UNIVERSITY OF SCIENCE,

HO CHI MINH

DIJKSTRA's APPLICATION IN BUSMAP AND ITS OPTIMIZATION USING DIFFERENT ALGORITHMS FOR IMPROVEMENTS

Floyd Warshall Algorithm – Bellman Ford Algorithm - D'Esopo-Pape Algorithm

TECHNICAL REPORT
W10

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: 02/02

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

2024

Contents

I) Comments and Approach

A) Comments

- As all the information has been provided in the previous reports, this report will only add information about the time for counting the most important stops, running all pairs and the average, fastest, and lowest time to run the shortest path for one vertice.

- Also, this report will discuss the optimisation of Dijkstra's algorithm, the implementation of some of the other techniques and give evidence as to why to run “all pairs shortest path”, Dijkstra’s algorithm seems to be the fastest way to implement in terms of time complexity and space complexity.

- We can see that if we just need to find the shortest path from one vertice to another vertices, a lot of algorithms will be suitable and can run faster than Dijkstra’s algorithm. However, this is the APSP problem (all pairs shortest path), therefore, for some algorithms, the time complexity is really big and we may have to wait for hours to see the result. Therefore, I will mention some of the algorithms that are suitable for this problem, but I won’t show all the results as It runs slow. For the faster one (D’Esopo-Pape Algorithm), I will show all of the relevant information and the result.

B) Approach

1) Floyd Warshall Algorithm

- It is a dynamic programming algorithm used to find the shortest paths between all pairs of vertices in a weighted graph.

- The algorithm initialises a matrix where each cell represents the shortest distance between two vertices. Initially, this matrix is filled with the weights of the edges between adjacent vertices, and infinity is used to represent the absence of a direct edge between vertices.

- The algorithm then iterates through all pairs of vertices and considers whether there exists a shorter path between them through an intermediate vertex. For each pair of vertices (i, j), it checks if the path from vertex i to vertex j can be improved by going through each intermediate vertex k. If the path from i to j through k is shorter than the current shortest path, the distance matrix is updated with the new shortest distance.

- However, the time complexity is $O(V^3)$ and recommend graph size is small, therefore It runs slowly in the graph.

2) Bellman Ford Algorithm

- It is a dynamic programming algorithm used to find the shortest path from a source vertex to all other vertices in a weighted graph, even in the presence of negative-weight edges. It's slower than Dijkstra's algorithm but more versatile as it can handle graphs with negative weight edges.

- First, it assigns a distance value to every vertex in the graph. Initialize the distance of the source vertex as 0 and all other vertices as infinity.

- Iterate through all edges |V| - 1 times (where |V| is the number of vertices in the graph), and for each edge (u, v) in the graph, relax the edge if there is a shorter path from the source to v through u.

- However, it's worth noting that while the Bellman-Ford algorithm is suitable for medium to small graphs and can handle negative weight edges, it is not optimized for all-pairs shortest path (APSP) problems. Its implementation may not be as efficient for large graphs or scenarios where APSP is the primary concern.

3) D’Esopo-Pape Algorithm

- Most suitable for this problem if it is a one-vertice shortest path, but for all pair shortest paths, this algorithm is not faster than Dijkstra’s algorithm. Sometimes it is, sometimes it isn’t.

- So what is this algorithm?

+ The D'Esopo-Pape algorithm, also known as the DP algorithm or the "Improved Shortest Path Faster Algorithm," is a variant of Dijkstra's algorithm for finding single-source shortest paths in graphs with non-negative edge weights, including graphs with negative cycles. Michele D'Esopo and Harry F. Pape independently discovered it in the 1960s.

- How it works?

+ Initialization: Initialize a distance array to store the shortest distances from the source vertex to all other vertices. Set the distance to the source vertex as 0 and the distance to all other vertices as infinity.

+ Relaxation Loop: Iterate through all edges in the graph repeatedly until no further improvements can be made to the distances. In each iteration, relax the edges by updating the distance to each adjacent vertex if a shorter path is found.

+ Handling Negative Cycles: Unlike Dijkstra's algorithm, which cannot handle negative cycles, the D'Esopo-Pape algorithm can detect and handle them. After a certain number of iterations, if the distances still decrease, it indicates the presence of a negative cycle reachable from the source vertex.

+ Termination: Once no further improvements can be made to the distances, the algorithm terminates, and the distance array contains the shortest distances from the source vertex to all other vertices.

- Some of the advantages of this algorithm:

+ Ability to Handle Negative Cycles: The algorithm can detect and report negative cycles in the graph, making it useful in scenarios where negative cycles need to be identified.

+ Efficiency: While Dijkstra's algorithm has a time complexity of O(V^2) with a priority queue and O(E log V) with a Fibonacci heap, the D'Esopo-Pape algorithm has a time complexity of O(V * E), which can be more efficient for some graphs.

+ Simplicity: The algorithm is relatively simple to implement and understand, making it accessible for a wide range of applications.

*** Notes about D’Esopo-Pape Algorithm:**

- Not counting the dist and trace we implemented in the previous week, this algorithm needs another sets of vertices:

+ M0: vertices, for which the distance has already been calculated (although it might not be the final distance)

+ M1: vertices, for which the distance currently is calculated

+ M2: vertices, for which the distance has not yet been calculated

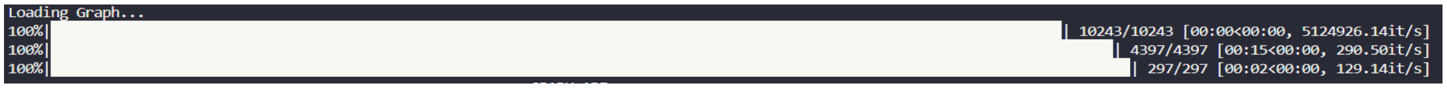
- At each step of the algorithm, we take the vertex from the set M1 (front of the queue) and call it u. Then we put this vertex into the set M0. Then we iterate over all edges coming out of this vertex. Let v be the second end of the current edge, and w its weight.

- Note that in our graph, there is a special thing that there are many paths from stop x to stop y. As we want to take the shortest path, we will delete all of the other paths and take the shortest one. Then we apply this algorithm.

II) Dijkstra’s running time

***Notes:** The time to read data into the graph will be mentioned here, hence, the running time for a, b, and c will not include the time to read data into the graph.

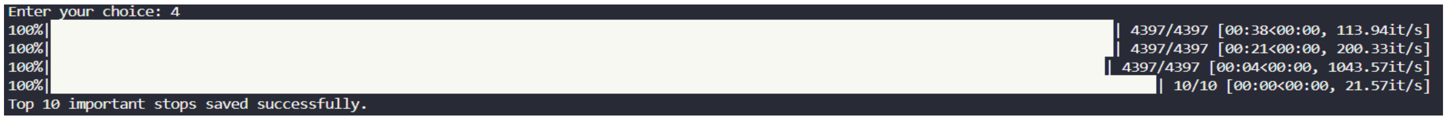
- Total time: 17 seconds



```
Loading Graph...
100%|██████████| 10243/10243 [00:00<00:00, 5124926.14it/s]
100%|██████████| 4397/4397 [00:15<00:00, 290.50it/s]
100%|██████████| 297/297 [00:02<00:00, 129.14it/s]
```

a) The most important stops’ running time

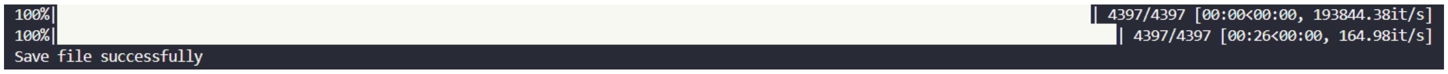
- Total time: 1 minute and 3 seconds (including time to export data into a file)



```
Enter your choice: 4
100%|██████████| 4397/4397 [00:38<00:00, 113.94it/s]
100%|██████████| 4397/4397 [00:21<00:00, 200.33it/s]
100%|██████████| 4397/4397 [00:04<00:00, 1043.57it/s]
100%|██████████| 10/10 [00:00<00:00, 21.57it/s]
Top 10 important stops saved successfully.
```

b) All pairs shortest path running time

- Total time: about 26 seconds



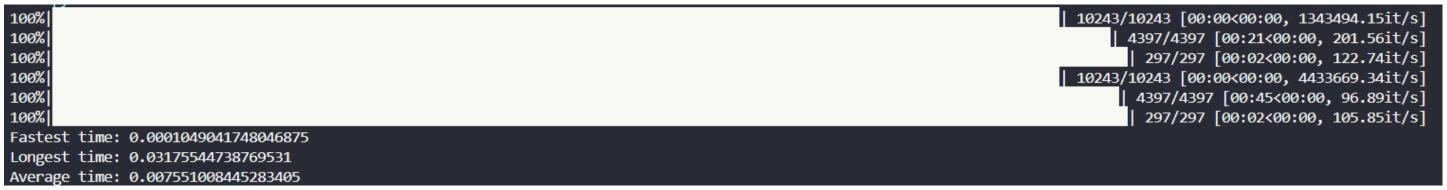
```
100%|██████████| 4397/4397 [00:00<00:00, 193844.38it/s]
100%|██████████| 4397/4397 [00:26<00:00, 164.98it/s]
Save file successfully
```

c) Average time, fastest, longest time running shortest path for one vertice (1..n)

+ **Average time:** 0.007551008445283405

+ **Fastest time:** 0.0001049041748046875

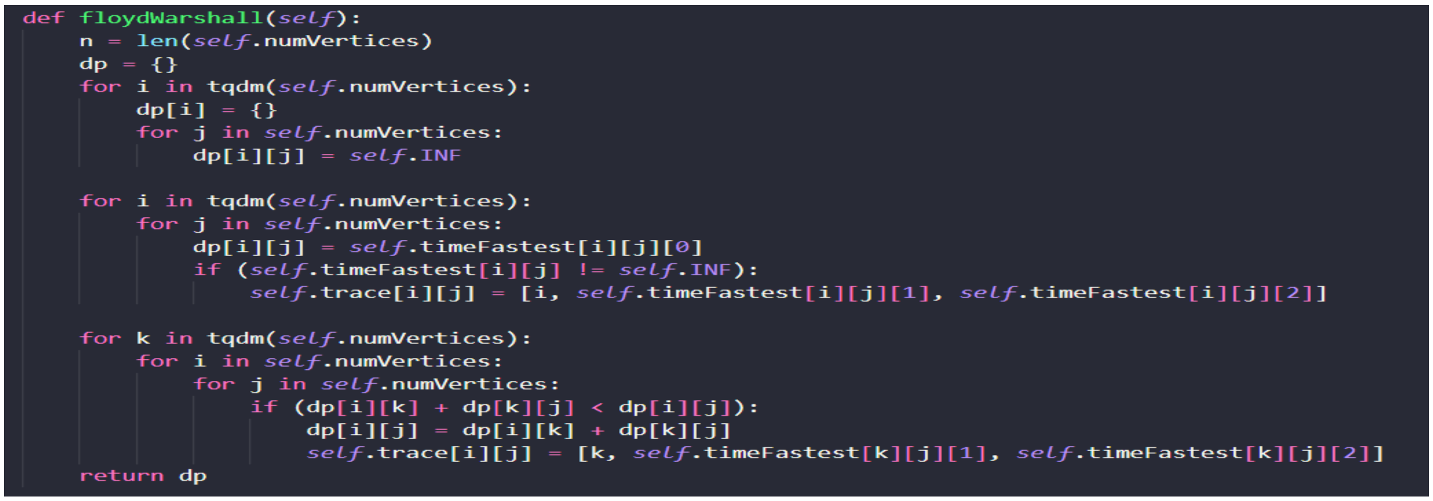
+ **Longest time:** 0.03175544738769531



```
100%|██████████| 10243/10243 [00:00<00:00, 1343494.15it/s]
100%|██████████| 4397/4397 [00:21<00:00, 201.56it/s]
100%|██████████| 297/297 [00:02<00:00, 122.74it/s]
100%|██████████| 10243/10243 [00:00<00:00, 4433669.34it/s]
100%|██████████| 4397/4397 [00:45<00:00, 96.89it/s]
100%|██████████| 297/297 [00:02<00:00, 105.85it/s]
Fastest time: 0.0001049041748046875
Longest time: 0.03175544738769531
Average time: 0.007551008445283405
```

III) Other algorithms for optimization

1) Floyd Warshall Algorithm



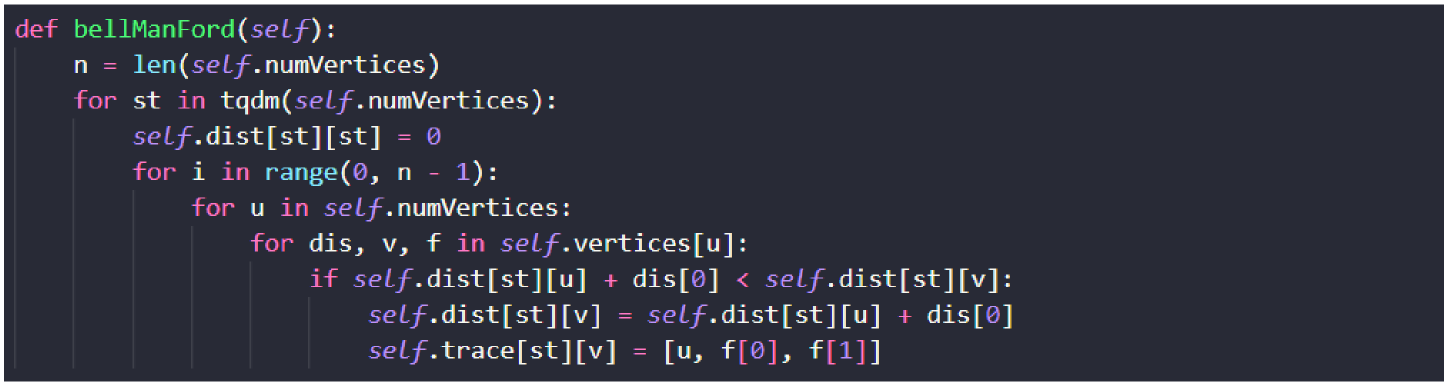
```python
def floydWarshall(self):
    n = len(self.numVertices)
    dp = {}
    for i in tqdm(self.numVertices):
        dp[i] = {}
        for j in self.numVertices:
            dp[i][j] = self.INF

    for i in tqdm(self.numVertices):
        for j in self.numVertices:
            dp[i][j] = self.timeFastest[i][j][0]
            if (self.timeFastest[i][j] != self.INF):
                self.trace[i][j] = [i, self.timeFastest[i][j][1], self.timeFastest[i][j][2]]

    for k in tqdm(self.numVertices):
        for i in self.numVertices:
            for j in self.numVertices:
                if (dp[i][k] + dp[k][j] < dp[i][j]):
                    dp[i][j] = dp[i][k] + dp[k][j]
                    self.trace[i][j] = [k, self.timeFastest[k][j][1], self.timeFastest[k][j][2]]
    return dp
```

- The method of this algorithm I have mentioned above, so I won't discuss here.

- This algorithm can't be used to optimize as its running time is extremely long, can be from one to two hours.

→ I will just give the code but not show the running time.

2) Bellman Ford Algorithm



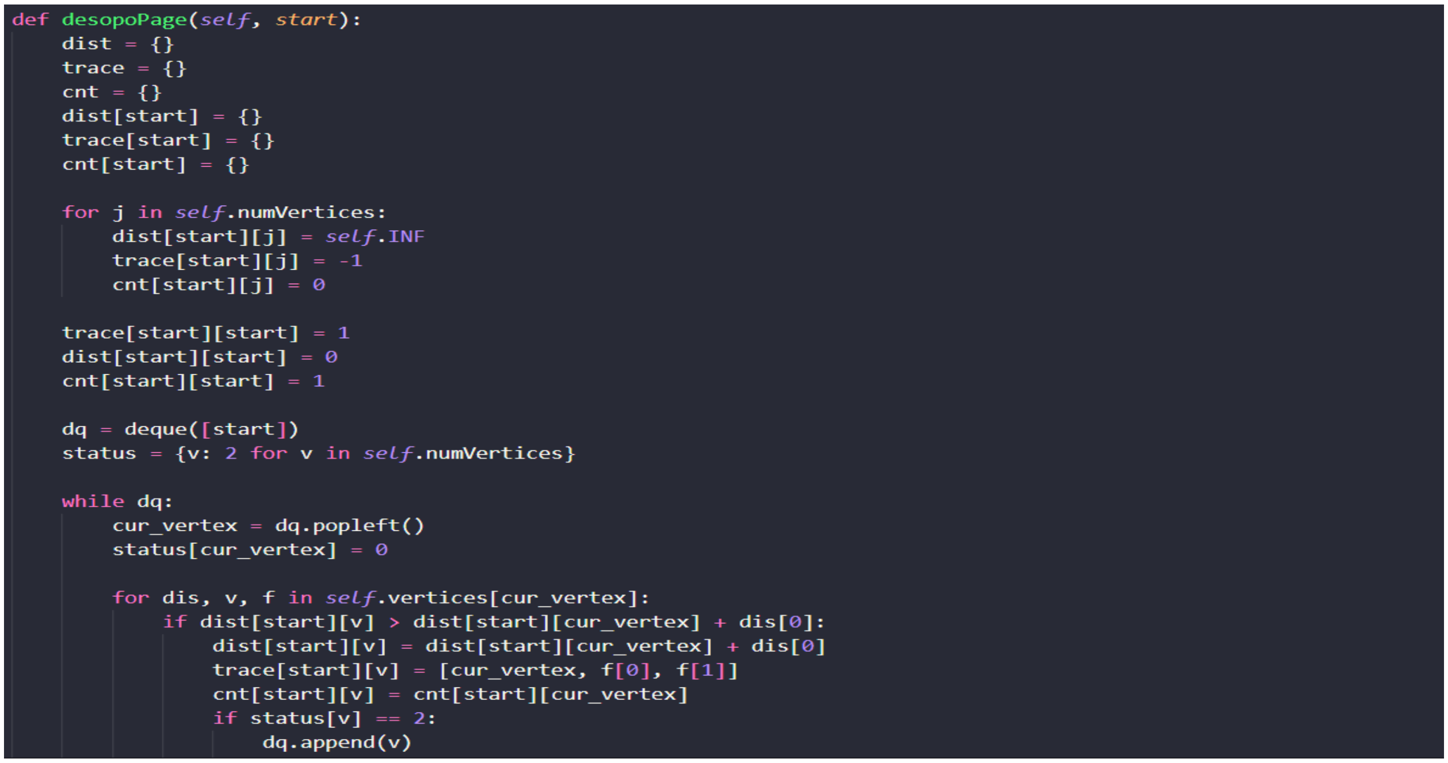
```python
def bellManFord(self):
    n = len(self.numVertices)
    for st in tqdm(self.numVertices):
        self.dist[st][st] = 0
        for i in range(0, n - 1):
            for u in self.numVertices:
                for dis, v, f in self.vertices[u]:
                    if self.dist[st][u] + dis[0] < self.dist[st][v]:
                        self.dist[st][v] = self.dist[st][u] + dis[0]
                        self.trace[st][v] = [u, f[0], f[1]]
```

- The method of this algorithm I have mentioned above, so I won't discuss here.

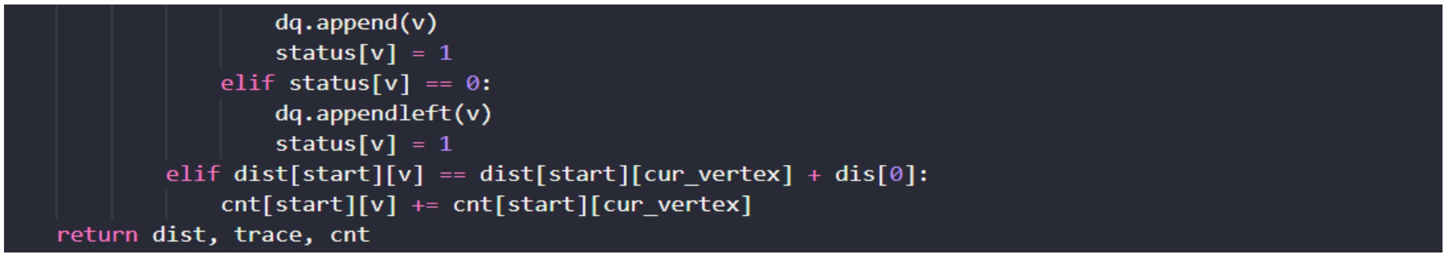
- This algorithm can't be used to optimize as its running time is extremely long, can be from one to two hours.

→ I will just give the code but not show the running time.

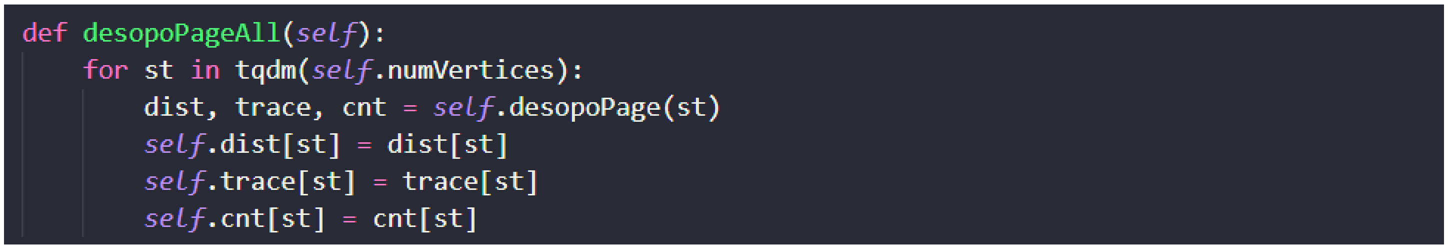
3) D'Esopo-Pape Algorithm



```python
def desopoPage(self, start):
    dist = {}
    trace = {}
    cnt = {}
    dist[start] = {}
    trace[start] = {}
    cnt[start] = {}

    for j in self.numVertices:
        dist[start][j] = self.INF
        trace[start][j] = -1
        cnt[start][j] = 0

    trace[start][start] = 1
    dist[start][start] = 0
    cnt[start][start] = 1

    dq = deque([start])
    status = {v: 2 for v in self.numVertices}

    while dq:
        cur_vertex = dq.popleft()
        status[cur_vertex] = 0

        for dis, v, f in self.vertices[cur_vertex]:
            if dist[start][v] > dist[start][cur_vertex] + dis[0]:
                dist[start][v] = dist[start][cur_vertex] + dis[0]
                trace[start][v] = [cur_vertex, f[0], f[1]]
                cnt[start][v] = cnt[start][cur_vertex]
                if status[v] == 2:
                    dq.append(v)
```



```python
                    dq.append(v)
                    status[v] = 1
                elif status[v] == 0:
                    dq.appendleft(v)
                    status[v] = 1
            elif dist[start][v] == dist[start][cur_vertex] + dis[0]:
                cnt[start][v] += cnt[start][cur_vertex]
    return dist, trace, cnt
```



```python
def desopoPageAll(self):
    for st in tqdm(self.numVertices):
        dist, trace, cnt = self.desopoPage(st)
        self.dist[st] = dist[st]
        self.trace[st] = trace[st]
        self.cnt[st] = cnt[st]
```

- This algorithm can be used to optimize the program for some cases, however, it does not guarantee that the whole program will work faster for all cases.

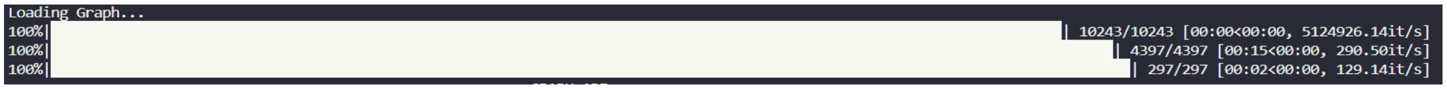
- As the method to implement this algorithm I have mentioned above, I won't disucss here any more.

- For the desopoPageAll, we will just find shortest path for all pairs, and then set the corresponding dist, trace, cnt to that of the start stop.

IV) D'Esopo-Pape running time

Notes: The time to read data into the graph will be mentioned here, hence, the running time for a, b, and c will not include the time to read data into the graph.

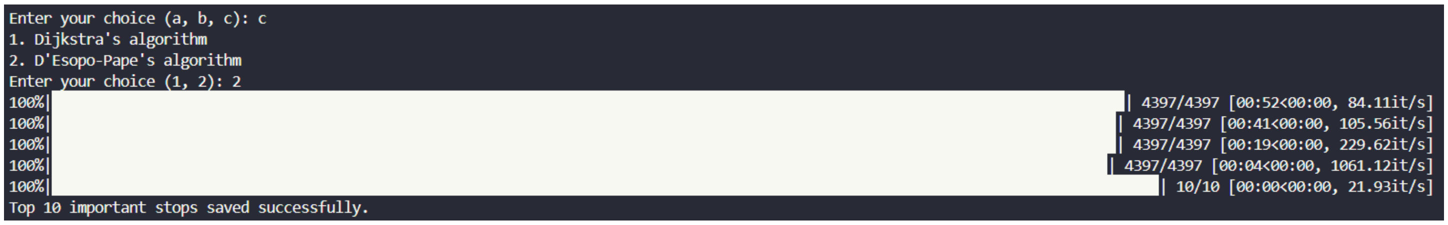
- Total time: 17 seconds



```
Loading Graph...
100%|██████████████████████████████████████████| 10243/10243 [00:00<00:00, 5124926.14it/s]
100%|██████████████████████████████████████████| 4397/4397 [00:15<00:00, 290.50it/s]
100%|██████████████████████████████████████████| 297/297 [00:02<00:00, 129.14it/s]
```

a) The most important stops' running time

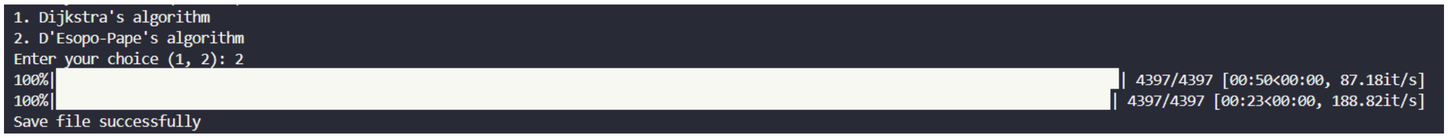
- Total time: 1 minute and 56 seconds



```
Enter your choice (a, b, c): c
1. Dijkstra's algorithm
2. D'Esopo-Pape's algorithm
Enter your choice (1, 2): 2
100%|██████████████████████████████████████████| 4397/4397 [00:52<00:00, 84.11it/s]
100%|██████████████████████████████████████████| 4397/4397 [00:41<00:00, 105.56it/s]
100%|██████████████████████████████████████████| 4397/4397 [00:19<00:00, 229.62it/s]
100%|██████████████████████████████████████████| 4397/4397 [00:04<00:00, 1061.12it/s]
100%|██████████████████████████████████████████| 10/10 [00:00<00:00, 21.93it/s]
Top 10 important stops saved successfully.
```

b) All pairs shortest path running time

- Total time: 1 minute and 13 seconds



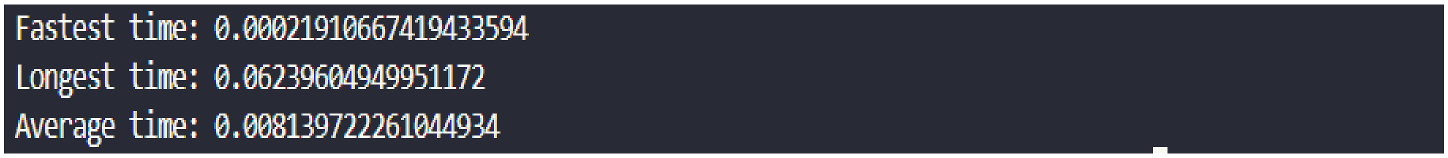
```
1. Dijkstra's algorithm
2. D'Esopo-Pape's algorithm
Enter your choice (1, 2): 2
100%|██████████████████████████████████████████| 4397/4397 [00:50<00:00, 87.18it/s]
100%|██████████████████████████████████████████| 4397/4397 [00:23<00:00, 188.82it/s]
Save file successfully
```

c) Average time, fastest, longest time running shortest path for one vertice (1..n)

+ **Average time:** 0.008139722261044934

+ **Fastest time:** 0.00021910667419433594

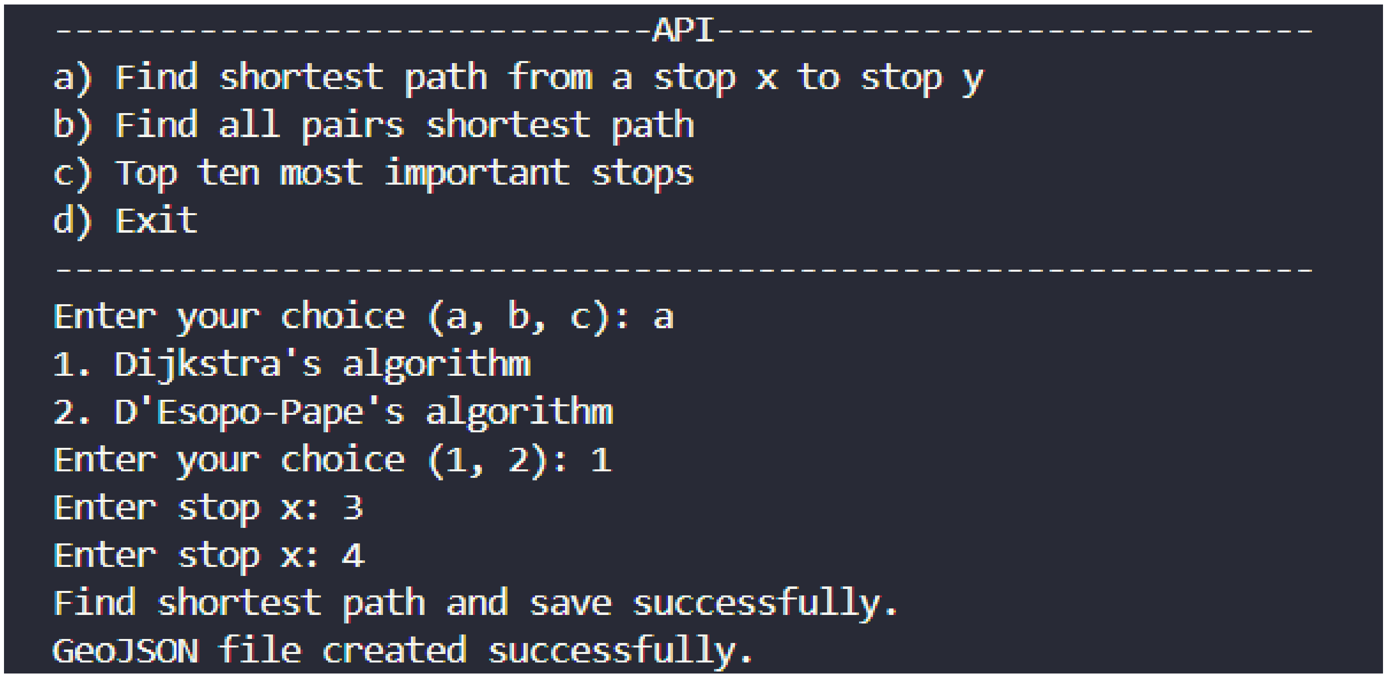
+ **Longest time:** 0.06239604949951172



```
Fastest time: 0.00021910667419433594
Longest time: 0.06239604949951172
Average time: 0.008139722261044934
```

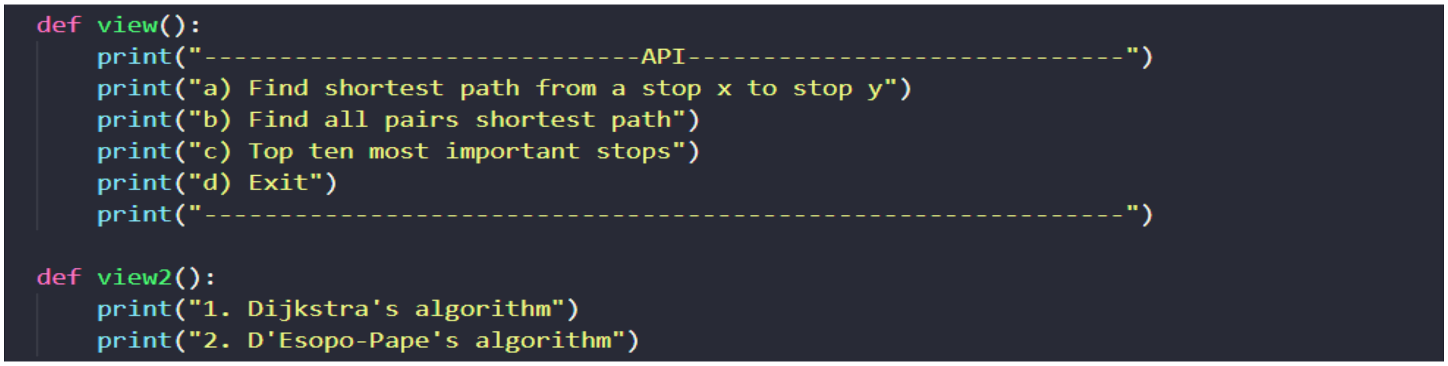
V) User interface and Benchmark Code

1) User Interface

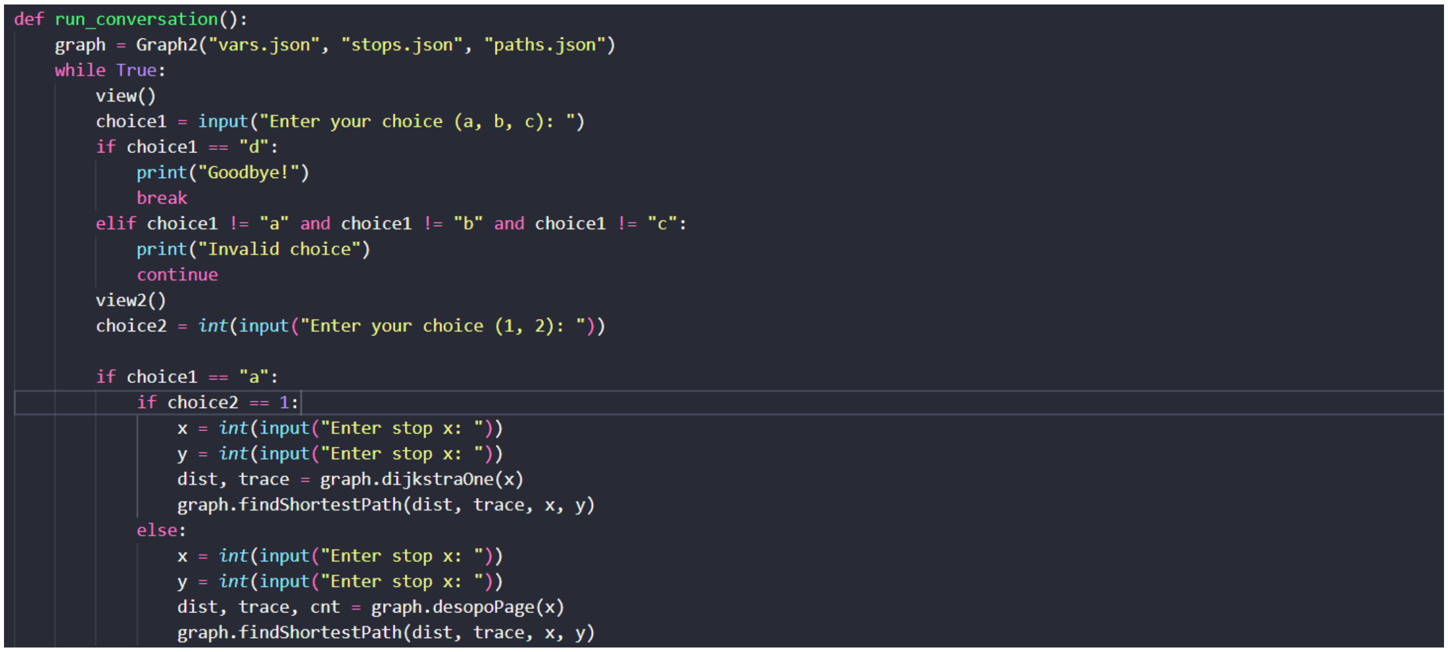


```
-----------------------------API-----------------------------
a) Find shortest path from a stop x to stop y
b) Find all pairs shortest path
c) Top ten most important stops
d) Exit
-------------------------------------------------------------
Enter your choice (a, b, c): a
1. Dijkstra's algorithm
2. D'Esopo-Pape's algorithm
Enter your choice (1, 2): 1
Enter stop x: 3
Enter stop x: 4
Find shortest path and save successfully.
GeoJSON file created successfully.
```

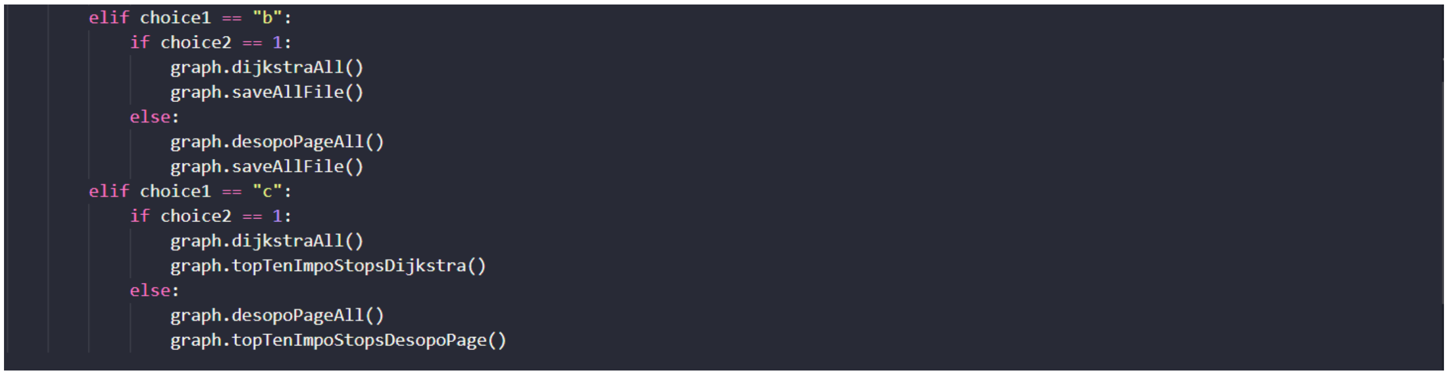
2) User interface code



```python
def view():
    print("-----------------------------API-----------------------------")
    print("a) Find shortest path from a stop x to stop y")
    print("b) Find all pairs shortest path")
    print("c) Top ten most important stops")
    print("d) Exit")
    print("-------------------------------------------------------------")

def view2():
    print("1. Dijkstra's algorithm")
    print("2. D'Esopo-Pape's algorithm")
```



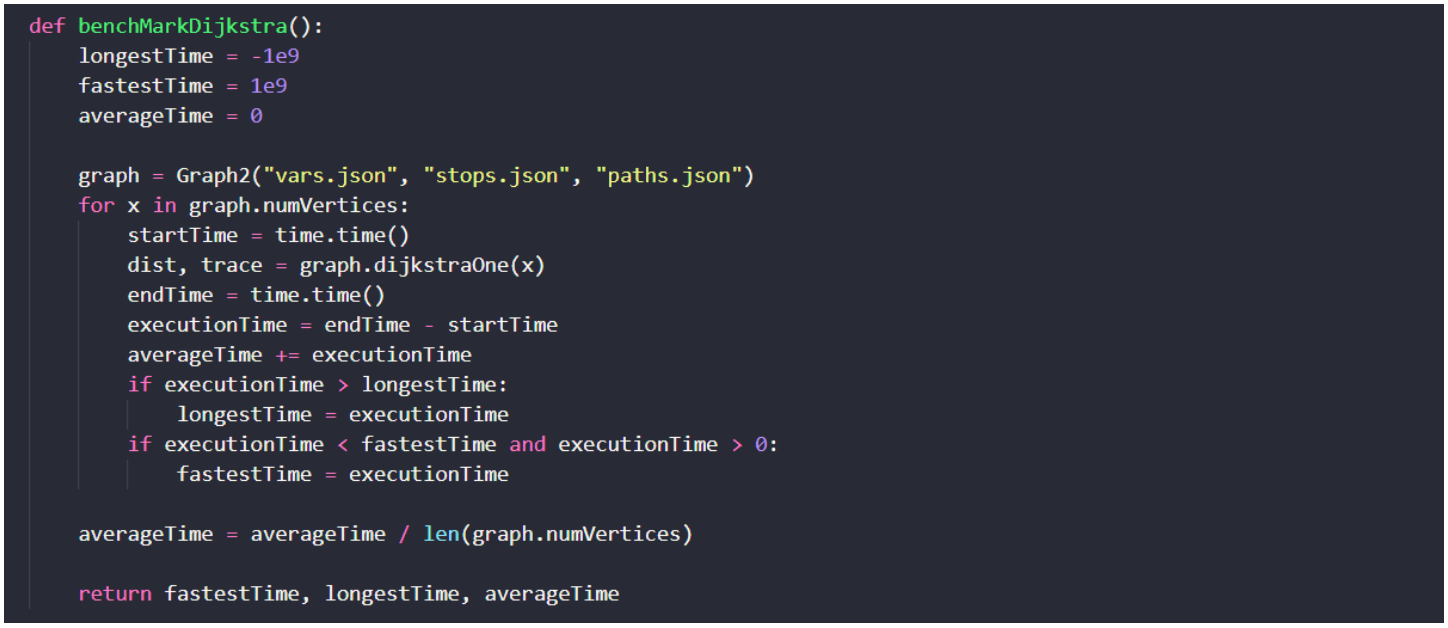
```python
def run_conversation():
    graph = Graph2("vars.json", "stops.json", "paths.json")
    while True:
        view()
        choice1 = input("Enter your choice (a, b, c): ")
        if choice1 == "d":
            print("Goodbye!")
            break
        elif choice1 != "a" and choice1 != "b" and choice1 != "c":
            print("Invalid choice")
            continue
        view2()
        choice2 = int(input("Enter your choice (1, 2): "))

        if choice1 == "a":
            if choice2 == 1:
                x = int(input("Enter stop x: "))
                y = int(input("Enter stop x: "))
                dist, trace = graph.dijkstraOne(x)
                graph.findShortestPath(dist, trace, x, y)
            else:
                x = int(input("Enter stop x: "))
                y = int(input("Enter stop x: "))
                dist, trace, cnt = graph.desopoPage(x)
                graph.findShortestPath(dist, trace, x, y)
```



```python
        elif choice1 == "b":
            if choice2 == 1:
                graph.dijkstraAll()
                graph.saveAllFile()
            else:
                graph.desopoPageAll()
                graph.saveAllFile()
        elif choice1 == "c":
            if choice2 == 1:
                graph.dijkstraAll()
                graph.topTenImpoStopsDijkstra()
            else:
                graph.desopoPageAll()
                graph.topTenImpoStopsDesopoPage()
```

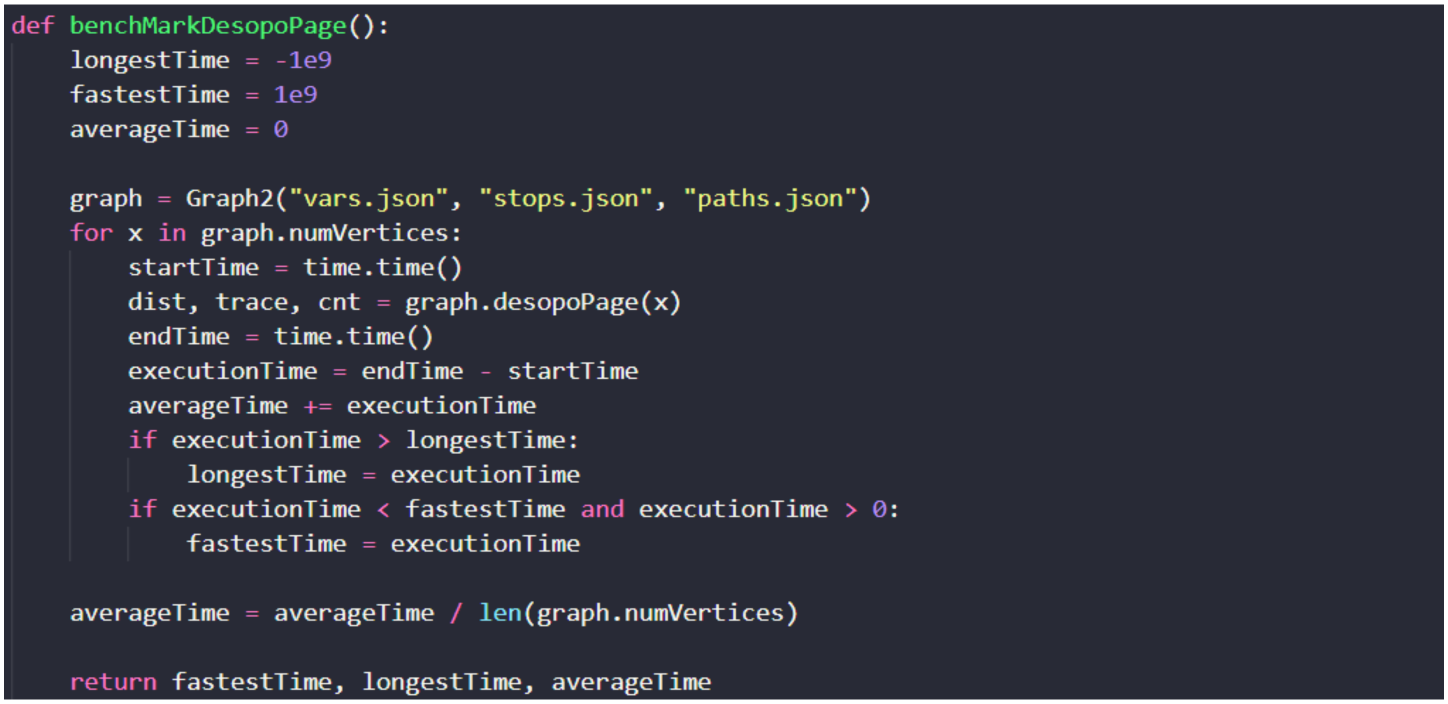
3) Benchmark Code

- Dijkstra:



```python
def benchMarkDijkstra():
    longestTime = -1e9
    fastestTime = 1e9
    averageTime = 0

    graph = Graph2("vars.json", "stops.json", "paths.json")
    for x in graph.numVertices:
        startTime = time.time()
        dist, trace = graph.dijkstraOne(x)
        endTime = time.time()
        executionTime = endTime - startTime
        averageTime += executionTime
        if executionTime > longestTime:
            longestTime = executionTime
        if executionTime < fastestTime and executionTime > 0:
            fastestTime = executionTime

    averageTime = averageTime / len(graph.numVertices)

    return fastestTime, longestTime, averageTime
```

- D’Esopo-Pape:



```python
def benchMarkDesopoPage():
    longestTime = -1e9
    fastestTime = 1e9
    averageTime = 0

    graph = Graph2("vars.json", "stops.json", "paths.json")
    for x in graph.numVertices:
        startTime = time.time()
        dist, trace, cnt = graph.desopoPage(x)
        endTime = time.time()
        executionTime = endTime - startTime
        averageTime += executionTime
        if executionTime > longestTime:
            longestTime = executionTime
        if executionTime < fastestTime and executionTime > 0:
            fastestTime = executionTime

    averageTime = averageTime / len(graph.numVertices)

    return fastestTime, longestTime, averageTime
```

V) Conclusion

- In conclusion, the exploration and implementation of Dijkstra's algorithm and its optimization using various other algorithms have been meticulously detailed throughout this project. I primarily focused on enhancing the efficiency and performance of the BusMap application at the University of Science, Ho Chi Minh.

- I initially provided insights into the essential aspects of Dijkstra's algorithm and its application in finding the shortest paths between vertices in a weighted graph. I delineated the methodology of Dijkstra's algorithm alongside other algorithms like Floyd Warshall and Bellman Ford, elucidating their suitability and limitations in the context of the BusMap project.

- Subsequently, I delved into the D'Esopo-Pape algorithm, showcasing its potential as an optimization technique for the project. While highlighting its advantages in handling negative cycles and its efficiency in certain scenarios, I also acknowledged that its applicability may vary across different cases within the project.

- Furthermore, I meticulously analyzed the running times of both Dijkstra's algorithm and the optimized D'Esopo-Pape algorithm across various scenarios, including the computation of the most important stops, all pairs shortest path, and the shortest path for a single vertex. These analyses provided crucial insights into the performance metrics of the algorithms, enabling a comprehensive understanding of their efficacy within the BusMap application.

- Moreover, I touched upon the user interface design and provided benchmark code snippets for both Dijkstra's and D'Esopo-Pape algorithms, offering practical insights into their implementation.

- In essence, this project has not only elucidated the intricacies of graph algorithms and their optimization but also provided valuable insights into their practical application within a real-world scenario. By meticulously analyzing the performance metrics and intricacies of each algorithm, I lay the foundation for further enhancements and optimizations within the BusMap application, thereby contributing to the ongoing advancements in transportation and navigation systems.