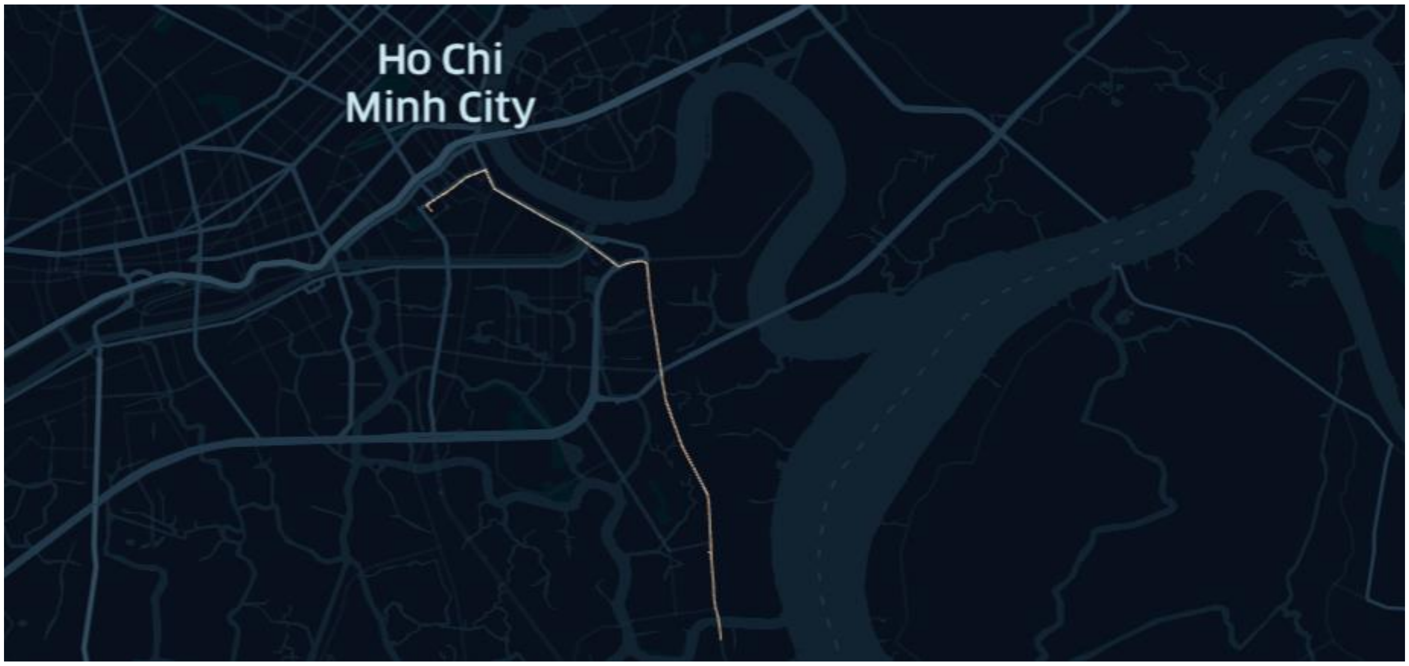
- Results:



```
From 2950 to 1055.
Zone of 2950: Quận 4.
Zone of 1055: Huyện Nhà Bè.
Path: [2950, 1252, 1254, 856, 855, 857, 858, 860, 859, 843, 2122, 2123, 2125, 2124, 2126,
Processing time: 0.002030611038208008
Find shortest path and save successfully.
GeoJSON file created successfully.
```



```
You, 2 minutes ago | 1 author (You)
Shortest path from 2950 to 1055:
2950->1252->1254->856->855->857->858->860->859->843->2122->2123->2125->2124->2126->2130->2127->2128->2132->2131->2135->2133->2136->2134-
>1362->1358->1360->1366->1055        You, 2 minutes ago • Uncommitted changes
Total time: 1457.8064902825477 seconds
From 2950->1252 : [106.70010376,10.7594099]->[106.70010376,10.7594099]->[106.69972992,10.75988007]->[106.69941711,10.76029015]->[106.
69972992,10.76053047]->[106.70041656,10.76099968]->[106.70107269,10.76144028]->[106.70146179,10.76167011]->[106.70171356,10.76181984]
From 1252->1254 : [106.70172119,10.76176834]->[106.70341492,10.76300621]
From 1254->856 : [106.70341492,10.76300621]->[106.70341492,10.76300621]->[106.70341492,10.76300621]->[106.7033844,10.76306438]->[106.
70465851,10.76404953]->[106.70645905,10.76477242]->[106.70650482,10.7647028]
From 856->855 : [106.70650482,10.7647028]->[106.70650482,10.7647028]->[106.70650482,10.7647028]->[106.70650482,10.7647028]->[106.
70650482,10.7647028]->[106.7064743,10.76478767]->[106.70718384,10.76508808]->[106.70771027,10.76387596]->[106.70762634,10.76385021]
From 855->857 : [106.70762634,10.76385021]->[106.70762634,10.76385021]->[106.70772552,10.76387119]->[106.70831299,10.76253223]->[106.
70935059,10.76189995]->[106.70932007,10.76179981]
From 857->858 : [106.70932007,10.76179981]->[106.70932007,10.76179981]->[106.70932007,10.76179981]->[106.70932007,10.76179981]->[106.
70932007,10.76179981]->[106.70936584,10.76187897]->[106.71340942,10.75953388]->[106.71335602,10.75942326]
From 858->860 : [106.71335602,10.75942326]->[106.71335602,10.75942326]->[106.71335602,10.75942326]->[106.71335602,10.75942326]->[106.
71335602,10.75942326]->[106.71341705,10.75950718]->[106.71601105,10.75799465]->[106.71598053,10.75790024]
From 860->859 : [106.71598053,10.75790024]->[106.71598053,10.75790024]->[106.71598053,10.75790024]->[106.71598053,10.75790024]->[106.
71598053,10.75790024]->[106.71598053,10.75790024]->[106.71598816,10.75800037]->[106.71800232,10.75702]->[106.72044373,10.75519085]->
[106.72398376,10.75251389]->[106.72393036,10.75242424]
From 859->843 : [106.72393036,10.75242424]->[106.72393036,10.75242424]->[106.72393036,10.75242424]->[106.72393036,10.75242424]->[106.
72396088,10.75249767]->[106.72427368,10.75226593]->[106.72480011,10.75187111]->[106.72537231,10.75216007]->[106.72597504,10.75237656]->
[106.7263031,10.7524662]->[106.72662354,10.75252438]->[106.72740173,10.75259304]->[106.72741699,10.75249767]
From 843->2122 : [106.72741699,10.75249767]->[106.72741699,10.75249767]->[106.72738647,10.75261879]->[106.72795105,10.75263977]->[106.
72828674,10.75247192]->[106.728508,10.75228024]->[106.72853851,10.75169182]->[106.7284317,10.75166035]
From 2122->2123 : [106.7284317,10.75166035]->[106.7284317,10.75166035]->[106.72853851,10.75168133]->[106.7287674,10.74915028]->[106.
72864532,10.74915123]
From 2123->2125 : [106.72864532,10.74915123]->[106.72864532,10.74915123]->[106.7287674,10.74915028]->[106.72889709,10.74776554]->[106.
72924805,10.74510384]->[106.72916412,10.74506092]
From 2125->2124 : [106.72916412,10.74506092]->[106.72916412,10.74506092]->[106.72925568,10.74509811]->[106.72960663,10.74291992]->[106.
72948456,10.74290085]
```



3) CACHING SEGMENTS

3.1) Overview of caching

- Caching is a crucial technique used by Google's map services to handle vast graphs in real-life scenarios. Algorithms alone are not sufficient for optimal performance, as discussed earlier. When searching for the shortest path between two points (u, v), precomputed segments or important routes, that have been cached, can be retrieved to minimize the total number of calculations. This significantly enhances the efficiency of the program, providing users with faster results.

- Google Maps employs caching to handle the vast and complex graph of global routes. By caching precomputed segments and important routes, the system can quickly assemble the shortest path between any two points. This approach not only reduces the computational burden but also ensures that users receive near-instantaneous route calculations.

3.2) Data caching

- **Caching Strategy:** It is not sufficient to cache everything during searching or querying. Typically, we cache the most important or frequently accessed data. This approach ensures that our cache remains efficient and effective.

- **Fixed Routes:** Another key aspect of caching is fixed routes, which play a crucial role in shortest path queries. As mentioned earlier, a path can be divided into smaller segments. If the route between two fixed points from the source and destination is already cached, we only need to find the path from the source to the first fixed route and then from the second fixed route to the destination.

- **Focus of this Module:** In this module, I will present the concept of caching important routes, which forms the foundation of effective caching strategies. The details of fixed routes caching will be covered in a subsequent module.

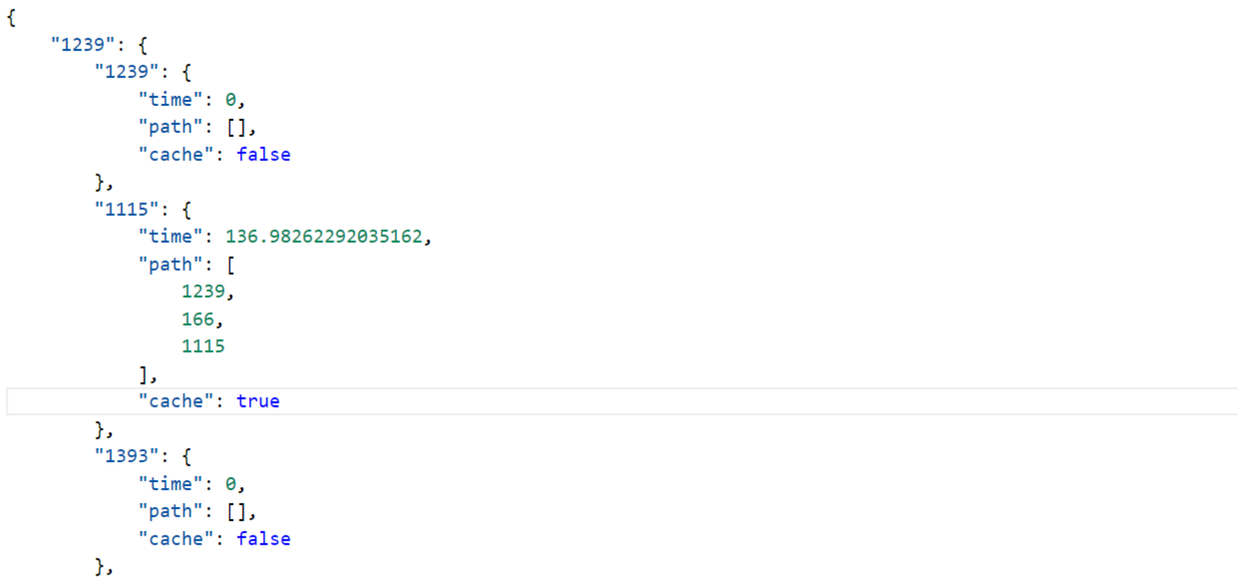
3.3) Caching important segments

3.3.1) Pre-calculate the top thousand important segments

- We will use the algorithm for finding important segments developed in the previous semester. However, first, we need to build cache storage on our local computer and adjust its properties according to our purpose. In this module, we will save 'time', 'path', and 'cache' as properties of important segments caching.

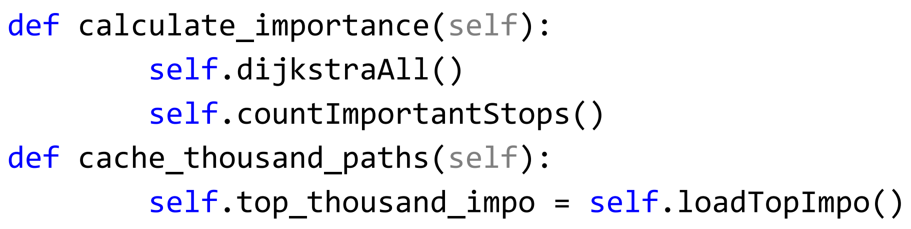
```
def init_impo(self):
        impo_cache = {}

        for u in self.top_thousand_impo:
            if u not in impo_cache:
                impo_cache[u] = {}
            for v in self.top_thousand_impo:
                if v not in impo_cache[u]:
                    impo_cache[u][v] = {}
                impo_cache[u][v]['time'] = 0
                impo_cache[u][v]['path'] = []
                impo_cache[u][v]['cache'] = False

        with open('input/impo_cache.json', 'w') as file:
            json_object = json.dumps(impo_cache, indent = 4)
            file.write(json_object)
            file.close()
```

- What will the cache look like?



```
{
    "1239": {
        "1239": {
            "time": 0,
            "path": [],
            "cache": false
        },
        "1115": {
            "time": 136.98262292035162,
            "path": [
                1239,
                166,
                1115
            ],
            "cache": true
        },
        "1393": {
            "time": 0,
            "path": [],
            "cache": false
        },
```

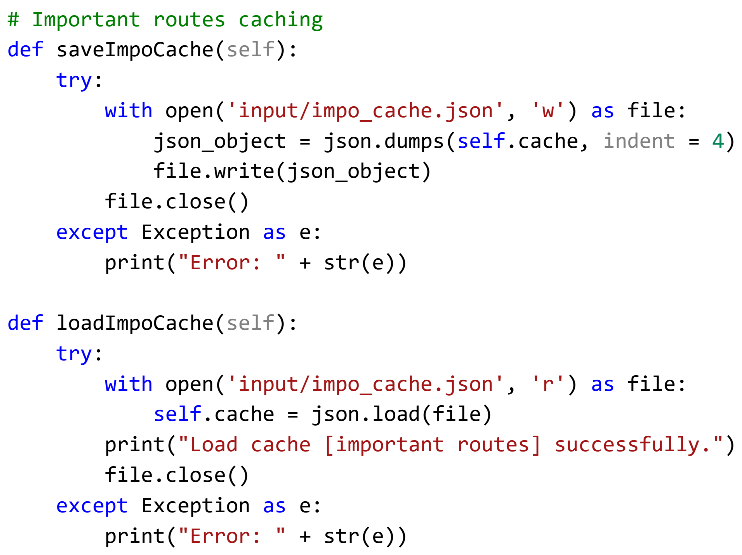
- Now, we can proceed to calculate the importance of all stops and then store the top 1,000 important ones in our cache:



```
def calculate_importance(self):
        self.dijkstraAll()
        self.countImportantStops()
def cache_thousand_paths(self):
        self.top_thousand_impo = self.loadTopImpo()
```

***Notes:** For the Dijkstra all pairs and count the importance of a stop, which I already presented in the previous report. The loadTopImpo function is used to load all the important stops into our program.

- For the helper functions of saving and loading top important segments into our cache:



```
# Important routes caching
def saveImpoCache(self):
    try:
        with open('input/impo_cache.json', 'w') as file:
            json_object = json.dumps(self.cache, indent = 4)
            file.write(json_object)
        file.close()
    except Exception as e:
        print("Error: " + str(e))

def loadImpoCache(self):
    try:
        with open('input/impo_cache.json', 'r') as file:
            self.cache = json.load(file)
        print("Load cache [important routes] successfully.")
        file.close()
    except Exception as e:
        print("Error: " + str(e))
```

3.3.2) A_Star_Search with caching of important segments

- Code:

```
    # 2.2) Get the cached important routes while searching for the shortest path.
    def a_star_search_cache(self, source, destination):
        if str(source) in self.cache and str(destination) in self.cache:
            if (self.cache[f'{source}'][f'{destination}']['cache'] == True):
                print('Found cache important route! No need to calculate
[important routes].')
                return self.cache[f'{source}'][f'{destination}']['time'],
self.cache[f'{source}'][f'{destination}']['path']

        g_score = {v: self.INF for v in self.numVertices}
        f_score = {v: self.INF for v in self.numVertices}
        trace = {v: -1 for v in self.numVertices}

        g_score[source] = 0
        f_score[source] = self.get_heuristic(source, destination)

        minHeap = []
        heapq.heappush(minHeap, (f_score[source], source))
        visited = set()

        while minHeap:
            f_cur, u = heapq.heappop(minHeap)
            if u == destination:
                break
            visited.add(u)

            # Optimization algorithm
            if (f_cur > f_score[u]):
                continue

            for info, neighbor, routeInfo in self.vertices[u]:
                if neighbor in visited:
                    continue

                time = info[0]
                distance = info[1]
                routeId = routeInfo[0]
                routeVarId = routeInfo[1]

                inherited_cost = g_score[u] + time
                f = inherited_cost + self.get_heuristic(neighbor, destination)

                if f < f_score[neighbor]:
                    g_score[neighbor] = inherited_cost
                    f_score[neighbor] = f
                    trace[neighbor] = u
                    heapq.heappush(minHeap, (f, neighbor))

        path = []
        current = destination
        while current != -1:
            path.append(current)
            current = trace[current]
            if current == source:
                path.append(source)
                break

        if current != source:
            return self.INF, []

        path.reverse()

        if str(source) in self.cache and str(destination) in
self.cache:
            print('Found new important route! Caching [important
routes].')
            self.cache[f'{source}'][f'{destination}']['time'] =
f_score[destination]
            self.cache[f'{source}'][f'{destination}']['path'] = path
            self.cache[f'{source}'][f'{destination}']['cache'] = True

        return f_score[destination], path
```

- Explanation:

+ The `a_star_search_cache` method implements the A* search algorithm with caching to find the shortest path between a source and destination in a graph.

+ Initially, it checks if there is a cached route for the given source and destination. If a cached route exists, it retrieves and returns the cached time and path, avoiding the need for recalculation.

+ If no cache is found, the algorithm initializes several data structures: `g_score` to track the cost from the start to each node, `f_score` for the estimated total cost from the start to the goal through each node, and `trace` to record the best previous node for path reconstruction. The scores for the source node are set to zero, and its heuristic cost to the destination is calculated.

+ A priority queue (min-heap) is used to efficiently process nodes based on their `f_score`. The main loop continues until there are no more nodes to process. The current node with the lowest `f_score` is popped from the heap, and if it is the destination, the loop ends.

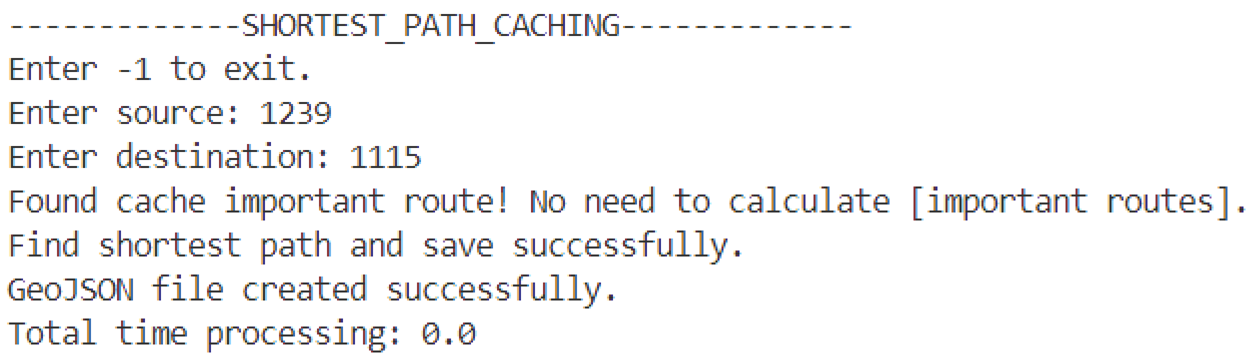
+ For each neighbour of the current node, if it has already been visited, it is skipped. The algorithm calculates the cost to reach each neighbour and updates the scores accordingly. If a better path is found to a neighbour, it updates the `g_score`, `f_score`, and `trace`.

+ Once the destination is reached, the method reconstructs the path using the `trace` information. If a new important route is found, it caches the time, and path, and sets a flag indicating that the route has been cached.

→ Finally, the method returns the time and path to the destination, whether retrieved from the cache or computed anew.

3.3.3) Results of A_Star_Search with Segments Caching

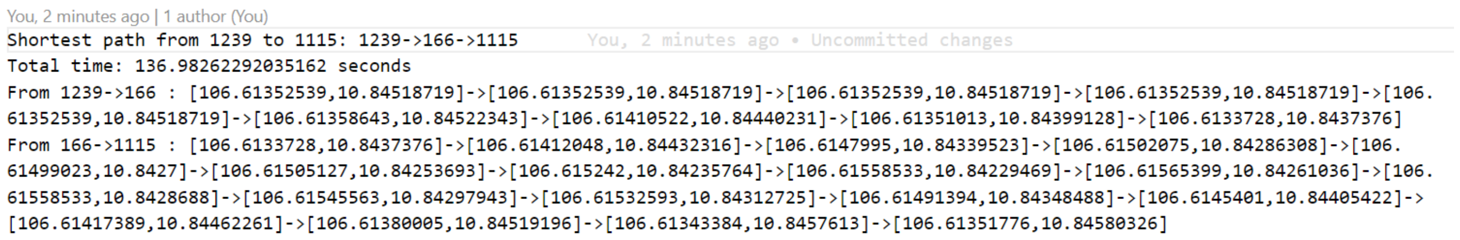
- Results of finding a pre-cached important route:



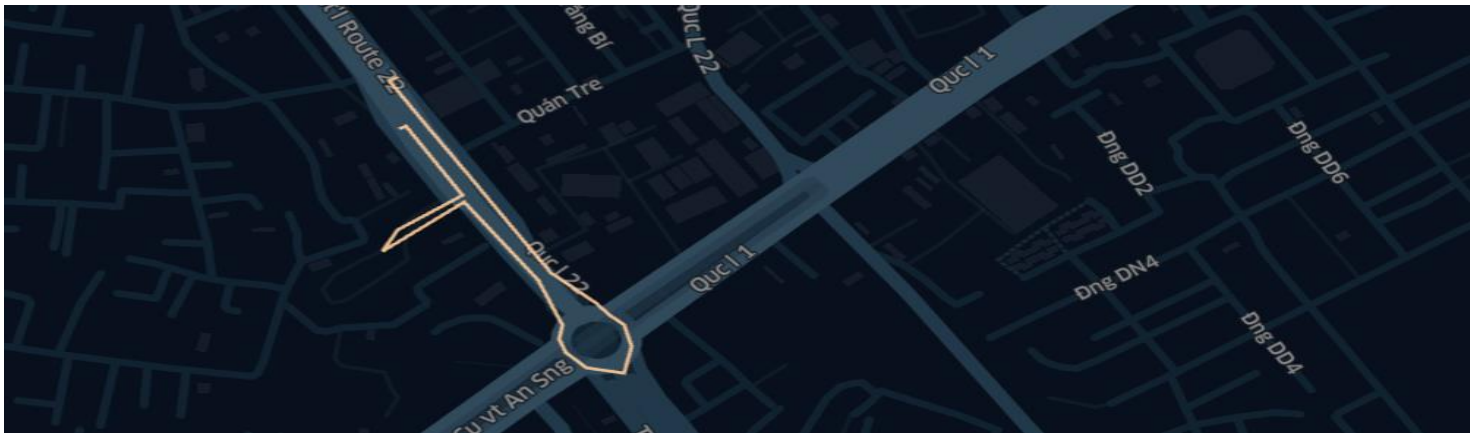
```
-------------SHORTEST_PATH_CACHING------------
Enter -1 to exit.
Enter source: 1239
Enter destination: 1115
Found cache important route! No need to calculate [important routes].
Find shortest path and save successfully.
GeoJSON file created successfully.
Total time processing: 0.0
```

→ The total time processing is 0.0 seconds as it is already cached, no calculations are needed.

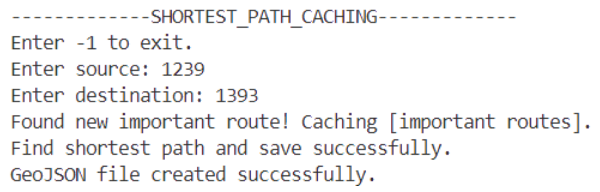
- Visualization of important routes:



```
You, 2 minutes ago | 1 author (You)
Shortest path from 1239 to 1115: 1239->166->1115        You, 2 minutes ago • Uncommitted changes
Total time: 136.98262292035162 seconds
From 1239->166 : [106.61352539,10.84518719]->[106.61352539,10.84518719]->[106.61352539,10.84518719]->[106.61352539,10.84518719]->[106.
61352539,10.84518719]->[106.61358643,10.84522343]->[106.61410522,10.84440231]->[106.61351013,10.84399128]->[106.6133728,10.8437376]
From 166->1115 : [106.6133728,10.8437376]->[106.61412048,10.84432316]->[106.6147995,10.84339523]->[106.61502075,10.84286308]->[106.
61499023,10.8427]->[106.61505127,10.84253693]->[106.615242,10.84235764]->[106.61558533,10.84229469]->[106.61565399,10.84261036]->[106.
61558533,10.8428688]->[106.61545563,10.84297943]->[106.61532593,10.84312725]->[106.61491394,10.84348488]->[106.6145401,10.84405422]->
[106.61417389,10.84462261]->[106.61380005,10.84519196]->[106.61343384,10.8457613]->[106.61351776,10.84580326]
```

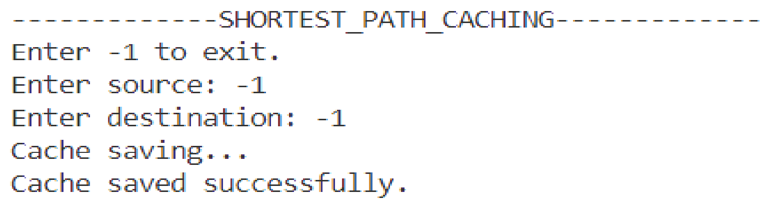


- If we query a route that belongs to the top important routes but is not cached, we will cache it for later use:



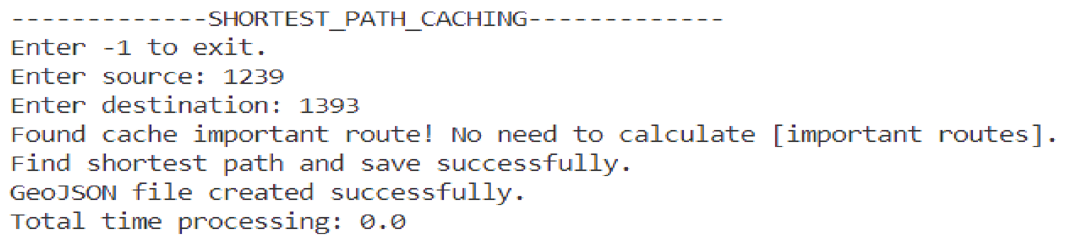
```
-------------SHORTEST_PATH_CACHING-------------
Enter -1 to exit.
Enter source: 1239
Enter destination: 1393
Found new important route! Caching [important routes].
Find shortest path and save successfully.
GeoJSON file created successfully.
```

- After we exit the program, we will cache it:



```
-------------SHORTEST_PATH_CACHING-------------
Enter -1 to exit.
Enter source: -1
Enter destination: -1
Cache saving...
Cache saved successfully.
```

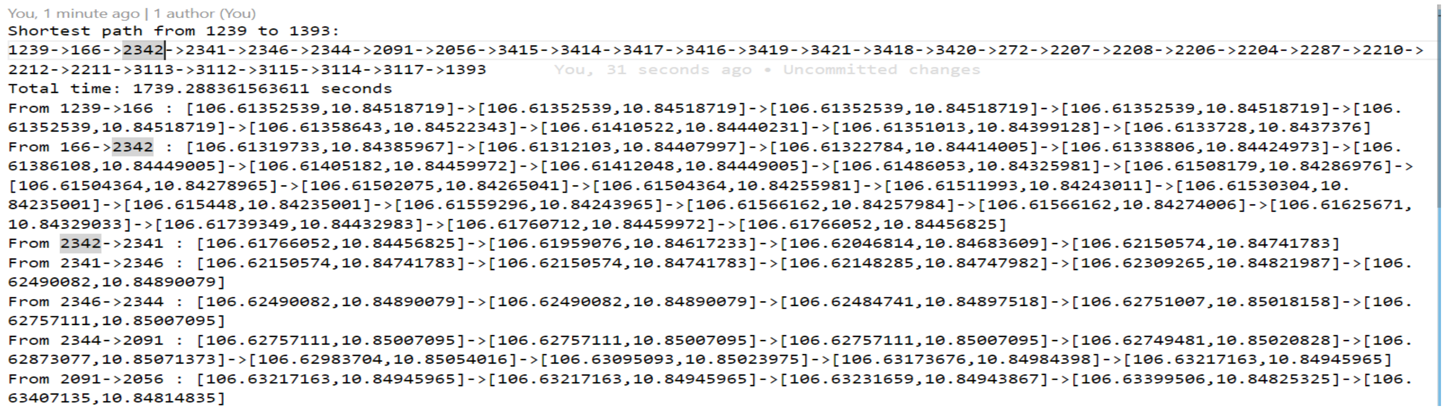
- Test again when we query stop 1239 and 1393:



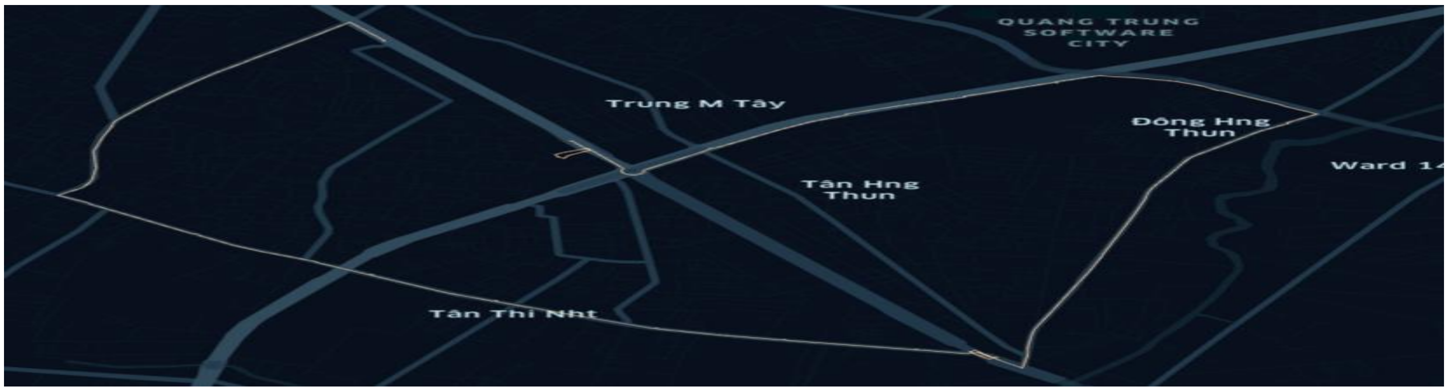
```
-------------SHORTEST_PATH_CACHING-------------
Enter -1 to exit.
Enter source: 1239
Enter destination: 1393
Found cache important route! No need to calculate [important routes].
Find shortest path and save successfully.
GeoJSON file created successfully.
Total time processing: 0.0
```

→ This is done successfully!

- Visualization of this route:



```
You, 1 minute ago | 1 author (You)
Shortest path from 1239 to 1393:
1239->166->2342->2341->2346->2344->2091->2056->3415->3414->3417->3416->3419->3421->3418->3420->272->2207->2208->2206->2204->2287->2210->
2212->2211->3113->3112->3115->3114->3117->1393        You, 31 seconds ago • Uncommitted changes
Total time: 1739.288361563611 seconds
From 1239->166 : [106.61352539,10.84518719]->[106.61352539,10.84518719]->[106.61352539,10.84518719]->[106.61352539,10.84518719]->[106.
61352539,10.84518719]->[106.61358643,10.84522343]->[106.61410522,10.84440231]->[106.61351013,10.84399128]->[106.6133728,10.8437376]
From 166->2342 : [106.61319733,10.84385967]->[106.61312103,10.84407997]->[106.61322784,10.84414005]->[106.61338806,10.84424973]->[106.
61386108,10.84449005]->[106.61405182,10.84459972]->[106.61412048,10.84449005]->[106.61486053,10.84325981]->[106.61508179,10.84286976]->
[106.61504364,10.84278965]->[106.61502075,10.84265041]->[106.61504364,10.84255981]->[106.61511993,10.84243011]->[106.61530304,10.
84235001]->[106.615448,10.84235001]->[106.61559296,10.84243965]->[106.61566162,10.84257984]->[106.61566162,10.84274006]->[106.61625671,
10.84329033]->[106.61739349,10.84432983]->[106.61760712,10.84459972]->[106.61766052,10.84456825]
From 2342->2341 : [106.61766052,10.84456825]->[106.61959076,10.84617233]->[106.62046814,10.84683609]->[106.62150574,10.84741783]
From 2341->2346 : [106.62150574,10.84741783]->[106.62150574,10.84741783]->[106.62148285,10.84747982]->[106.62309265,10.84821987]->[106.
62490082,10.84890079]
From 2346->2344 : [106.62490082,10.84890079]->[106.62490082,10.84890079]->[106.62484741,10.84897518]->[106.62751007,10.85018158]->[106.
62757111,10.85007095]
From 2344->2091 : [106.62757111,10.85007095]->[106.62757111,10.85007095]->[106.62757111,10.85007095]->[106.62749481,10.85020828]->[106.
62873077,10.85071373]->[106.62983704,10.85054016]->[106.63095093,10.85023975]->[106.63173676,10.84984398]->[106.63217163,10.84945965]
From 2091->2056 : [106.63217163,10.84945965]->[106.63217163,10.84945965]->[106.63231659,10.84943867]->[106.63399506,10.84825325]->[106.
63407135,10.84814835]
```



3.3.4) Time complexity and Space complexity after Caching Segments

- The time complexity of the A* algorithm, even with caching, primarily depends on the number of nodes explored. In the worst case, the complexity is:

- **Without Caching:** $O(b^d)$, where b is the branching factor and d is the depth of the solution.
- **With Caching:** The initial cache check adds a constant time overhead, so the overall complexity remains $O(b^d)$. However, if a cached route is found, the time complexity for that specific query becomes $O(1)$.

- The space complexity accounts for the storage of various data structures:

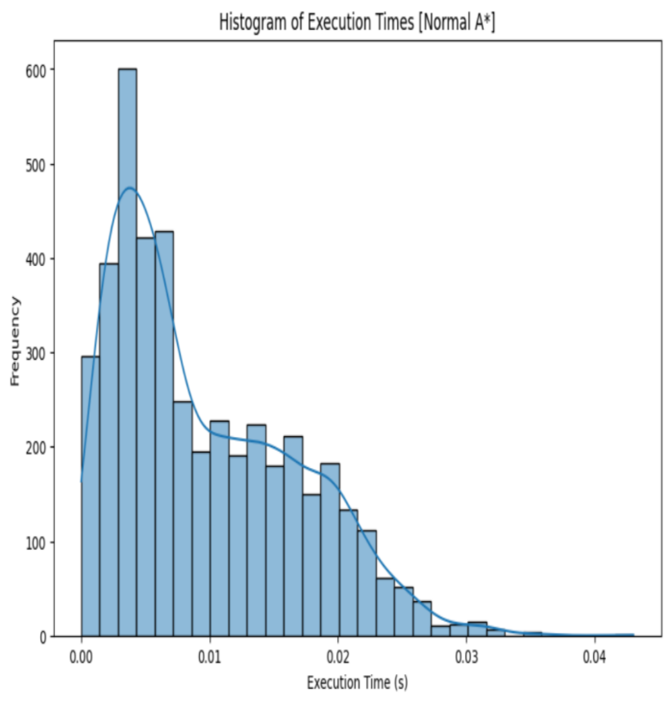
- **g_score, f_score, trace:** Each of these dictionaries stores a value for each vertex, leading to $O(V)$, where V is the number of vertices.
- **Min-Heap:** The heap can contain at most $O(V)$ nodes at any time.
- **Visited Set:** The visited set also takes up $O(V)$ space.
- **Cache Storage:** The cache's space complexity depends on how many routes are cached. If caching the top important routes, this can lead to $O(K)$, where K is the number of cached routes.

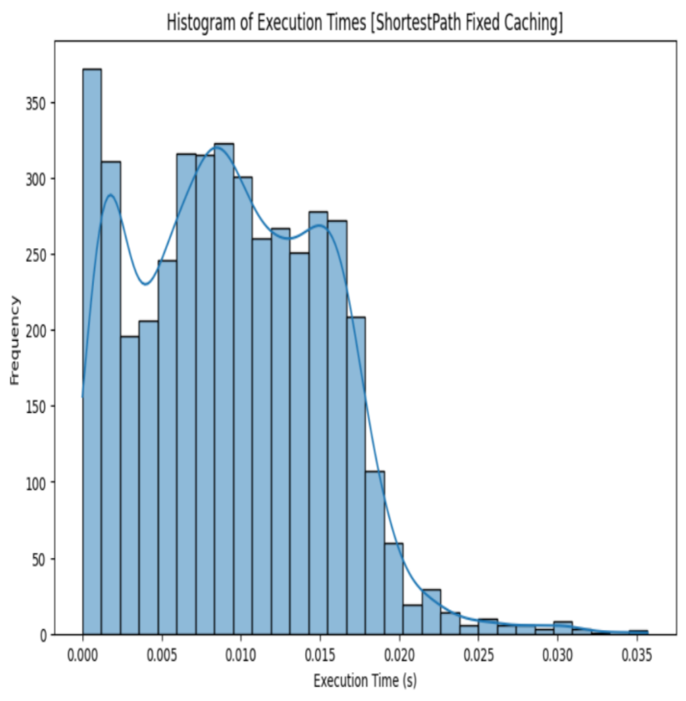
→ Overall, the space complexity is:

- **With Caching:** $O(V + K)$, where V is the number of vertices and K is the number of cached routes.
- **Without Caching:** $O(V)$, where V is the number of vertices.

- We can see that to reach $O(1)$In time complexity for important segments, we have to sacrifice the space complexity to store the cache. However, this is a good tradeoff as more enhancements will be added to the cache in a later module. This module just gives a foundation for segment caching.

- Some visualizations of the performance of A_Star with caching and A_Star without caching (details will be shown in the performance module in this report)





4) FIXED ROUTES CACHING

4.1) Path Divisions

- Firstly, to understand what fixed routes are, it is essential to define beacons. Beacons are vertices or nodes that are highly relevant for navigation within a specific area but are less significant from an external perspective. For example, when driving from point A to point B, one must decide where to enter a specific area (Point C) and where to exit (Point D). This decision allows the journey to be broken down into three smaller segments: reaching the entrance, navigating to the exit, and finally arriving at the destination using the optimal route. Consequently, a routing algorithm only needs to consider these special points (beacons) to navigate efficiently between points A and B, enabling it to find the shortest and most accurate path more quickly.

- Now you understand what the techniques are about, we will consider those beacons as fixed_routes and save them for later use (caching → previous module).

- As this techniques require many outside functions, I will split it into another class which is inherited from the class A_Star, as I will combine fixed_caching with a_star_search_cache that I have mentioned and described in the previous module:

```
class Fixed_Caching(A_Star):
    def __init__ (self, vars, stops, paths):
        super().__init__(vars, stops, paths)
        #self.cache_thousand_paths()
        self.loadImpoCache()  # Load cache 1000 important paths
        self.fixed_route = {}
        self.load_fixed_route() # Load cache fixed route
```

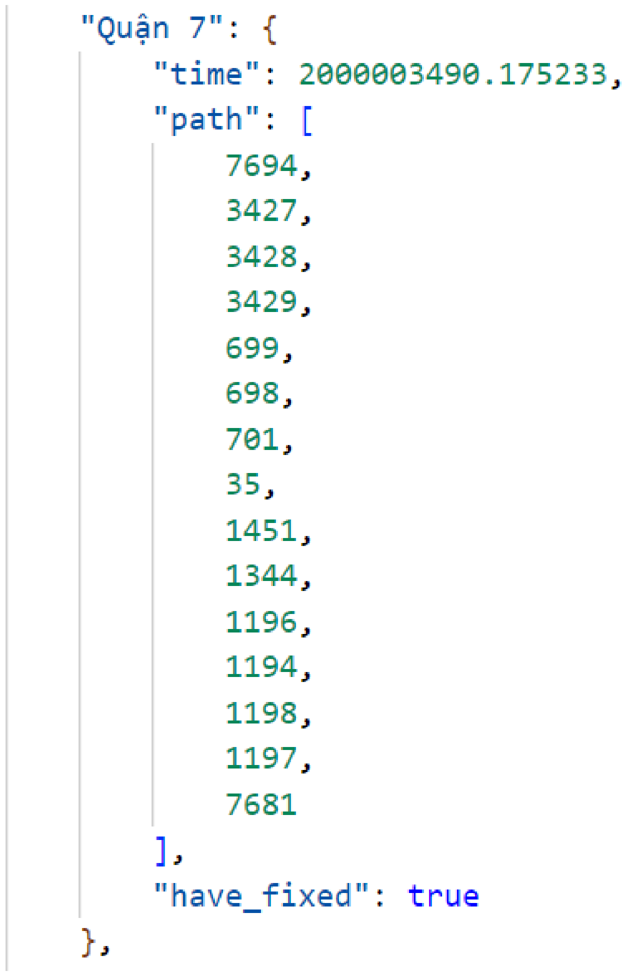
- Notice that to cache the fixed_routes, we have to load it first, I will dive into it in detail in the 4.2 sub-module.

4.2) Initialization and helper functions

- Like the previous caching module, we have to init a cache form first:

```
def init_cache(self):
    with open('input/stop_zone.json', 'r', encoding='utf-8') as file:
        stop_zone = json.load(file)
        file.close()

    fixed_route = {}
    for u in tqdm(self.numVertices):
        for v in self.numVertices:
            u_zone = stop_zone[str(u)]['zone']
            v_zone = stop_zone[str(v)]['zone']
            if u_zone not in fixed_route:
                fixed_route[u_zone] = {}
            if v_zone not in fixed_route[u_zone]:
                fixed_route[u_zone][v_zone] = {}

            fixed_route[u_zone][v_zone]['time'] = 0
            fixed_route[u_zone][v_zone]['path'] = []
            fixed_route[u_zone][v_zone]['have_fixed'] = ''

    with open('input/fixed_route.json', 'w', encoding='utf-8') as file:
        json_object = json.dumps(fixed_route, indent = 4, ensure_ascii = False)
        file.write(json_object)
        file.close()
    print("Init caching for fixed routes successfully.")
```

- Its visualization would look like this:



```
"Quận 7": {
    "time": 2000003490.175233,
    "path": [
        7694,
        3427,
        3428,
        3429,
        699,
        698,
        701,
        35,
        1451,
        1344,
        1196,
        1194,
        1198,
        1197,
        7681
    ],
    "have_fixed": true
},
```

- You will notice that there is a stop_zone, this is used to make the cache_fixed_routes as we have to know the zones of all stops in our graph to check if two_zone have fixed routes or not:

```python
def save_stop_json(self):
    stop_zone = {}
    for i in tqdm(self.numVertices):
        stop_zone[i] = {}
        stop_zone[i]['lat'] = self.coordinatesAll[i][0]
        stop_zone[i]['lng'] = self.coordinatesAll[i][1]
        stop_zone[i]['zone'] = self.zoneAll[i]

    with open('output/stop_zone.json', 'w', encoding='utf-8') as file:
        json_object = json.dumps(stop_zone, indent = 4, ensure_ascii = False)
        file.write(json_object)
        file.close()
    print("Stop zone saved successfully.")
```

4.3) Algorithms and Dynamic Updating

4.3.1) Logic and algorithms

- Our goal is to develop an algorithm to find the shortest paths using three segments: from the source to the first fixed point, from the first fixed point to the second fixed point, and from the second fixed point to the destination. Let us delve into the algorithm:

- Notations

- u: source
- v: destination

*** Cases to consider:**

- *Case 1: Same Zone*

+ If u_zone is the same as v_zone, there is no benefit in using fixed routes. In this scenario, we simply use the A* search algorithm.

- *Case 2: Different Zones Without Fixed Routes*

+ If u_zone differs from v_zone and there are no fixed routes between them, we proceed with the A* search algorithm as usual.

- *Case 3: Different Zones With Fixed Routes*

+ If u_zone and v_zone are different and there are fixed routes between them, we encounter a path division problem:

  - We calculate the total time from u to v without using any fixed routes, denoted as new_time.

  - Let cur_time be the sum of the time from the source to the first fixed point, from the first fixed point to the second fixed point, and the second fixed point to the destination.

  - If new_time is greater than or equal to cur_time, we return cur_time and cur_path (which uses the fixed routes).

  - If new_time is less than cur_time, we return new_time and new_path and update the fixed routes as these provide a more efficient route.

- *Case 4: Unknown Fixed Routes Between Different Zones*

+ If u_zone and v_zone are different and it is unknown whether there are fixed routes between them, we:

  - Check if there is another zone along the shortest path from u to v. If such a zone exists, we mark this shortest path as a fixed route.

  - If no such intermediate zone exists, we conclude that there are no fixed routes between these two zones and do not cache fixed routes for adjacent zones.

→ This method effectively identifies the shortest paths by leveraging both direct and fixed routes. By categorizing different cases based on zone relationships and the presence of fixed routes, the algorithm ensures optimal pathfinding. It dynamically updates fixed routes when more efficient paths are discovered, enhancing future path computations. This approach balances efficiency and adaptability, providing a robust solution for shortest-path determination in complex networks. **Below is the code in detail:**

```python
def shortest_path_fixed(self, source, destination):
    u_zone = self.zoneAll[source]
    v_zone = self.zoneAll[destination]
    if (u_zone == v_zone):
        print("Same zone, no need caching [fixed routes].")
        time, path = self.a_star_search_cache(source, destination)
        return time, path
    elif (self.fixed_route[u_zone][v_zone]['have_fixed'] == False):
        print("Two zones have no fixed route, no need caching [fixed routes].")
        time, path = self.a_star_search_cache(source, destination)
        return time, path
    elif (self.fixed_route[u_zone][v_zone]['have_fixed'] == True):
        print("Two zones have fixed route, use caching [fixed routes].")

        new_time, new_path = self.a_star_search_cache(source, destination)
        cache_time = self.fixed_route[u_zone][v_zone]['time']
        cache_path = self.fixed_route[u_zone][v_zone]['path']
        time1, path1 = self.a_star_search_cache(source, cache_path[0])
        time2, path2 = self.a_star_search_cache(cache_path[-1], destination)
        curTime = time1 + cache_time + time2
        curPath = path1 + cache_path + path2

        if new_time >= curTime:
            return curTime, curPath
        else:
            tmp_time, update = self.update_fix_route(curPath, new_path, curTime, new_time)
            if (update):
                self.fixed_route[u_zone][v_zone]['time'] = tmp_time
                self.fixed_route[u_zone][v_zone]['path'] = update
                self.fixed_route[u_zone][v_zone]['have_fixed'] = True
            else:
                self.fixed_route[u_zone][v_zone]['time'] = 0
                self.fixed_route[u_zone][v_zone]['path'] = []
                self.fixed_route[u_zone][v_zone]['have_fixed'] = False
            return new_time, new_path

    elif (self.fixed_route[u_zone][v_zone]['have_fixed'] == ""):
        print("Two zones have no fixed route, consider for caching [fixed routes].")
        time, path = self.a_star_search_cache(source, destination)
        for v in path:
            if (self.zoneAll[v] != u_zone and self.zoneAll[v] != v_zone):
                self.fixed_route[u_zone][v_zone]['time'] = time
                self.fixed_route[u_zone][v_zone]['path'] = path
                self.fixed_route[u_zone][v_zone]['have_fixed'] = True
                return time, path
        self.fixed_route[u_zone][v_zone]['time'] = 0
        self.fixed_route[u_zone][v_zone]['path'] = []
        self.fixed_route[u_zone][v_zone]['have_fixed'] = False
        return time, path
```

4.3.2) Dynamic updating

- In the algorithm above, I mentioned about update fixed_routes when the new_shortest path between two different zones is more efficient than the old fixed_routes. When this happens, we will find the longest common subsequence of stops (LCS) between the new shortest_path and the fixed_routes. If they have LCS, we will update the new fixed routes between the two zones. On the other hand, if no LCS is found, which means this new shortest path goes in different directions, we will mark there are no fixed routes between the two zones.

- This dynamic updating will ensure that every new better fixed_routes is updated as soon as possible and make sure that not so many fixed_routes are found (which makes our space complexity efficient)!

- Code:

```
def update_fix_route(self, path1, path2, time1, time2):
    counter1 = Counter(path1)
    counter2 = Counter(path2)

    common = counter1 & counter2
    common = list(common.elements())

    new_time = 0

    if len(common) < 2:
        return new_time, common

    for u, v in zip(common, common[1:]):
        new_time += self.timeAll[u][v]

    return new_time, common
```

- Explanation about finding LCS:

+ Normally, we will have to use dynamic programming to find the LCS between two subsequences for better efficiency. After I check by this method (using the Counter library), no errors occur, which means that if there is an LCS, they will be in the same direction → still giving us the correct path and time of the shortest path.

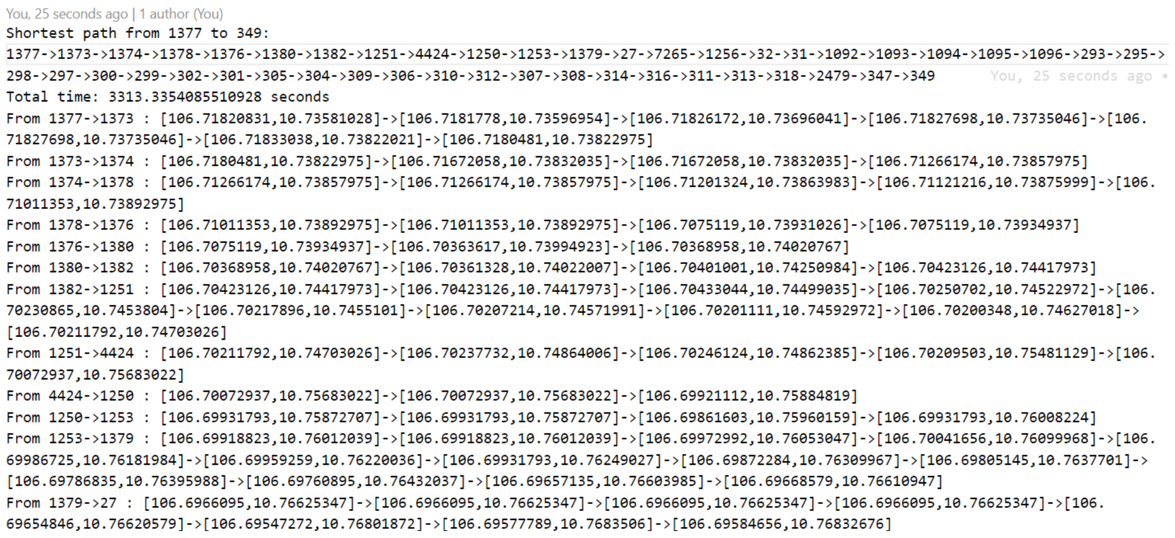
4.4) Results of fixed_routes_caching with A_Star_Search

- Code for testing random query(u,v):

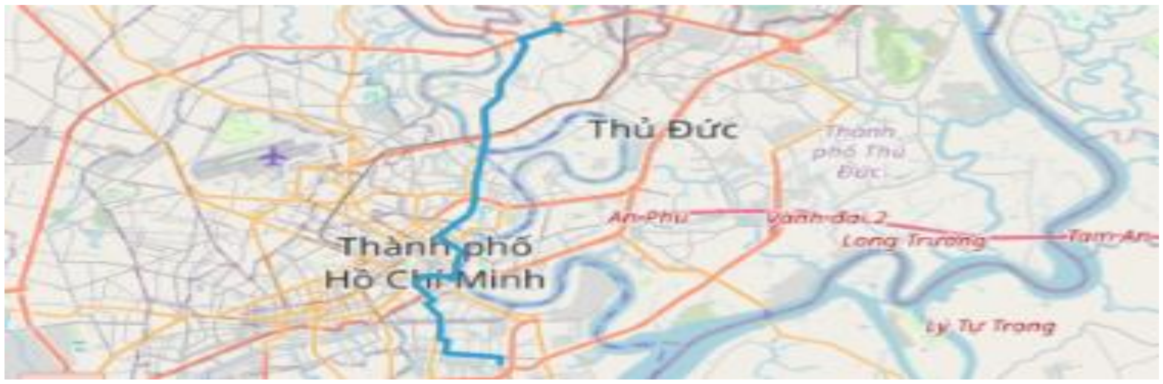
```
all_stops = []
for stop in graph_fixed.numVertices:
    all_stops.append(stop)
source = all_stops[random.randint(0, len(all_stops) - 1)]
destination = all_stops[random.randint(0, len(all_stops) - 1)]
print(f"From {source} to {destination}.")
print(f"Zone of {source}: {graph_fixed.zoneAll[source]}.")
print(f"Zone of {destination}: {graph_fixed.zoneAll[destination]}.")
time1 = time.time()
totalTime, path = graph_fixed.shortest_path_fixed(source, destination)
time2 = time.time()
print("Path:", path)
print("Processing time:", time2 - time1)
graph_fixed.saveOneSP(totalTime, path, source, destination)
```

- Results:

```
From 1377 to 349.
Zone of 1377: Quận 7.
Zone of 349: Quận Thủ Đức.
Two zones have no fixed route, no need caching [fixed routes].
Path: [1377, 1373, 1374, 1378, 1376, 1380, 1382, 1251, 4424, 1250, 1253, 1379, 27, 7265, 1256, 32, 31, 1092, 1093, 1094, 1095, 1096, 293, 2
Processing time: 0.008261680603027344
Find shortest path and save successfully.
GeoJSON file created successfully.
```



```
You, 25 seconds ago | 1 author (You)
Shortest path from 1377 to 349:
1377->1373->1374->1378->1376->1380->1382->1251->4424->1250->1253->1379->27->7265->1256->32->31->1092->1093->1094->1095->1096->293->295->
298->297->300->299->302->301->305->304->309->306->310->312->307->308->314->316->311->313->318->2479->347->349        You, 25 seconds ago •
Total time: 3313.3354085510928 seconds
From 1377->1373 : [106.71820831,10.73581028]->[106.7181778,10.73596954]->[106.71826172,10.73696041]->[106.71827698,10.73735046]->[106.
71827698,10.73735046]->[106.71833038,10.73822021]->[106.7180481,10.73822975]
From 1373->1374 : [106.7180481,10.73822975]->[106.71672058,10.73832035]->[106.71672058,10.73832035]->[106.71266174,10.73857975]
From 1374->1378 : [106.71266174,10.73857975]->[106.71266174,10.73857975]->[106.71201324,10.73863983]->[106.71121216,10.73875999]->[106.
71011353,10.73892975]
From 1378->1376 : [106.71011353,10.73892975]->[106.71011353,10.73892975]->[106.7075119,10.73931026]->[106.7075119,10.73934937]
From 1376->1380 : [106.7075119,10.73934937]->[106.70363617,10.73994923]->[106.70368958,10.74020767]
From 1380->1382 : [106.70368958,10.74020767]->[106.70361328,10.74022007]->[106.70401001,10.74250984]->[106.70423126,10.74417973]
From 1382->1251 : [106.70423126,10.74417973]->[106.70423126,10.74417973]->[106.70433044,10.74499035]->[106.70250702,10.74522972]->[106.
70230865,10.7453804]->[106.70217896,10.7455101]->[106.70207214,10.74571991]->[106.70201111,10.74592972]->[106.70200348,10.74627018]->
[106.70211792,10.74703026]
From 1251->4424 : [106.70211792,10.74703026]->[106.70237732,10.74864006]->[106.70246124,10.74862385]->[106.70209503,10.75481129]->[106.
70072937,10.75683022]
From 4424->1250 : [106.70072937,10.75683022]->[106.70072937,10.75683022]->[106.69921112,10.75884819]
From 1250->1253 : [106.69931793,10.75872707]->[106.69931793,10.75872707]->[106.69861603,10.75960159]->[106.69931793,10.76008224]
From 1253->1379 : [106.69918823,10.76012039]->[106.69918823,10.76012039]->[106.69972992,10.76053047]->[106.70041656,10.76099968]->[106.
69986725,10.76181984]->[106.69959259,10.76220036]->[106.69931793,10.76249027]->[106.69872284,10.7637701]->[106.69805145,10.7637701]->
[106.69786835,10.76395988]->[106.69760895,10.76432037]->[106.69657135,10.76603985]->[106.69668579,10.76610947]
From 1379->27 : [106.6966095,10.76625347]->[106.6966095,10.76625347]->[106.6966095,10.76625347]->[106.6966095,10.76625347]->[106.
69654846,10.76620579]->[106.69547272,10.76801872]->[106.69577789,10.7683506]->[106.69584656,10.76832676]
```



5) PRECOMPUTING – SAVING CACHE

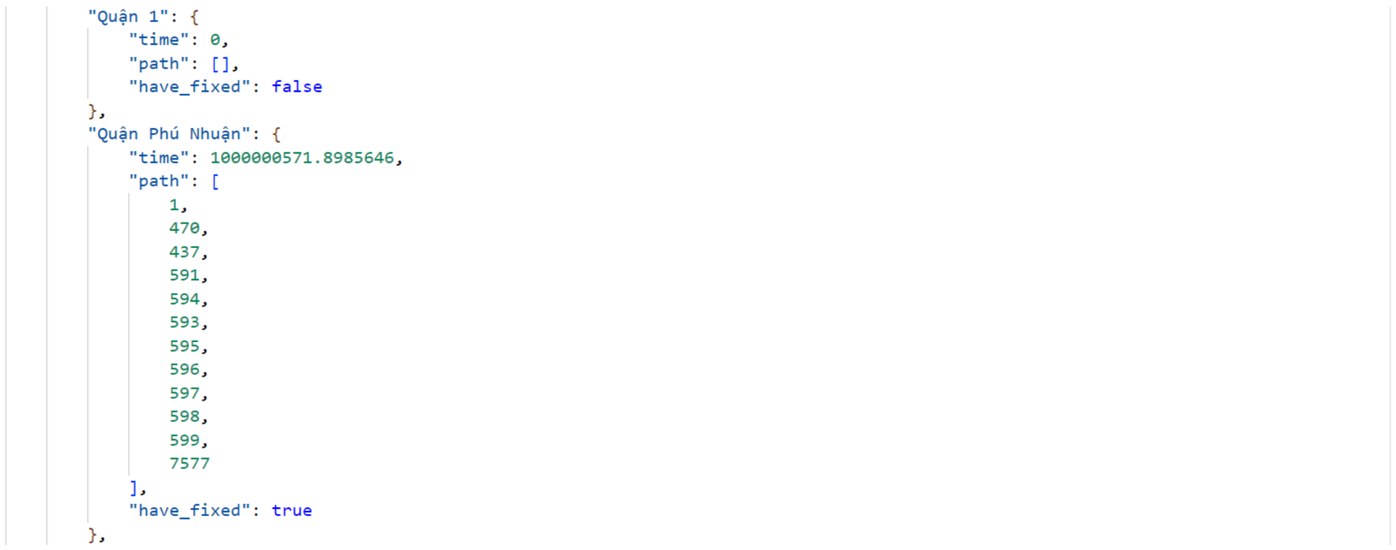
5.1) Precomputing

- As we know, caching is only sufficient if we users search for many queries as the cache will be updated dynamically. However, as we train and implement this on our local computer, it will take a bunch amount of time to pre-calculate the cache. Hence, we have to precompute it first by finding the shortest paths of all pairs using Dijkstra combined with fixed_routes_caching:

- Code:

```
def precompute_all_pairs(self):
    self.dijkstraAll()

    for u in tqdm(self.numVertices):
        for v in self.numVertices:
            self.shortest_path_fixed(u, v)

    self.save_fixed_route()
    print("Precompute all pairs successfully.")
```

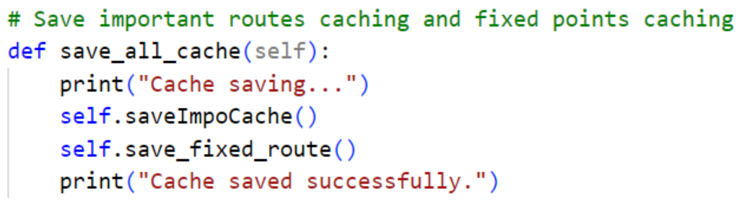
- Results after precomputing:



```
"Quận 1": {
    "time": 0,
    "path": [],
    "have_fixed": false
},
"Quận Phú Nhuận": {
    "time": 1000000571.8985646,
    "path": [
        1,
        470,
        437,
        591,
        594,
        593,
        595,
        596,
        597,
        598,
        599,
        7577
    ],
    "have_fixed": true
},
```

5.2) Saving cache after querying

- After querying, we have to save the top important cache and fixed_routes_cache for later use:



```
# Save important routes caching and fixed points caching
def save_all_cache(self):
    print("Cache saving...")
    self.saveImpoCache()
    self.save_fixed_route()
    print("Cache saved successfully.")
```

6) BIDIRECTIONAL DIJKSTRA

6.1) Overview

- This kind of algorithm is an improvement of normal Dijkstra, when we will search in two ways: from source to destination and from destination to our source. When a node is settled by both searches, we will end our algorithm.

6.2) Code

```python
def dijkstra_modified(self, source):
    dist = {i: float('inf') for i in self.numVertices}
    trace = {i: None for i in self.numVertices}
    dist[source] = 0
    priority_queue = [(0, source)]

    settled = set()

    while priority_queue:
        current_dist, current_node = heapq.heappop(priority_queue)

        if current_dist > dist[current_node]:
            continue

        settled.add(current_node)

        for neighbor in neighbors:
            if neighbor in settled:
                continue

            if new_dist < dist[neighbor]:
                dist[neighbor] = new_dist
                trace[neighbor] = current_node
                heapq.heappush(priority_queue, (new_dist, neighbor))

    return dist, settled, trace

def bidirectional_dijkstra(self, source, des):
    dist_foward, settled_foward, trace_forward = self.dijkstra_modified(source)
    dist_backward, settled_backward, trace_backward = self.dijkstra_modified(des)

    v = None
    min_dist = float('inf')

    L = [x for x in settled_foward if x in settled_backward]

    for u in L:
        if dist_foward[u] + dist_backward[u] < min_dist:
            min_dist = dist_foward[u] + dist_backward[u]
            v = u

    return min_dist, v, trace_forward, trace_backward
```

7) CONTRACTION HIERARCHY

7.1) Overview

- Why do we have to consider about Contraction Hierarchy? This is because Google's map is a hierarchical structure! This means that we can optimize our search function based on the level of importance which is the order of contraction. In this way, we will consider going on the highway rather than a crowded street, which results in better performance overall. In addition, we will base on the module of bidirectional Dijkstra for better efficiency.

- In more detail, contraction hierarchies are a speed-up method optimized to exploit the properties of graphs representing road networks. The speed-up is achieved by creating shortcuts in a preprocessing phase, which are then used during a shortest-path query to skip over "unimportant" vertices. This is based on the observation that road networks are highly hierarchical. Some intersections, such as highway junctions, are "more important" and higher up in the hierarchy than, for example, a junction leading into a dead end. Shortcuts are used to store the precomputed distance between two important junctions, allowing the algorithm to avoid considering the full path between these junctions at query time. Contraction hierarchies do not inherently know which roads humans consider "important" (e.g., highways), but they can assign importance to vertices using heuristics based on the provided graph.

7.2) Algorithm

7.2.1) Preprocessing stage

- In this stage, we will have three main steps:

+ Step 1: Order all nodes to some kind of importance (we will dive into this later).

+ Step 2: Contract each node in order.

+ Step 3: Have an overlay graph G* containing all original nodes and edges, and also all shortcut edges added during the contraction phase. Then implement the bidirectional Dijkstra on this G* graph.

- We will go for some helper functions first:

+ Count difference: We will take the importance equal to the difference between the shortcuts after removing a vertice from the graph and the initial neighbours of a node.

```
def count_edge_diff_one(self, u):
    cnt_sides = len(self.children[u]) + len(self.parents[u])
    cnt_shortcuts = self.count_shortcuts(u)
    self.edge_difference[u] = cnt_shortcuts - cnt_sides

def count_edge_diff_all(self):
    for u in self.numVertices:
        self.count_edge_diff_one(u)
```

+ Adding shortcuts: If a vertice (which is going to be removed from the graph) is on any shortest path, we will make shortcuts here and update the G* graph.

```
def add_shortcuts(self, v):
    # Incoming edges
    U = self.parents[v]
    # Outgoing edges
    W = self.children[v]

    for u in U:
        P = {}
        Pmax = -float('inf')
        for w in W:
            Pw = self.timeAll[u][v] + self.timeAll[v][w]
            P[w] = Pw
            Pmax = max(Pmax, Pw)

        dist = self.dijkstra_excluding_v(u, v, Pmax)
        for w in W:
            if dist[w] > P[w]:
                self.G_star_edges[u].append([P[w], w])
                self.G_star_shortcuts[u][w] = v
                self.G_star_children[u].add(w)
                self.G_star_parents[w].add(u)

                self.timeAll[u][w] = P[w]
                self.parents[w].add(u)
                self.children[u].add(w)
```

+ In the above code, we will follow this method of adding shortcuts:

- Notation:

+ Consider the set of all vertices with edges incoming to *v* as *U*, and the set of all vertices with incoming edges from *v* as *W*.

- We can do the following for every u in U:

1. For every node w in W, compute Pw as the cost from u to w through v, which is the sum of the edge weights w(u, v) + w(v, w).
2. Then Pmax is the maximum Pw over all w in W.
3. Perform a standard Dijkstra's shortest-path search from u on the subgraph excluding v.
4. Once a node is settled with a shortest path score greater than Pmax, we stop.
5. Then for each w, if dist(u, w) > Pw, we add a shortcut edge uw with weight Pw. If this condition doesn't hold, no shortcut is added.

+ Dijkstra excluding a specific vertice:

```
def dijkstra_excluding_v(self, start, exclude, Pmax):
    dist = {i: float('inf') for i in self.numVertices}
    dist[start] = 0
    priority_queue = [(0, start)]
    settled = set()

    while priority_queue:
        current_dist, current_node = heapq.heappop(priority_queue)

        if current_node in settled:
            continue

        settled.add(current_node)

        if current_dist > Pmax:
            break

        for neighbor in self.children[current_node]:
            if neighbor == exclude or neighbor in settled:
                continue

            new_dist = current_dist + self.timeAll[current_node][neighbor]

            if new_dist < dist[neighbor]:
                dist[neighbor] = new_dist
                heapq.heappush(priority_queue, (new_dist, neighbor))

    return dist
```

+ Remove a node after adding shortcuts:

```
def remove_node(self, v):
    # Remove all edges to and from node v
    for u in list(self.parents[v]):
        self.children[u].remove(v)
    for u in list(self.children[v]):
        self.parents[u].remove(v)

    # Clear the parents and children lists of node v
    self.parents[v].clear()
    self.children[v].clear()
```

- Now we will come to our preprocessing function, we will base it on the Lazy Update algorithm to enhance the efficiency of our program:

*** Lazy update:**

+ To use this heuristic, consider that we've already computed our initial node ordering. Now we're in the process of actually contracting each node in order by extracting the minimum from our priority queue.

+ Before contracting the next minimum node, we recompute its edge difference. If it's still the smallest in the priority queue, we proceed with contracting it. If its edge difference is no longer the minimum, we update its cost and rebalance our priority queue. We then check the next minimum node and continue this process. Performing these lazy updates results in improved query times due to a better, updated contraction order.

- Code:

```
def pre_processing(self):
    self.count_edge_diff_all()
    pq = []

    for u in self.numVertices:
        heapq.heappush(pq, (self.edge_difference[u], u))

    while pq:
        cur, u = heapq.heappop(pq)
        self.count_edge_diff_one(u)

        if pq and self.edge_difference[u] > pq[0][0]:
            heapq.heappush(pq, (self.edge_difference[u], u))
            continue

        self.add_shortcuts(u)
        self.remove_node(u)
        self.contracted_order[u] = len(self.contracted_order) + 1

    print("Preprocessing done!")
```

7.2.2) Querying stage

- For this stage, assume that we do the preprocessing first, then for random u (source) and v (destination), we will run bidirectional Dijkstra from both directions and then for all vertices in the set L, which are all nodes that are both settled by the bidirectional Dijkstra, we will choose the one that has the shortest path compared to others.

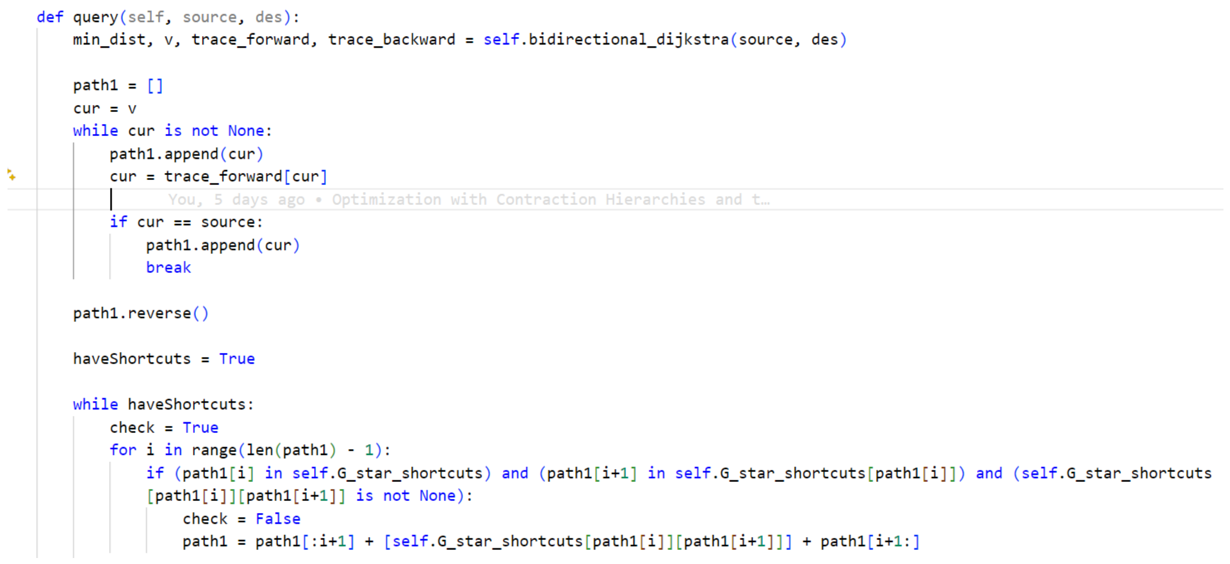
- Notes that the Dijkstra this time will only consider the nodes that have higher contracted order and we will save any shortcuts between two vertices.

- Code:

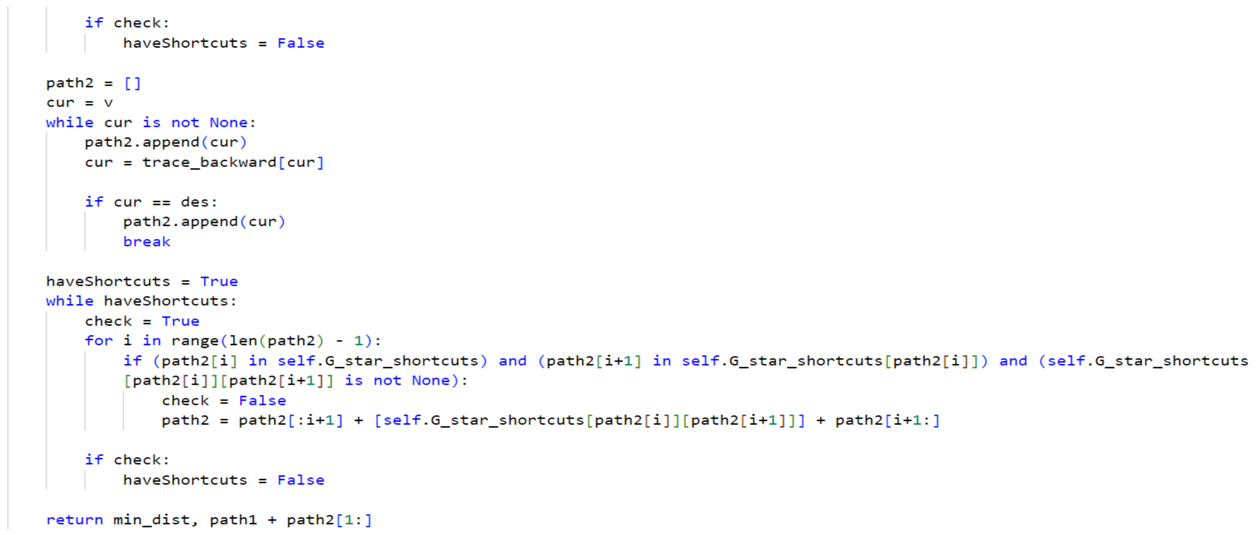
```python
def dijkstra_modified(self, source, forward):
    dist = {i: float('inf') for i in self.numVertices}
    trace = {i: None for i in self.numVertices}
    dist[source] = 0
    priority_queue = [(0, source)]

    settled = set()

    while priority_queue:
        current_dist, current_node = heapq.heappop(priority_queue)

        if current_dist > dist[current_node]:
            continue

        settled.add(current_node)

        if forward:
            neighbors = self.G_star_children[current_node]
        else:
            neighbors = self.G_star_parents[current_node]

        for neighbor in neighbors:
            if neighbor in settled:
                continue

            if self.contracted_order[neighbor] <= self.contracted_order[current_node]:
                continue

            if forward:
                new_dist = current_dist + self.timeAll[current_node][neighbor]
            else:
                new_dist = current_dist + self.timeAll[neighbor][current_node]

            if new_dist < dist[neighbor]:
                dist[neighbor] = new_dist
                trace[neighbor] = current_node
                heapq.heappush(priority_queue, (new_dist, neighbor))

    return dist, settled, trace
```

```python
def bidirectional_dijkstra(self, source, des):
    dist_foward, settled_foward, trace_forward = self.dijkstra_modified(source, forward=True)
    dist_backward, settled_backward, trace_backward = self.dijkstra_modified(des, forward=False)

    v = None
    min_dist = float('inf')

    L = [x for x in settled_foward if x in settled_backward]

    for u in L:
        if dist_foward[u] + dist_backward[u] < min_dist:
            min_dist = dist_foward[u] + dist_backward[u]
            v = u

    return min_dist, v, trace_forward, trace_backward
```

- Query functions:



```python
def query(self, source, des):
    min_dist, v, trace_forward, trace_backward = self.bidirectional_dijkstra(source, des)

    path1 = []
    cur = v
    while cur is not None:
        path1.append(cur)
        cur = trace_forward[cur]
                You, 5 days ago • Optimization with Contraction Hierarchies and t…
        if cur == source:
            path1.append(cur)
            break

    path1.reverse()

    haveShortcuts = True

    while haveShortcuts:
        check = True
        for i in range(len(path1) - 1):
            if (path1[i] in self.G_star_shortcuts) and (path1[i+1] in self.G_star_shortcuts[path1[i]]) and (self.G_star_shortcuts
            [path1[i]][path1[i+1]] is not None):
                check = False
                path1 = path1[:i+1] + [self.G_star_shortcuts[path1[i]][path1[i+1]]] + path1[i+1:]
```



```python
        if check:
            haveShortcuts = False

    path2 = []
    cur = v
    while cur is not None:
        path2.append(cur)
        cur = trace_backward[cur]

        if cur == des:
            path2.append(cur)
            break

    haveShortcuts = True
    while haveShortcuts:
        check = True
        for i in range(len(path2) - 1):
            if (path2[i] in self.G_star_shortcuts) and (path2[i+1] in self.G_star_shortcuts[path2[i]]) and (self.G_star_shortcuts
            [path2[i]][path2[i+1]] is not None):
                check = False
                path2 = path2[:i+1] + [self.G_star_shortcuts[path2[i]][path2[i+1]]] + path2[i+1:]

        if check:
            haveShortcuts = False

    return min_dist, path1 + path2[1:]
```

7.3) Time complexity and Space search

- Time complexity:

+ Initialization: $O(V + E)$.

+ Preprocessing: $O(V \log V + VE)$. This includes running bidirectional Dijkstra's algorithm on the modified graph with shortcuts.

+ Query: $O(E + V \log V)$.

→ Overall time complexity: $O(V \log V + VE)$.

- Space complexity:

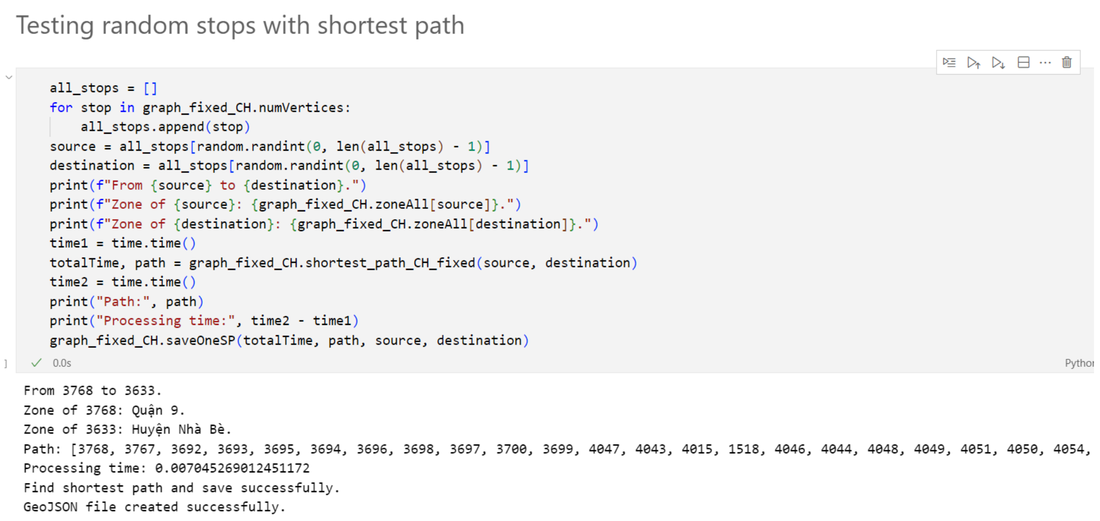
+ Graph storage: $O(V + E)$.

+ Additional data structures: $O(V + E)$. This includes dictionaries for shortcuts, edges, parents, children, edge differences, and contracted order.

→ Overall space complexity: $O(V + E)$.

7.4) Results of querying

- Using Contraction Hierarchy with caching and fixed_routes:

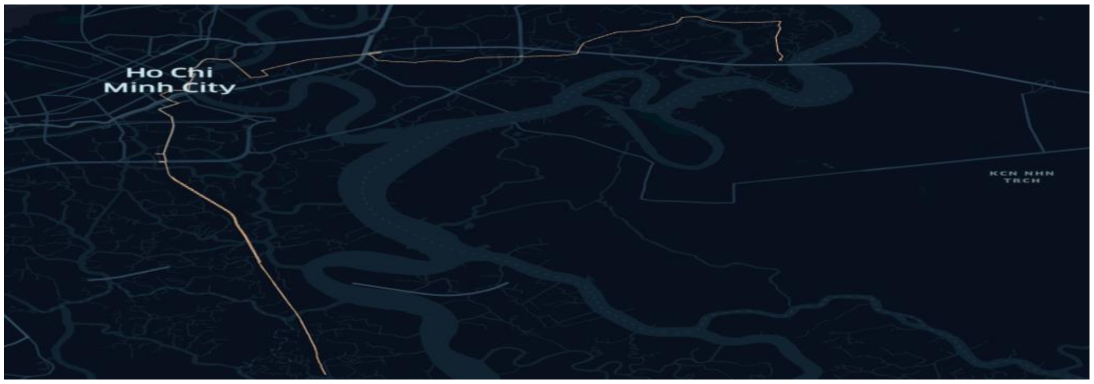


Testing random stops with shortest path

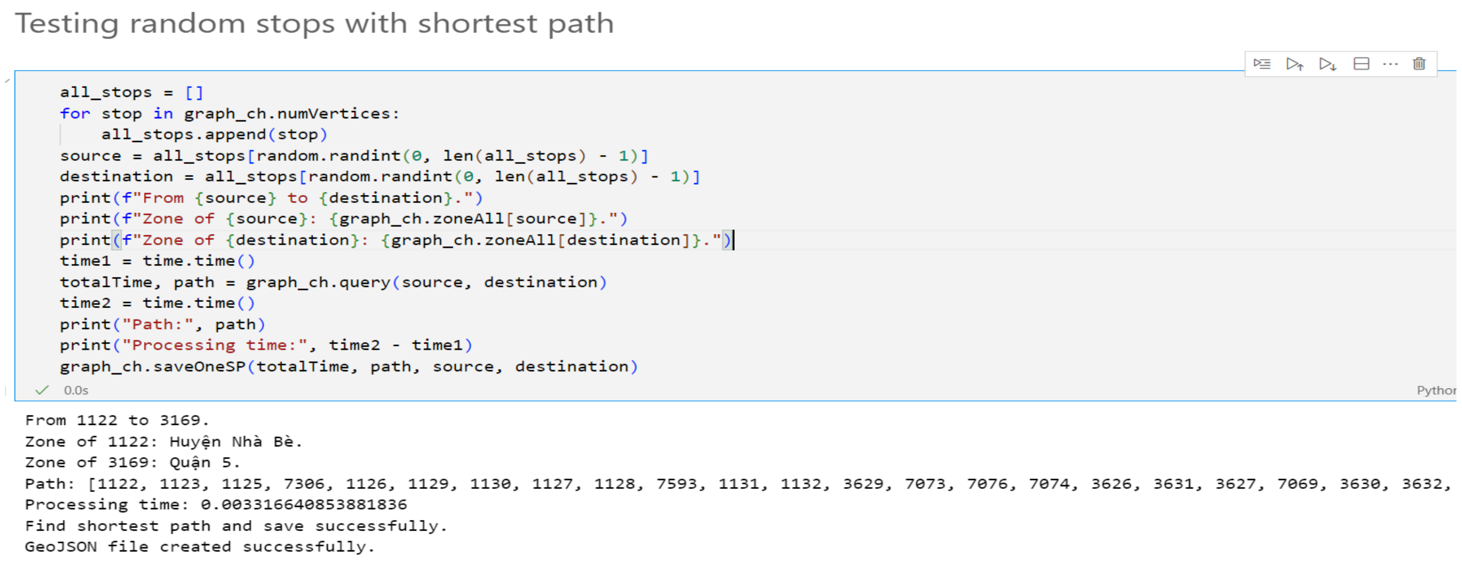
```python
all_stops = []
for stop in graph_fixed_CH.numVertices:
    all_stops.append(stop)
source = all_stops[random.randint(0, len(all_stops) - 1)]
destination = all_stops[random.randint(0, len(all_stops) - 1)]
print(f"From {source} to {destination}.")
print(f"Zone of {source}: {graph_fixed_CH.zoneAll[source]}.")
print(f"Zone of {destination}: {graph_fixed_CH.zoneAll[destination]}.")
time1 = time.time()
totalTime, path = graph_fixed_CH.shortest_path_CH_fixed(source, destination)
time2 = time.time()
print("Path:", path)
print("Processing time:", time2 - time1)
graph_fixed_CH.saveOneSP(totalTime, path, source, destination)
```

✓ 0.0s

```
From 3768 to 3633.
Zone of 3768: Quận 9.
Zone of 3633: Huyện Nhà Bè.
Path: [3768, 3767, 3692, 3693, 3695, 3694, 3696, 3698, 3697, 3700, 3699, 4047, 4043, 4015, 1518, 4046, 4044, 4048, 4049, 4051, 4050, 4054,
Processing time: 0.007045269012451172
Find shortest path and save successfully.
GeoJSON file created successfully.
```



- Using normal Contraction Hierarchy without fixed_routes caching:

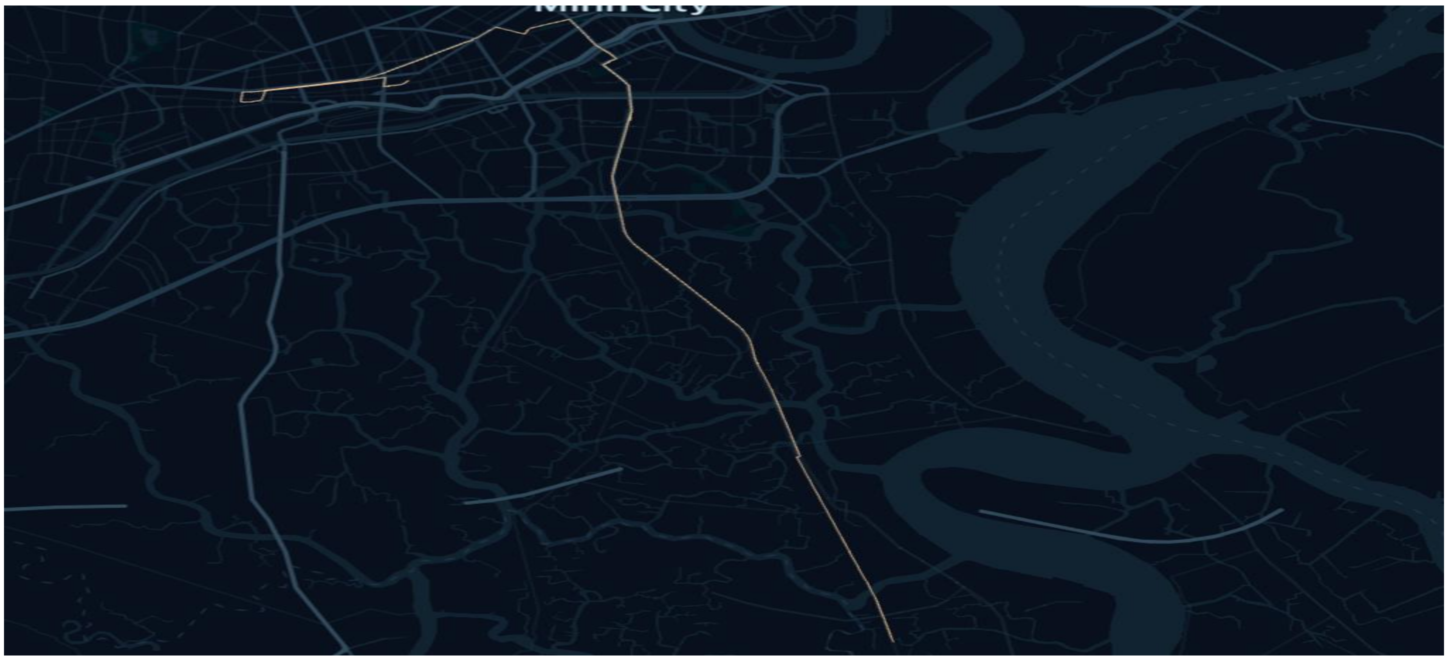


Testing random stops with shortest path

```python
all_stops = []
for stop in graph_ch.numVertices:
    all_stops.append(stop)
source = all_stops[random.randint(0, len(all_stops) - 1)]
destination = all_stops[random.randint(0, len(all_stops) - 1)]
print(f"From {source} to {destination}.")
print(f"Zone of {source}: {graph_ch.zoneAll[source]}.")
print(f"Zone of {destination}: {graph_ch.zoneAll[destination]}.")
time1 = time.time()
totalTime, path = graph_ch.query(source, destination)
time2 = time.time()
print("Path:", path)
print("Processing time:", time2 - time1)
graph_ch.saveOneSP(totalTime, path, source, destination)
```

✓ 0.0s

```
From 1122 to 3169.
Zone of 1122: Huyện Nhà Bè.
Zone of 3169: Quận 5.
Path: [1122, 1123, 1125, 7306, 1126, 1129, 1130, 1127, 1128, 7593, 1131, 1132, 3629, 7073, 7076, 7074, 3626, 3631, 3627, 7069, 3630, 3632,
Processing time: 0.003316640853881836
Find shortest path and save successfully.
GeoJSON file created successfully.
```



```
Shortest path from 1122 to 3169:
1122->1123->1125->7306->1126->1129->1130->1127->1128->7593->1131->1132->3629->7073->7076->7074->3626->3631->3627->7069->3630->3632->3633
->1244->1246->1245->1247->1249->1251->4424->1250->1253->1379->3080->7274->7275->86->89->385->387->3182->433->434->436->435->438->8->464-
>470->437->439->440->1981->3166->3169          You, 2 minutes ago • Uncommitted changes
Total time: 4079.0087766478655 seconds
From 1122->1123 : [106.73737335,10.62600231]->[106.73737335,10.62600231]->[106.73737335,10.62600231]->[106.73737335,10.62600231]->[106.
73737335,10.62600231]->[106.73728943,10.62596989]->[106.73709869,10.62664986]->[106.73696136,10.6272831]->[106.73682404,10.6279211]->
[106.73668671,10.62855911]->[106.73653412,10.62920284]->[106.73638916,10.62984657]->[106.73623657,10.63050461]->[106.73610687,10.
63116932]
From 1123->1125 : [106.73610687,10.63116932]->[106.73610687,10.63116932]->[106.73610687,10.63116932]->[106.73610687,10.63116932]->[106.
7359848,10.63164806]->[106.73583984,10.63212299]->[106.73569489,10.63277721]->[106.73554993,10.63343048]->[106.7352829,10.63462734]->
[106.73536682,10.63463783]
From 1125->7306 : [106.73536682,10.63463783]->[106.73536682,10.63463783]->[106.73536682,10.63463783]->[106.73536682,10.63463783]->[106.
7352829,10.63462162]->[106.73498535,10.63583946]->[106.73460388,10.63707352]->[106.73441315,10.63768005]->[106.73422241,10.63828659]->
[106.73405457,10.6388979]->[106.73388672,10.6395092]->[106.73394775,10.63951969]
From 7306->1126 : [106.73394775,10.63951969]->[106.73394775,10.63951969]->[106.73394775,10.63951969]->[106.73394775,10.63951969]->[106.
73388672,10.63949871]->[106.73371124,10.6400156]->[106.73348999,10.64053726]->[106.73313904,10.64155006]->[106.73287964,10.64236164]->
[106.73262024,10.64317322]->[106.73267365,10.64318943]
From 1126->1129 : [106.73267365,10.64318943]->[106.73199463,10.64512444]->[106.73153687,10.64649487]->[106.73138428,10.64717007]
From 1129->1130 : [106.73138428,10.64717007]->[106.73130035,10.64732838]->[106.73119354,10.64762878]->[106.73097229,10.64828777]->[106.
73083496,10.64882565]->[106.73066711,10.64933681]->[106.73052979,10.64987469]
From 1130->1127 : [106.73052979,10.64987469]->[106.73033905,10.65040112]->[106.73011017,10.65111828]->[106.72976685,10.65203571]->[106.
72951508,10.65294266]->[106.72941589,10.65339088]
From 1127->1128 : [106.72941589,10.65339088]->[106.72941589,10.65339088]->[106.72916412,10.65400219]->[106.72890472,10.65478802]->[106.
72853851,10.65595818]->[106.72829437,10.65676498]->[106.72820282,10.6571331]
From 1128->7593 : [106.72820282,10.6571331]->[106.72820282,10.6571331]->[106.72820282,10.6571331]->[106.72820282,10.6571331]->[106.
72815704,10.65711784]->[106.72789001,10.65795326]->[106.72762299,10.65878963]->[106.72736359,10.65962696]->[106.72709656,10.66046524]->
[106.72720337,10.66050339]
From 7593->1131 : [106.72720337,10.66050339]->[106.72720337,10.66050339]->[106.72720337,10.66050339]->[106.72720337,10.66050339]->[106.
72709656,10.66047096]->[106.72685242,10.6612196]->[106.72660828,10.66196823]->[106.72637177,10.66271114]->[106.72613525,10.66345501]->
[106.72621155,10.66346455]
```

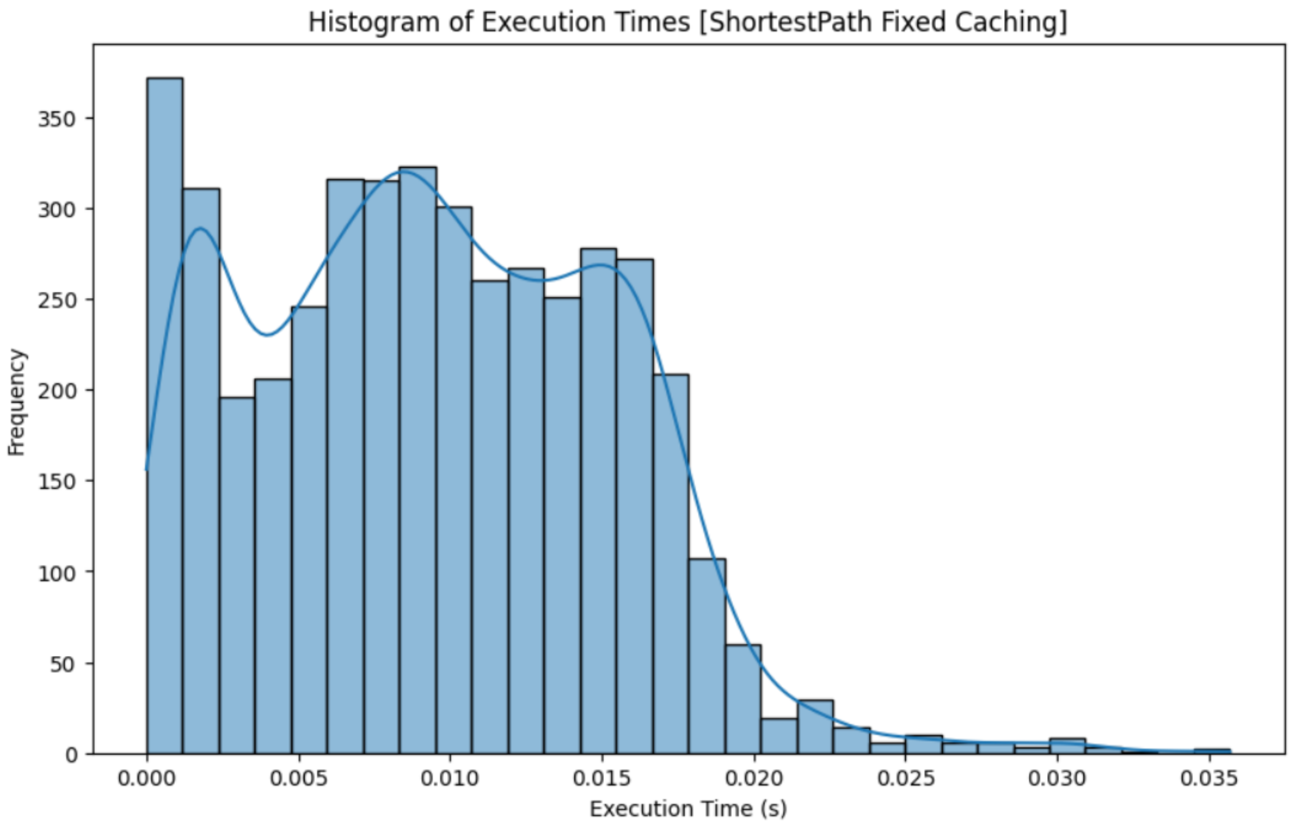
8) PERFORMANCE [JUPYTER]

8.1) Benchmark A_Star_Cache with Fixed_Routes

- Average from one stop to all other stops:

```
100%|██████████| 4397/4397 [00:41<00:00, 105.00it/s]
c:\Users\USER\anaconda3\Lib\site-packages\seaborn\_oldcore.py:1119: FutureWarning: use_inf_as_na option is deprecated and will be removed i
  with pd.option_context('mode.use_inf_as_na', True):
Fastest time: 0.000110626220703125
Slowest time: 0.03569459915161133
Average time: 0.00946417992457168
```

- Visualization:



8.2) Benchmark A_Star_ without Fixed_Routes

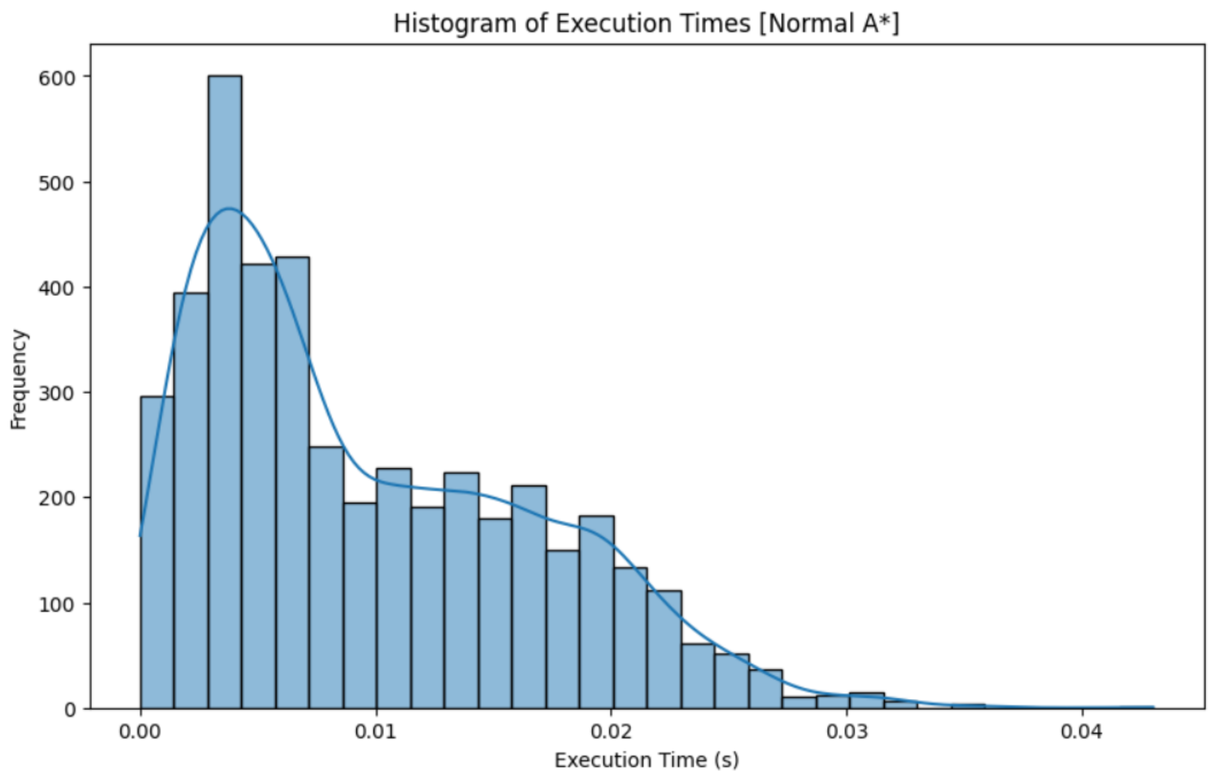
- Average from one stop to all other stops:

```
normalAStarBenchMark = A_Star("vars.json", "stops.json", "paths.json")
```

Python

```
100%|██████████| 10243/10243 [00:00<00:00, 5006089.01it/s]
100%|██████████| 4397/4397 [00:01<00:00, 3103.36it/s]
100%|██████████| 297/297 [00:02<00:00, 109.86it/s]


100%|██████████| 4397/4397 [00:42<00:00, 102.46it/s]
Fastest time: 0.0005030632019042969
Slowest time: 0.04304099082946777
Average time: 0.009690641787530943
```

- Visualization:

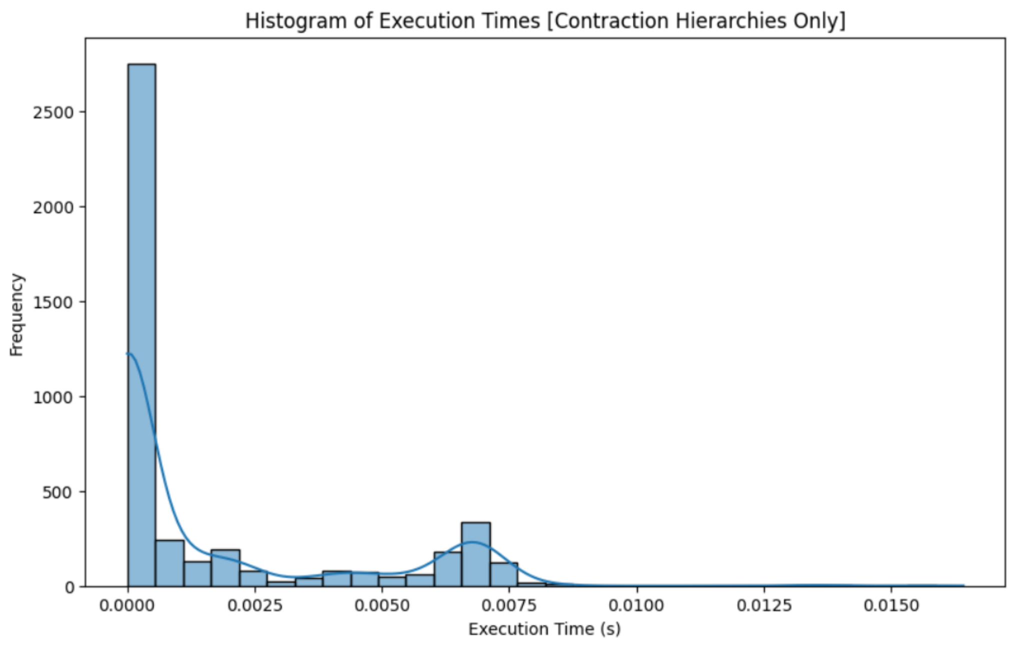


8.3) Benchmark Contraction Hierarchy Only

- Average from one stop to all other stops:

```
100%|██████████| 4397/4397 [00:07<00:00, 600.92it/s]
Fastest time: 7.748603820800781e-05
Slowest time: 0.016424179077148438
Average time: 0.001652893815117585
```

- Visualization:

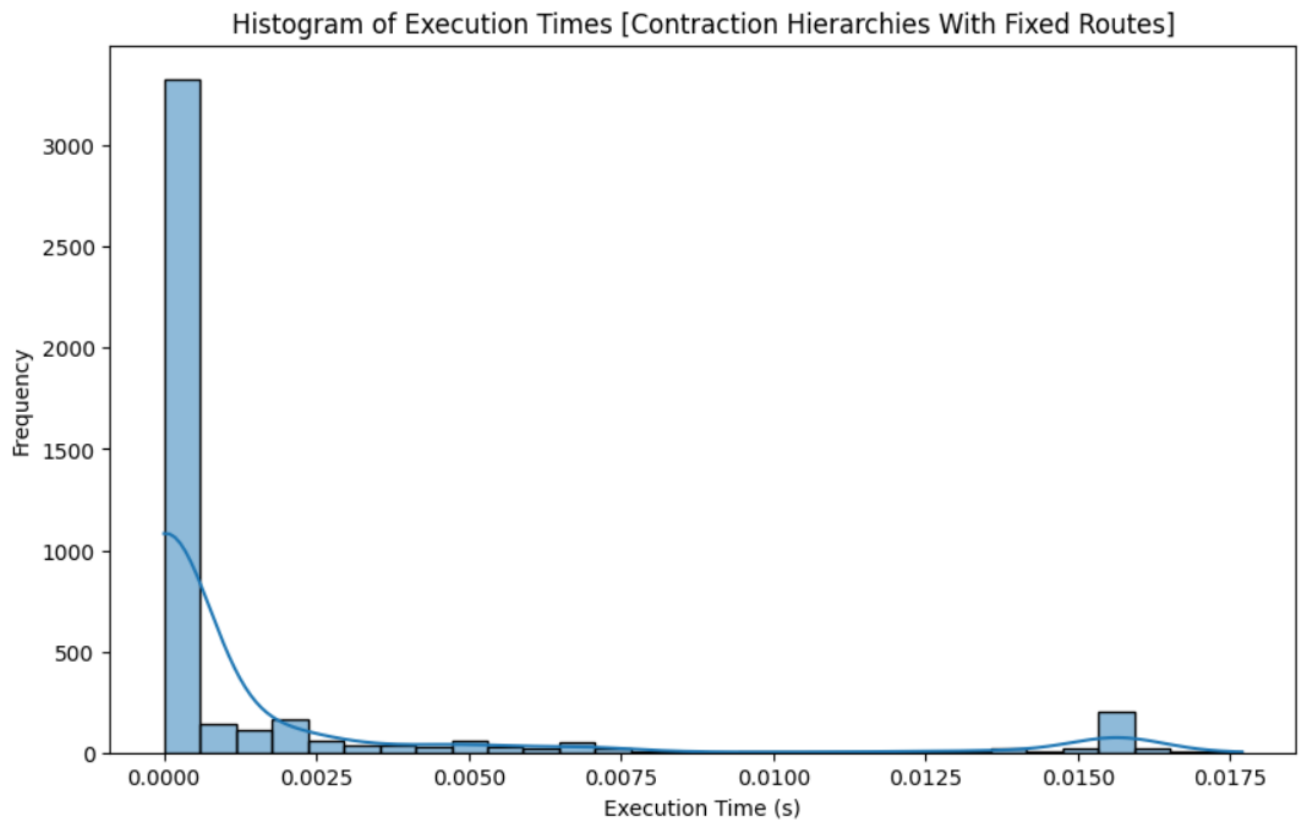


8.4) Benchmark Contraction Hierarchy with Fixed_Routes Caching

- Average from one stop to all other stops:

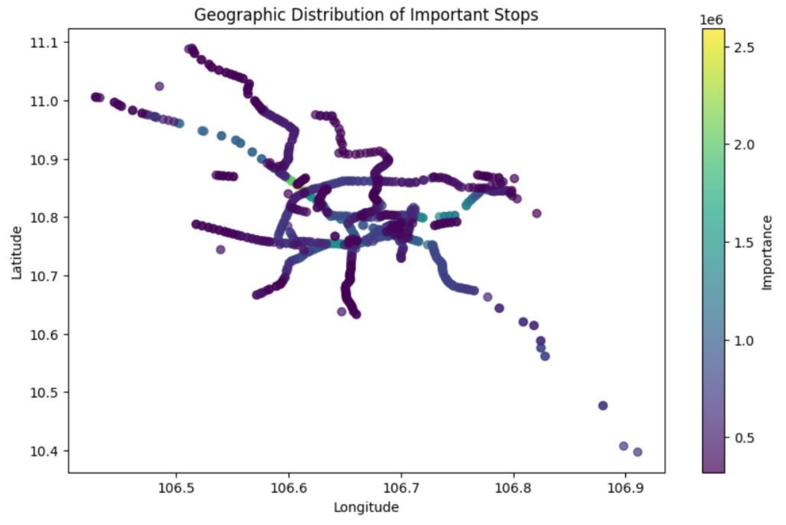
```
100%|██████████| 4397/4397 [00:06<00:00, 631.37it/s]
c:\Users\USER\anaconda3\Lib\site-packages\seaborn\_oldcore.py:1119: FutureWarning: use_inf_as_na option is deprecated and will be removed
  with pd.option_context('mode.use_inf_as_na', True):
Fastest time: 5.14984130859375e-05
Slowest time: 0.017695903778076172
Average time: 0.0015751560736927954
```

- Visualization:

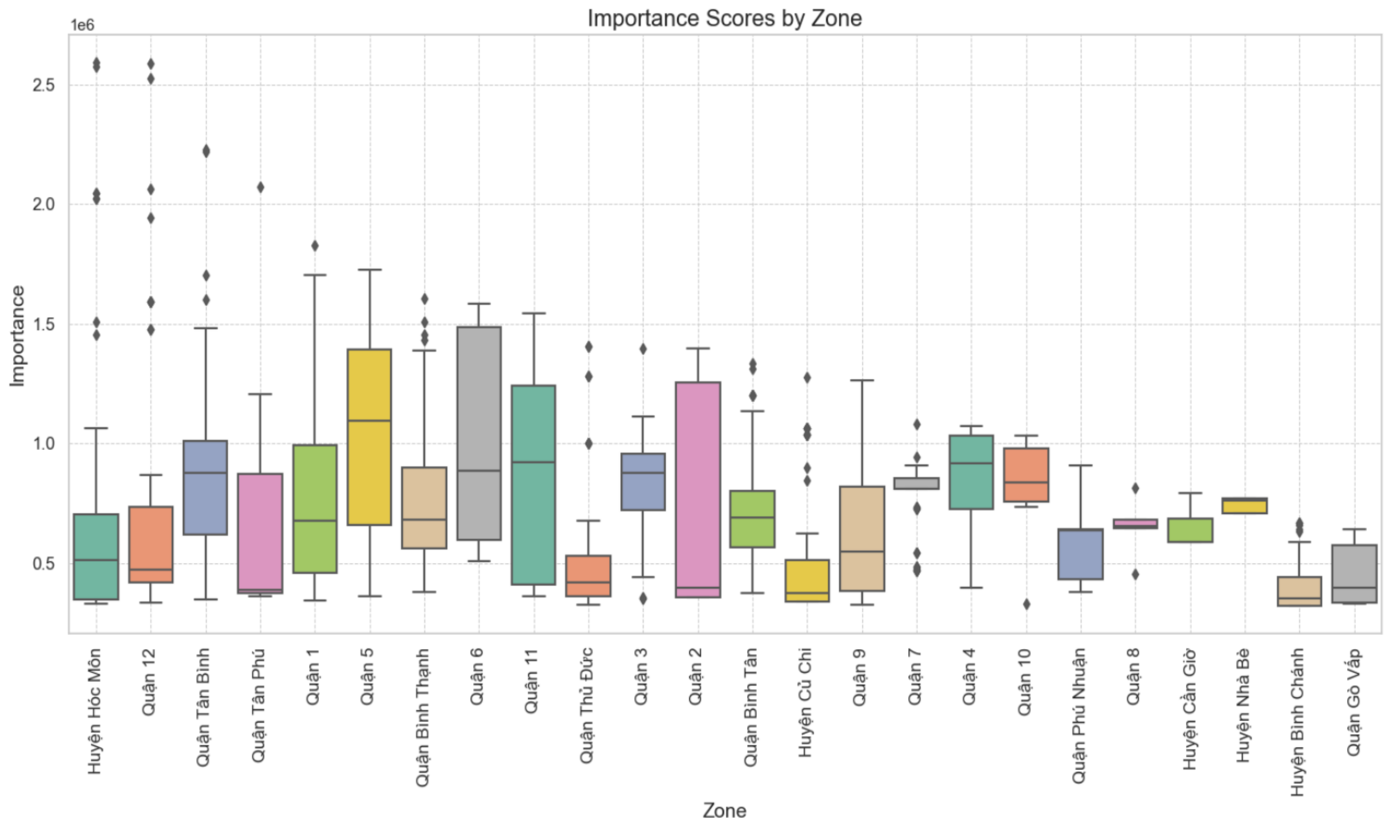


8.5) Analysis of important segments

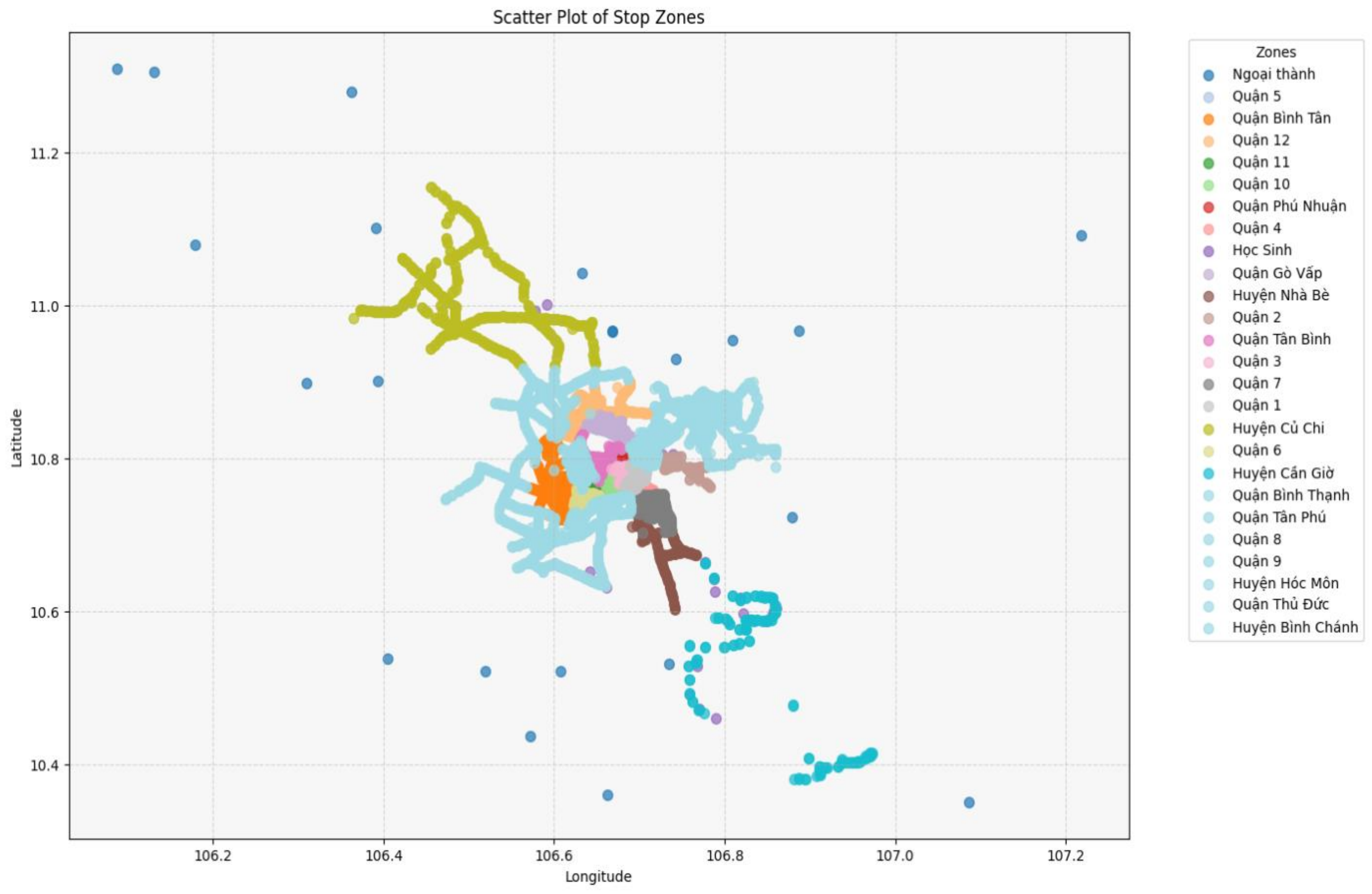
- Distribution of important stops:



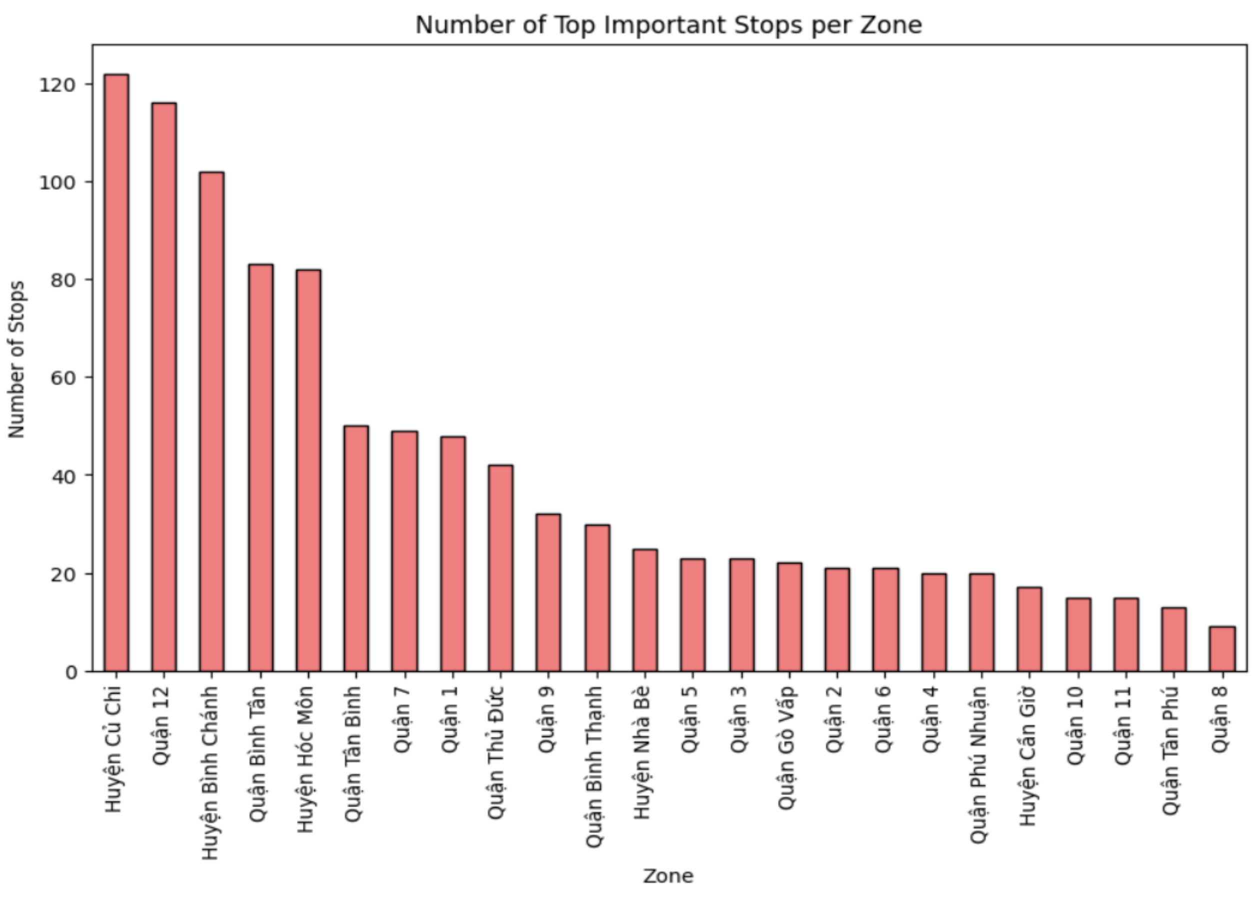
- Important stops by zone:



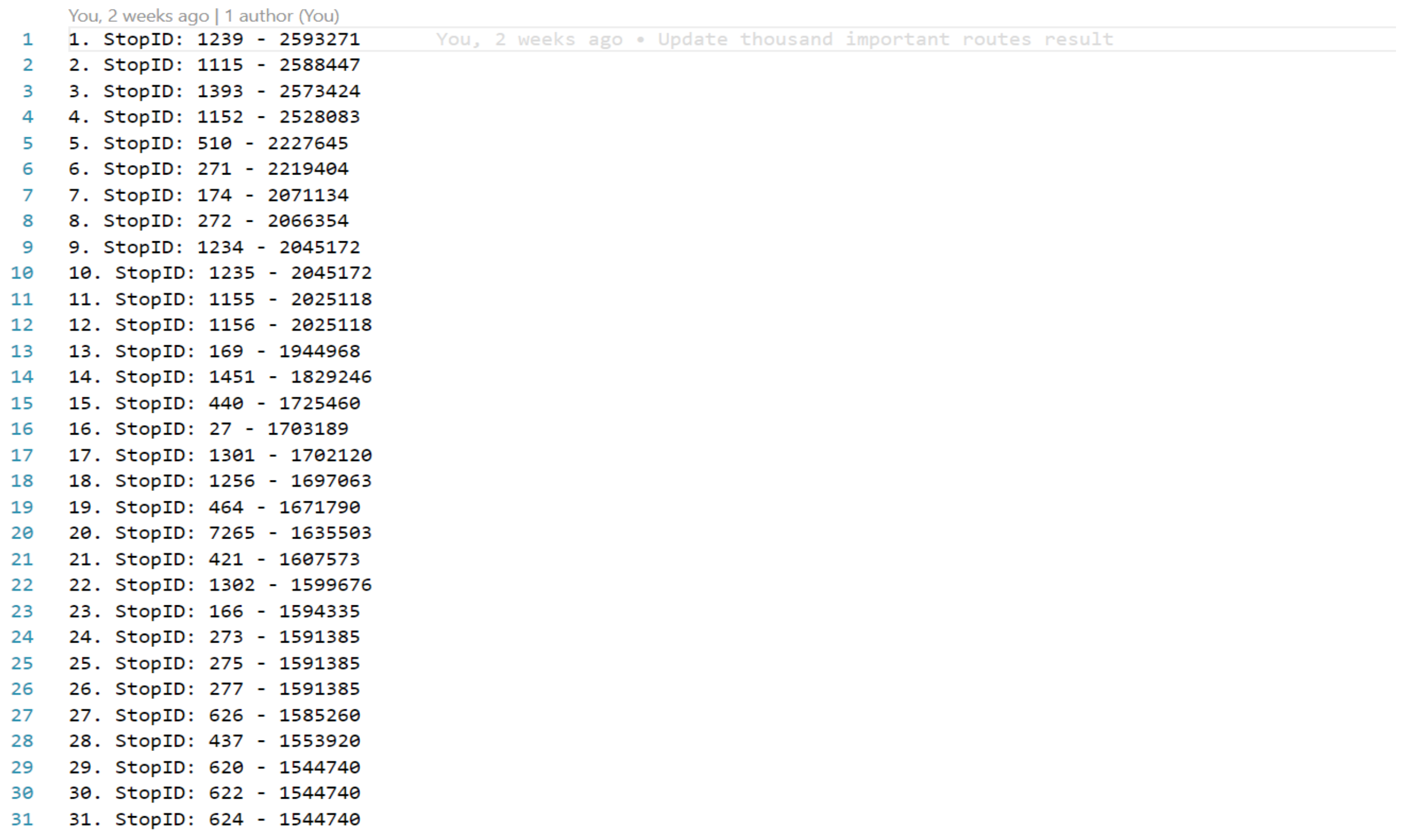
- Scatter Plot of Stop Zones:



- Distribution of top important stops per zone:



- Top important stops:


```
You, 2 weeks ago | 1 author (You)
1. StopID: 1239 - 2593271        You, 2 weeks ago • Update thousand important routes result
2. StopID: 1115 - 2588447
3. StopID: 1393 - 2573424
4. StopID: 1152 - 2528083
5. StopID: 510 - 2227645
6. StopID: 271 - 2219404
7. StopID: 174 - 2071134
8. StopID: 272 - 2066354
9. StopID: 1234 - 2045172
10. StopID: 1235 - 2045172
11. StopID: 1155 - 2025118
12. StopID: 1156 - 2025118
13. StopID: 169 - 1944968
14. StopID: 1451 - 1829246
15. StopID: 440 - 1725460
16. StopID: 27 - 1703189
17. StopID: 1301 - 1702120
18. StopID: 1256 - 1697063
19. StopID: 464 - 1671790
20. StopID: 7265 - 1635503
21. StopID: 421 - 1607573
22. StopID: 1302 - 1599676
23. StopID: 166 - 1594335
24. StopID: 273 - 1591385
25. StopID: 275 - 1591385
26. StopID: 277 - 1591385
27. StopID: 626 - 1585260
28. StopID: 437 - 1553920
29. StopID: 620 - 1544740
30. StopID: 622 - 1544740
31. StopID: 624 - 1544740
```

9) PROGRAM ANALYSIS

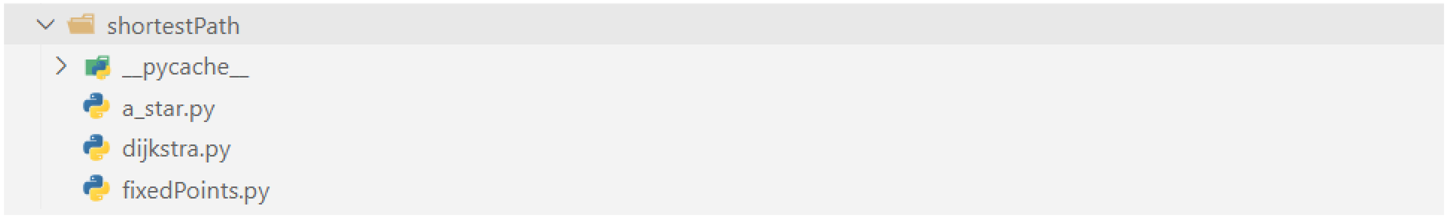
9.1) Module Distribution

- As the two main algorithms used by Google's Map are Contraction Hierarchy and A* combined with two techniques (caching and fixed_routes), our program will focus on this and be split into different modules for better implementation.

9.2) Details

9.2.1) Shortest Path Module

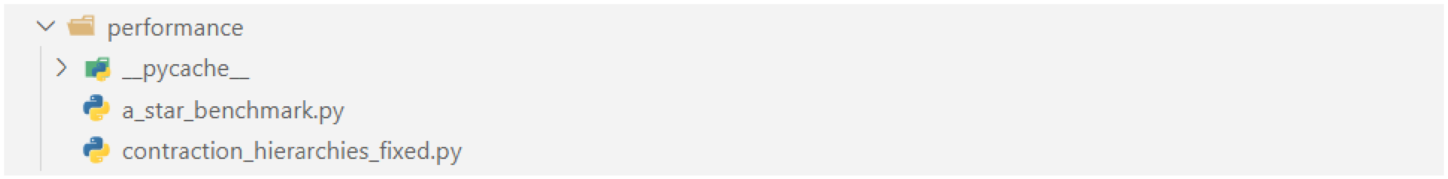
- We will have three main files here, fixedPoints.py will be an A_Star algorithm using fixed_routes technique.



```
shortestPath
  __pycache__
  a_star.py
  dijkstra.py
  fixedPoints.py
```

9.2.2) Performance Module

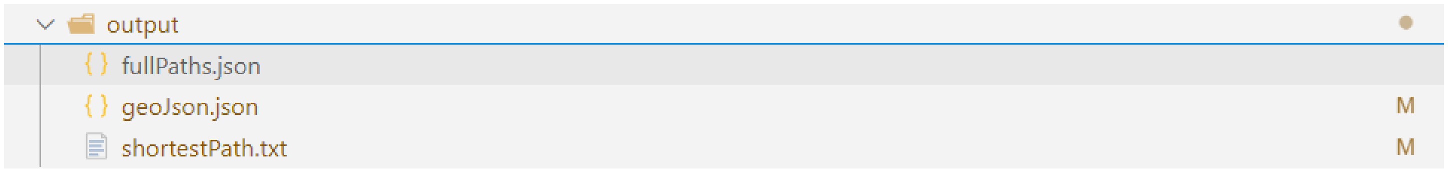
- This module will focus on testing the performance of some algorithms without any comments:



```
performance
  __pycache__
  a_star_benchmark.py
  contraction_hierarchies_fixed.py
```

9.2.3) Output Module

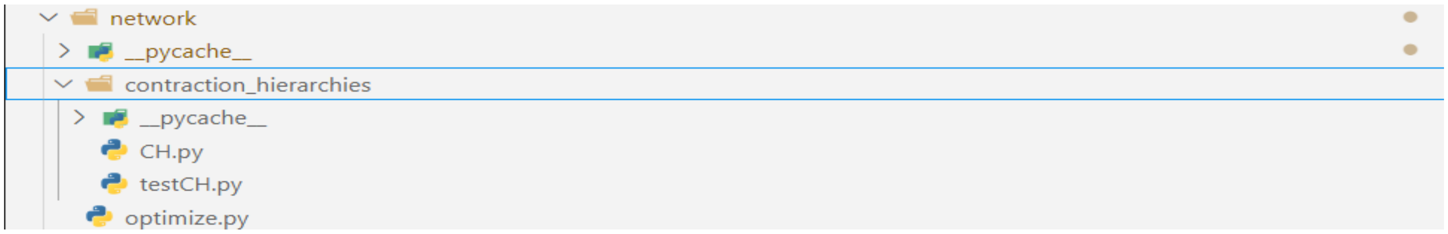
- Generating shortest path in txt and Json:



```
output
  fullPaths.json
  geoJson.json          M
  shortestPath.txt      M
```

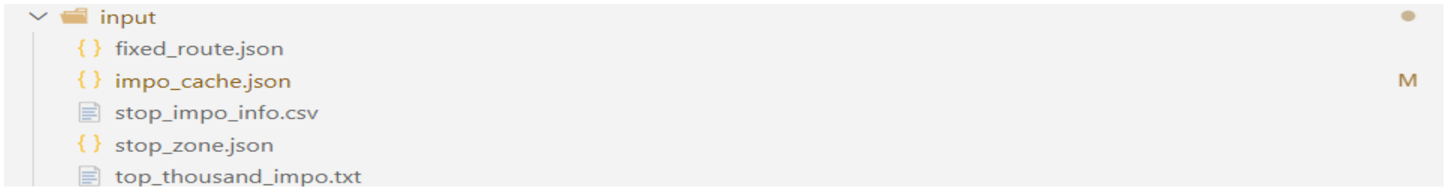
9.2.4) Network Module

- This will store our graph, including the Contraction Hierarchy technique as CH is a graph structure optimisation:



```
network
  __pycache__
  contraction_hierarchies
    __pycache__
    CH.py
    testCH.py
  optimize.py
```

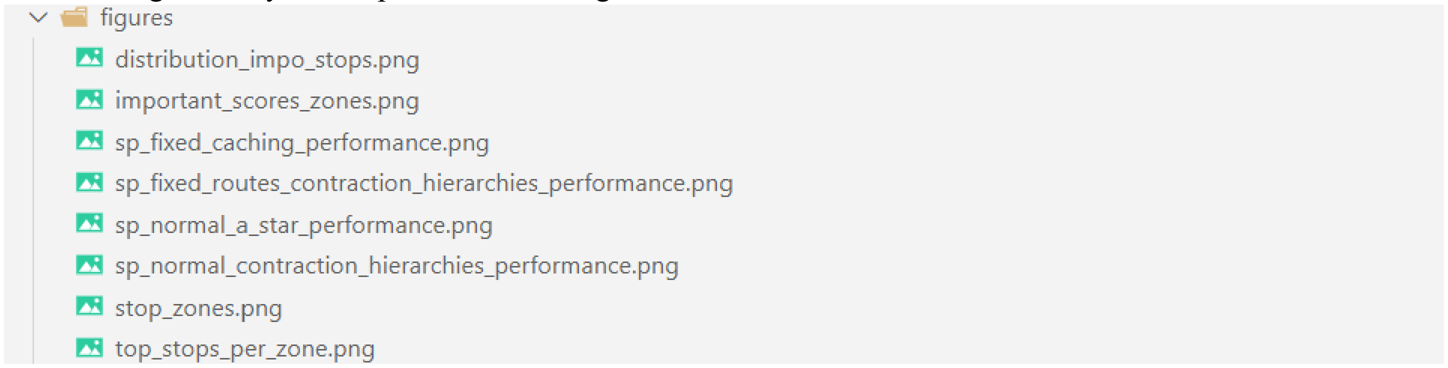
9.2.5) Input Module



```
input
  fixed_route.json
  impo_cache.json       M
  stop_impo_info.csv
  stop_zone.json
  top_thousand_impo.txt
```

9.2.6) Figures

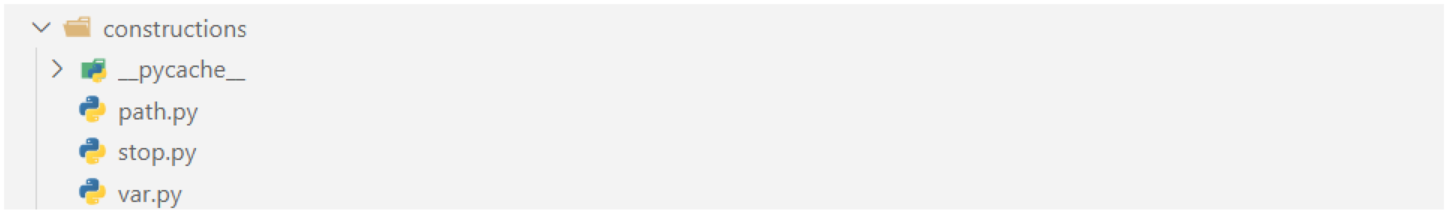
- Containing all analysis and performance images:



```
figures
    distribution_impo_stops.png
    important_scores_zones.png
    sp_fixed_caching_performance.png
    sp_fixed_routes_contraction_hierarchies_performance.png
    sp_normal_a_star_performance.png
    sp_normal_contraction_hierarchies_performance.png
    stop_zones.png
    top_stops_per_zone.png
```

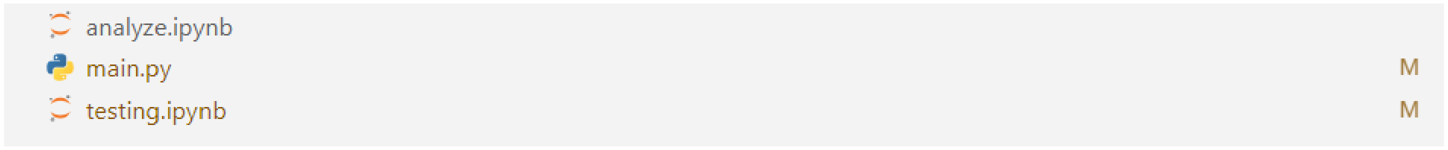
9.2.7) Construction Module

- These files are used to construct our graph:



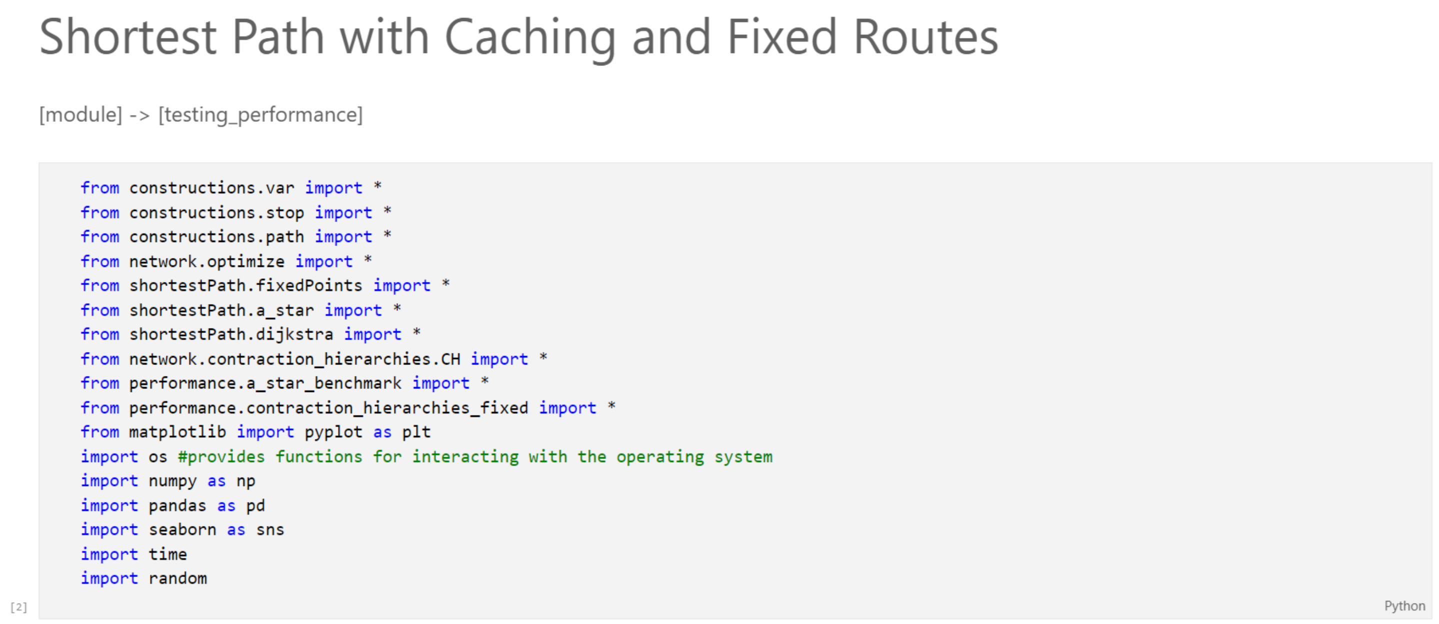
```
constructions
  > __pycache__
    path.py
    stop.py
    var.py
```

9.2.8) Main and Testing



```
analyze.ipynb
main.py                M
testing.ipynb          M
```

+ For testing.ipynb, this is used for performance benchmark:



Shortest Path with Caching and Fixed Routes

[module] -> [testing_performance]

```python
from constructions.var import *
from constructions.stop import *
from constructions.path import *
from network.optimize import *
from shortestPath.fixedPoints import *
from shortestPath.a_star import *
from shortestPath.dijkstra import *
from network.contraction_hierarchies.CH import *
from performance.a_star_benchmark import *
from performance.contraction_hierarchies_fixed import *
from matplotlib import pyplot as plt
import os #provides functions for interacting with the operating system
import numpy as np
import pandas as pd
import seaborn as sns
import time
import random
```

[2] Python

+ For analyse.ipynb, this is used to generate analysis about our graph

+ For the main.py, we can do random algorithms and techniques to test the shortest path in our graph.

10) REFERENCES

10.1) A_Star

1. Russell, Stuart J. (2018). Artificial intelligence a modern approach. Norvig, Peter (4th ed.). Boston: Pearson. ISBN 978-0134610993. OCLC 1021874142.

2. Delling, D.; Sanders, P.; Schultes, D.; Wagner, D. (2009). "Engineering Route Planning Algorithms". Algorithmics of Large and Complex Networks: Design, Analysis, and Simulation. Lecture Notes in Computer Science. Vol. 5515. Springer. pp. 117–139. doi:10.1007/978-3-642-02094-0_7. ISBN 978-3-642-02093-3.

10.2) Fixed_Routes Caching

1. Efficient Partitioning of Road Networks (research.google)

10.3) Contraction Hierarchy

1. Abraham, Ittai, et al. “Highway dimension, shortest paths, and provably efficient algorithms.” Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete Algorithms. Society for Industrial and Applied Mathematics, 2010.
2. Bast, Hannah, et al. “Route planning in transportation networks.” Algorithm engineering. Springer, Cham, 2016. 19-80.
3. Bast, Hannah. Lecture Notes from “Efficient Route Planning” . University of Freiburg, Freiburg Im Breisgau, Germany, 2012, ad-wiki.informatik.unifreiburg.de/teaching/EfficientRoutePlanningSS2012.
4. Bauer, Reinhard, et al. “Preprocessing speed-up techniques is hard.” International Conference on Algorithms and Complexity. Springer, Berlin, Heidelberg, 2010.