COMP 1828 - Designing, developing and testing solutions for the London Underground system

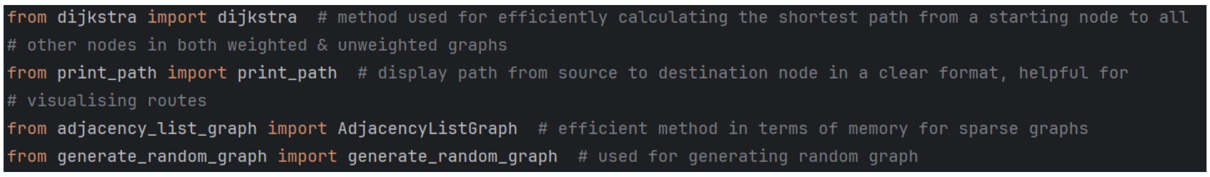
TASK 1

Manual versus Code-Based Execution of the Algorithm

Adjacency List: This data structure is efficient for representing sparse graphs as it requires less memory by only storing existing edges rather than all possible connections. This fits well with the London Underground model, where not all stations are directly connected keeping the graph memory efficient.

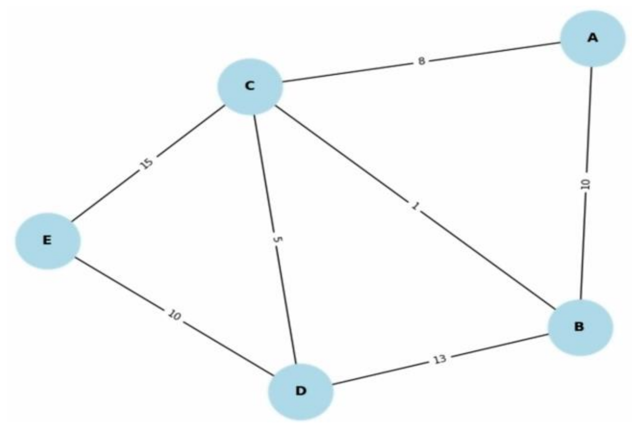
Dijkstra's Algorithm: This algorithm efficiently finds the shortest path from a starting node to all other nodes in both weighted and unweighted graphs, making it ideal for calculating the minimum journey times in a transport network.

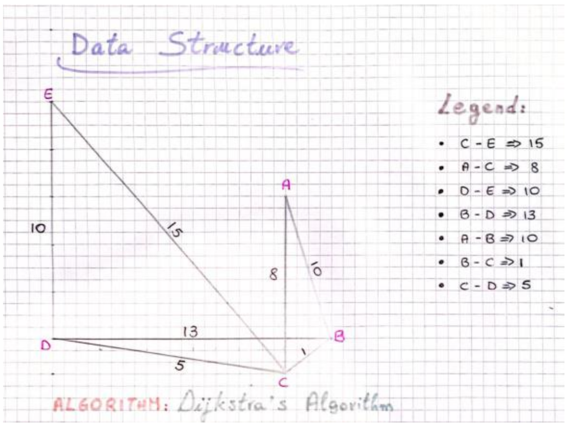
Together, the adjacency list and Dijkstra's algorithm, provide an efficient solution for calculating shortest paths and analysing performance based on different network sizes.



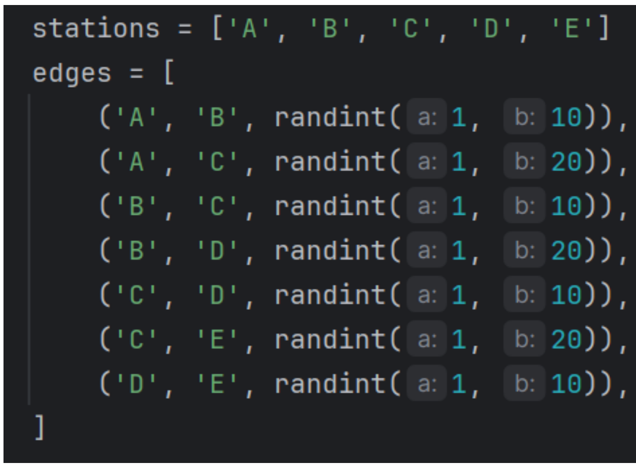
```python
from dijkstra import dijkstra  # method used for efficiently calculating the shortest path from a starting node to all
# other nodes in both weighted & unweighted graphs
from print_path import print_path  # display path from source to destination node in a clear format, helpful for
# visualising routes
from adjacency_list_graph import AdjacencyListGraph  # efficient method in terms of memory for sparse graphs
from generate_random_graph import generate_random_graph  # used for generating random graph
```

Simple Dataset & Data Structure

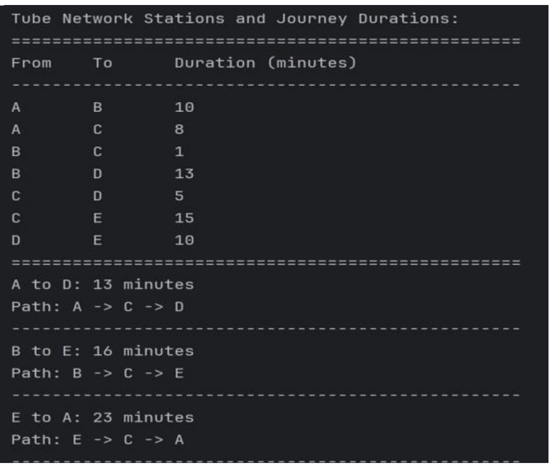




Manual Algorithm Execution, Tube Network Stations & Journey Durations [mins]



```python
stations = ['A', 'B', 'C', 'D', 'E']
edges = [
    ('A', 'B', randint( a: 1,  b: 10)),
    ('A', 'C', randint( a: 1,  b: 20)),
    ('B', 'C', randint( a: 1,  b: 10)),
    ('B', 'D', randint( a: 1,  b: 20)),
    ('C', 'D', randint( a: 1,  b: 10)),
    ('C', 'E', randint( a: 1,  b: 20)),
    ('D', 'E', randint( a: 1,  b: 10)),
]
```



```
Tube Network Stations and Journey Durations:
==================================================
From    To      Duration (minutes)
--------------------------------------------------
A       B       10
A       C       8
B       C       1
B       D       13
C       D       5
C       E       15
D       E       10
==================================================
A to D: 13 minutes
Path: A -> C -> D
--------------------------------------------------
B to E: 16 minutes
Path: B -> C -> E
--------------------------------------------------
E to A: 23 minutes
Path: E -> C -> A
--------------------------------------------------
```

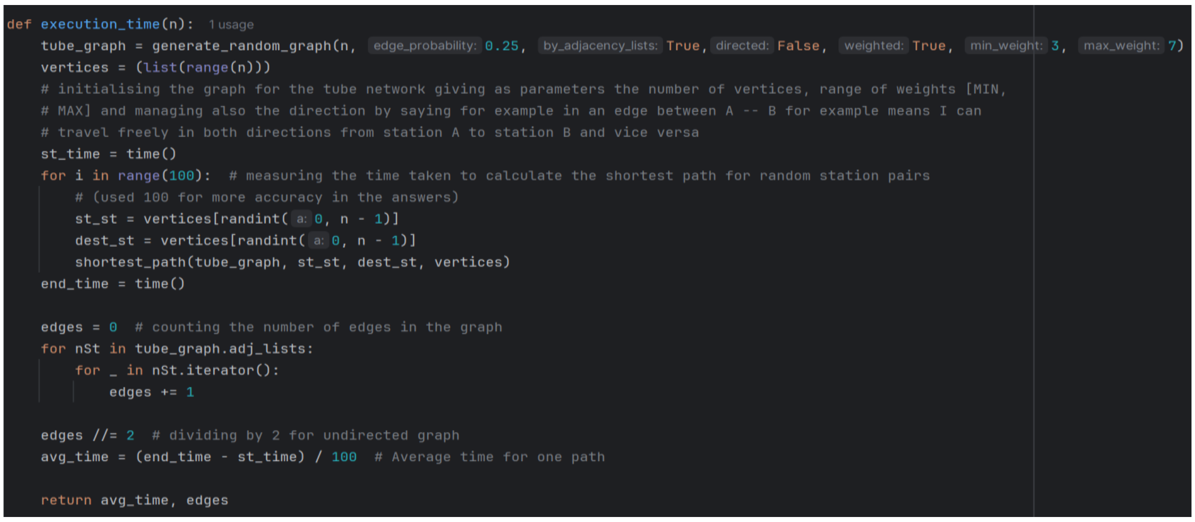
A - D Possible Paths				
A -> C -> D	A -> C: 8 min	C -> D: 5 min	-	Total: 13 min
A -> B -> D	A -> B: 10 min	B -> D: 13 min	-	Total: 23 min
A -> B -> C -> D	A -> B: 10 min	B -> C: 1 min	C -> D: 5 min	Total: 16 min
The shortest path is A -> C -> D with 13 minutes (Same result in Python)				

B - E Possible Paths				
B -> C -> E	B -> C: 1 min	C -> E: 15 min	-	Total: 16 min
B -> D -> E	B -> D: 13 min	D -> E: 10 min	-	Total: 23 min
B -> C -> D -> E	B -> C: 1 min	C -> D: 5 min	D -> E: 10 min	Total: 16 min
In this case we have two paths with the same journey time duration but in this case the algorithm makes a deterministic choice and choose B -> C -> E with 16 minutes as the shortest path as it is the first one it encounters (Same result in Python)				

E - A Possible Paths					
E -> C -> A	E -> C: 15 min	C -> A: 8 min	-	-	Total: 23 min
E -> D -> B -> A	E -> D: 10 min	D -> B: 13 min	B -> A: 10 min	-	Total: 33 min
E -> C -> B -> A	E -> C: 15 min	C -> B: 1 min	B -> A: 10 min	-	Total: 26 min
E -> D -> C -> B -> A	E -> D: 10 min	D -> C: 5 min	C -> B: 1 min	B -> A: 10 min	Total: 26 min
The shortest path is E -> C -> A with 23 minutes (Same result in Python)					

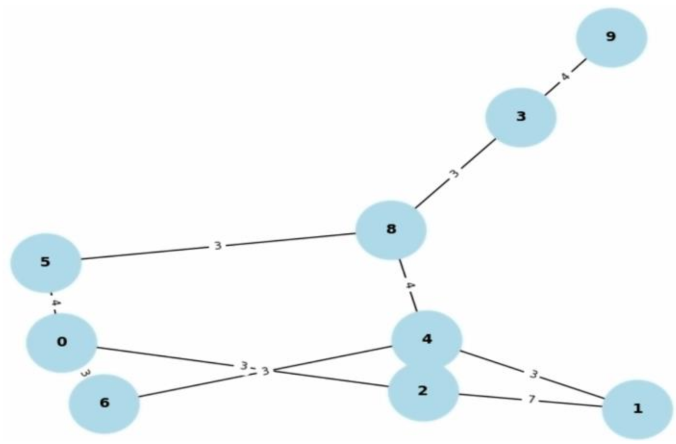
Empirical Measurement of Time Complexity

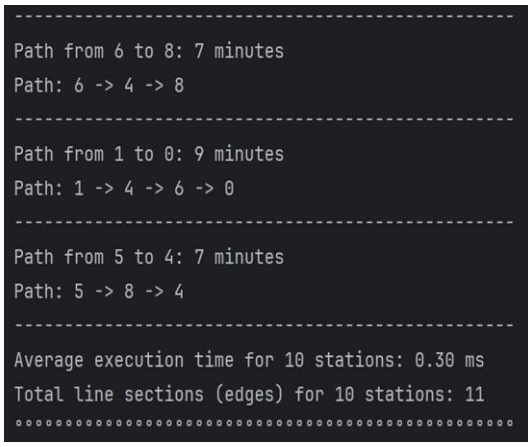
Execution Time Measurement



```
def execution_time(n):  1 usage
    tube_graph = generate_random_graph(n, edge_probability: 0.25, by_adjacency_lists: True, directed: False, weighted: True, min_weight: 3, max_weight: 7)
    vertices = (list(range(n)))
    # initialising the graph for the tube network giving as parameters the number of vertices, range of weights [MIN,
    # MAX] and managing also the direction by saying for example in an edge between A -- B for example means I can
    # travel freely in both directions from station A to station B and vice versa
    st_time = time()
    for i in range(100):  # measuring the time taken to calculate the shortest path for random station pairs
        # (used 100 for more accuracy in the answers)
        st_st = vertices[randint( a: 0, n - 1)]
        dest_st = vertices[randint( a: 0, n - 1)]
        shortest_path(tube_graph, st_st, dest_st, vertices)
    end_time = time()

    edges = 0  # counting the number of edges in the graph
    for nSt in tube_graph.adj_lists:
        for _ in nSt.iterator():
            edges += 1

    edges //= 2  # dividing by 2 for undirected graph
    avg_time = (end_time - st_time) / 100  # Average time for one path

    return avg_time, edges
```

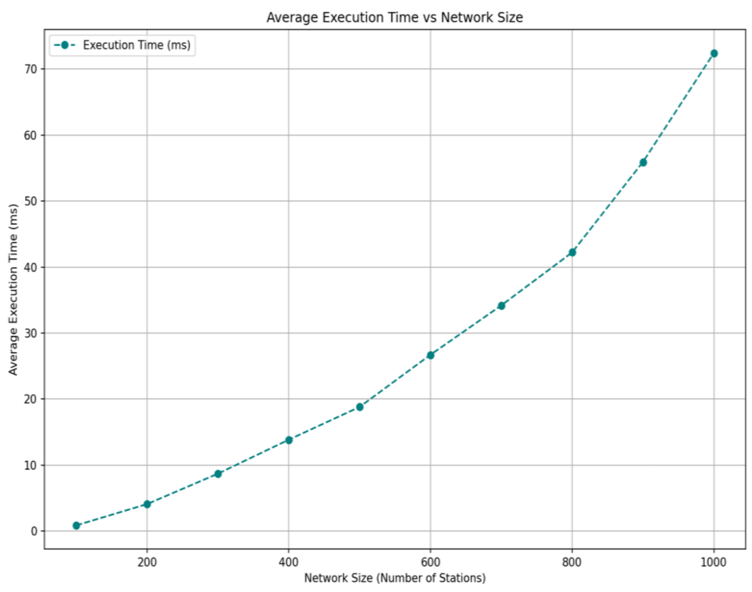
Example of Generated Graph & Artificial Network Generation

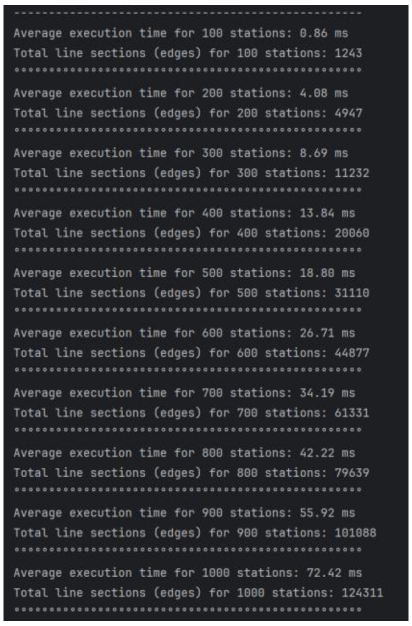




```
------------------------------------------------
Path from 6 to 8: 7 minutes
Path: 6 -> 4 -> 8
------------------------------------------------
Path from 1 to 0: 9 minutes
Path: 1 -> 4 -> 6 -> 0
------------------------------------------------
Path from 5 to 4: 7 minutes
Path: 5 -> 8 -> 4
------------------------------------------------
Average execution time for 10 stations: 0.30 ms
Total line sections (edges) for 10 stations: 11
ooooooooooooooooooooooooooooooooooooooooooooooo
```

Time Complexity Graph & Total Line Sections





```
------------------------------------------------------
Average execution time for 100 stations: 0.86 ms
Total line sections (edges) for 100 stations: 1243
oooooooooooooooooooooooooooooooooooooooooooooooooooooo
Average execution time for 200 stations: 4.08 ms
Total line sections (edges) for 200 stations: 4947
oooooooooooooooooooooooooooooooooooooooooooooooooooooo
Average execution time for 300 stations: 8.69 ms
Total line sections (edges) for 300 stations: 11232
oooooooooooooooooooooooooooooooooooooooooooooooooooooo
Average execution time for 400 stations: 13.84 ms
Total line sections (edges) for 400 stations: 20060
oooooooooooooooooooooooooooooooooooooooooooooooooooooo
Average execution time for 500 stations: 18.80 ms
Total line sections (edges) for 500 stations: 31110
oooooooooooooooooooooooooooooooooooooooooooooooooooooo
Average execution time for 600 stations: 26.71 ms
Total line sections (edges) for 600 stations: 44877
oooooooooooooooooooooooooooooooooooooooooooooooooooooo
Average execution time for 700 stations: 34.19 ms
Total line sections (edges) for 700 stations: 61331
oooooooooooooooooooooooooooooooooooooooooooooooooooooo
Average execution time for 800 stations: 42.22 ms
Total line sections (edges) for 800 stations: 79639
oooooooooooooooooooooooooooooooooooooooooooooooooooooo
Average execution time for 900 stations: 55.92 ms
Total line sections (edges) for 900 stations: 101088
oooooooooooooooooooooooooooooooooooooooooooooooooooooo
Average execution time for 1000 stations: 72.42 ms
Total line sections (edges) for 1000 stations: 124311
oooooooooooooooooooooooooooooooooooooooooooooooooooooo
```

The tool used for graph generation is **Matplotlib.pyplot.**

Theoretical Time Complexity

Dijkstra's algorithm using a binary min-heap and an adjacency list has a time complexity of O((V + E) log V), where:

- V is the number of vertices (stations).
- E is the number of edges (connections between stations).

For densely connected graphs the number of edges scales quadratically with the number of vertices (E - V^2) but for more sparsely connected ones, E grows more linearly with V. In the plot we can see how with the increase of the numbers of stations, there's a gradual, non-linear increase in the execution time because V(log V) dominates the complexity when E is small.

Key Observations

1. The observed growth in the execution time is consistent with the theoretical time complexity of O(V log V) for sparse graphs;

2. There are no major discrepancies as there aren't any drastic deviations in the plotted graph and the gradual increase in execution time means how efficiently the algorithm is handling larger graphs;
3. There could be a possible discrepancy for larger graphs, but we are talking about thousands and thousands of stations, and we can notice this as a deviation in the plotted graph.

TASK 2

Manual versus Code-Based Execution of the Algorithm

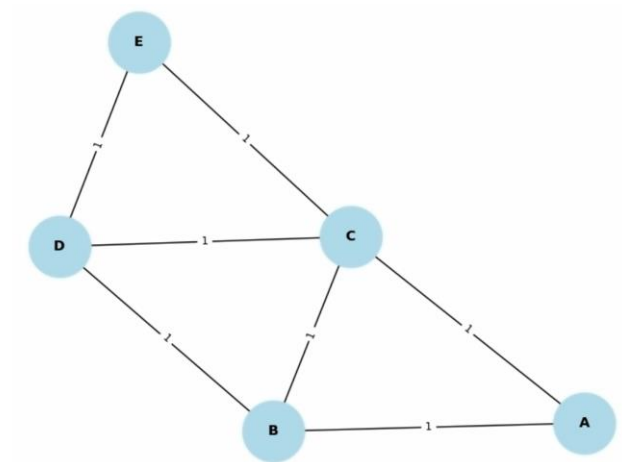
Adjacency List: This data structure is efficient for representing sparse graphs as it requires less memory by only storing existing edges rather than all possible connections, fitting well with the London Underground model, where not all stations are directly connected keeping the graph memory efficient.

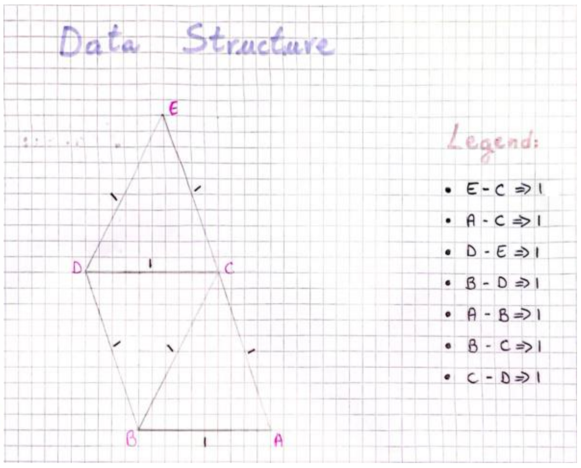
Dijkstra's Algorithm: This algorithm efficiently finds the shortest path from a starting node to all other nodes in both weighted and unweighted graphs, making it a versatile choice and ideal for calculating the shortest path based on the number of stops.

Together, the adjacency list and Dijkstra, provide an efficient solution for calculating shortest paths and analysing performance based on different network sizes.

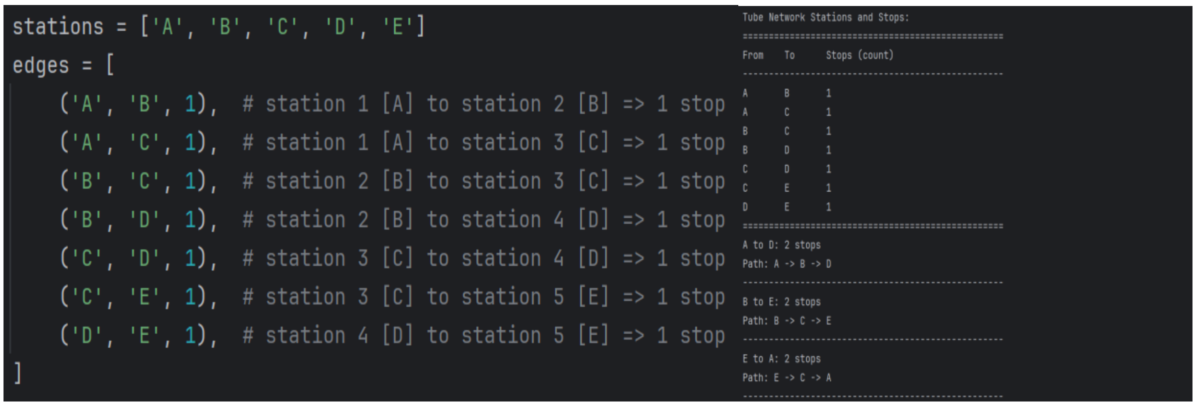
```python
# libraries and methods used in task1
from task1 import AdjacencyListGraph, dijkstra, print_path, plot_execution_times, generate_random_graph, time, randint
```

Simple Dataset & Data Structure





Manual Algorithm Execution, Tube Network Stations and Journey Durations [Stops]



```python
stations = ['A', 'B', 'C', 'D', 'E']
edges = [
    ('A', 'B', 1),  # station 1 [A] to station 2 [B] => 1 stop
    ('A', 'C', 1),  # station 1 [A] to station 3 [C] => 1 stop
    ('B', 'C', 1),  # station 2 [B] to station 3 [C] => 1 stop
    ('B', 'D', 1),  # station 2 [B] to station 4 [D] => 1 stop
    ('C', 'D', 1),  # station 3 [C] to station 4 [D] => 1 stop
    ('C', 'E', 1),  # station 3 [C] to station 5 [E] => 1 stop
    ('D', 'E', 1),  # station 4 [D] to station 5 [E] => 1 stop
]
```

```
Tube Network Stations and Stops:
==================================================
From    To      Stops (count)
--------------------------------------------------
A       B       1
A       C       1
B       C       1
B       D       1
C       D       1
C       E       1
D       E       1
==================================================
A to D: 2 stops
Path: A -> B -> D
--------------------------------------------------
B to E: 2 stops
Path: B -> C -> E
--------------------------------------------------
E to A: 2 stops
Path: E -> C -> A
--------------------------------------------------
```

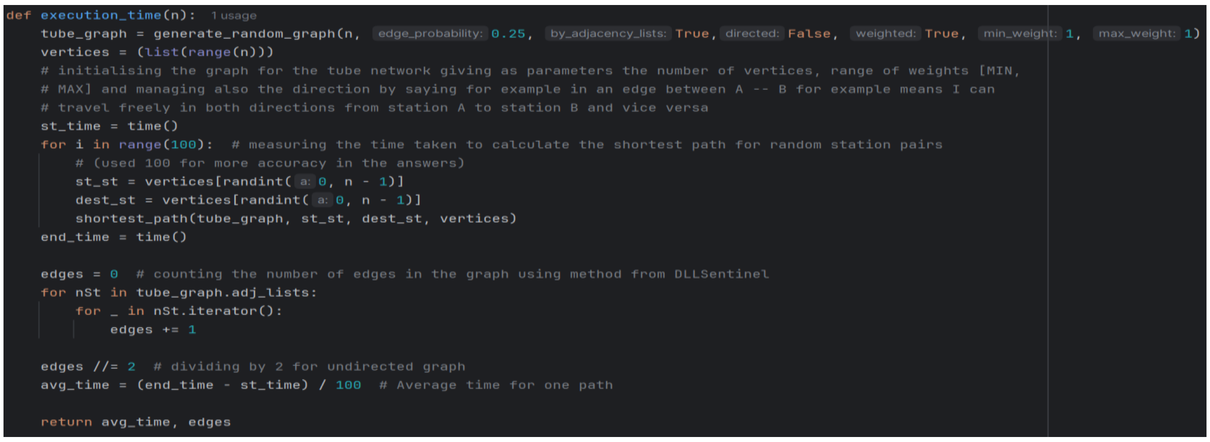
A - D Possible Paths				
A -> C -> D	**A -> C: 1 stop**	**C -> D: 1 stop**	**-**	**Total: 2 stops**
A -> B -> D	**A -> B: 1 stop**	**B -> D: 1 stop**	**-**	**Total: 2 stops**
A -> B -> C -> D	**A -> B: 1 stop**	**B -> C: 1 stop**	**C -> D: 1 stop**	**Total: 3 stops**
In this case we have two paths with the same number of stops but in this case the algorithm makes a deterministic choice and choose A -> C -> D with 2 stops as the shortest path as it is the first one it encounters (Same result in Python)				

B - E Possible Paths				
B -> C -> E	**B -> C: 1 stop**	**C -> E: 1 stop**	**-**	**Total: 2 stops**
B -> D -> E	**B -> D: 1 stop**	**D -> E: 1 stop**	**-**	**Total: 2 stops**
B -> C -> D -> E	**B -> C: 1 stop**	**C -> D: 1 stop**	**D -> E: 1 stop**	**Total: 3 stops**
In this case we have two paths with the same number of stops but in this case the algorithm makes a deterministic choice and choose B -> C -> E with 2 stops as the shortest path as it is the first one it encounters (Same result in Python)				

E - A Possible Paths					
E -> C -> A	**E -> C: 1 stop**	**C -> A: 1 stop**	**-**	**-**	**Total: 2 stops**
E -> D -> B -> A	**E -> D: 1 stop**	**D -> B: 1 stop**	**B -> A: 1 stop**	**-**	**Total: 3 stops**
E -> C -> B -> A	**E -> C: 1 stop**	**C -> B: 1 stop**	**B -> A: 1 stop**	**-**	**Total: 3 stops**
E -> D -> C -> B -> A	**E -> D: 1 stop**	**D -> C: 1 stop**	**C -> B: 1 stop**	**B -> A: 1 stop**	**Total: 4 stops**
The shortest path is E -> C -> A with 2 stops (Same result in Python)					

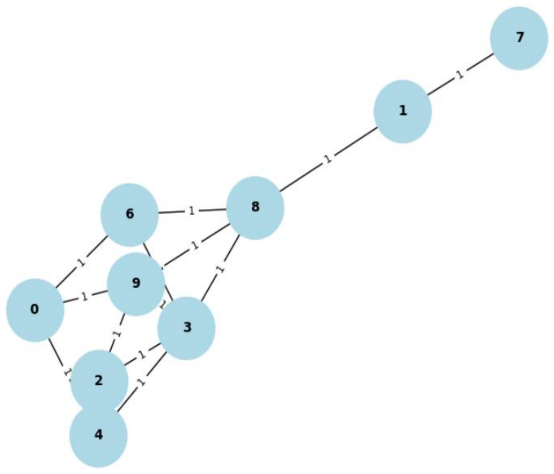
Empirical Measurement of Time Complexity

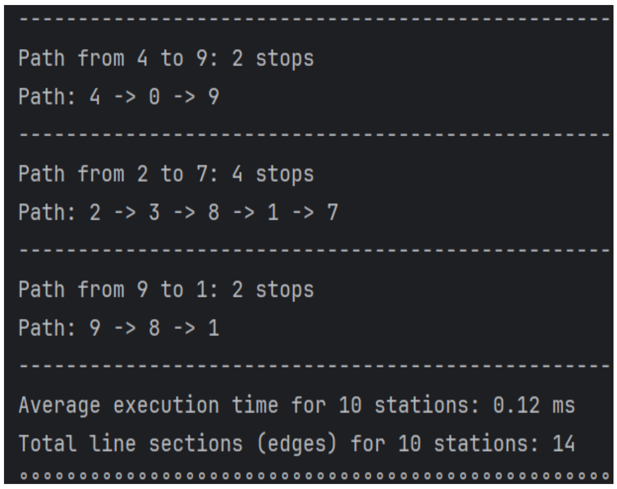
Execution Time Measurement



```python
def execution_time(n):  1 usage
    tube_graph = generate_random_graph(n,  edge_probability: 0.25,  by_adjacency_lists: True,  directed: False,  weighted: True,  min_weight: 1,  max_weight: 1)
    vertices = (list(range(n)))
    # initialising the graph for the tube network giving as parameters the number of vertices, range of weights [MIN,
    # MAX] and managing also the direction by saying for example in an edge between A -- B for example means I can
    # travel freely in both directions from station A to station B and vice versa
    st_time = time()
    for i in range(100):  # measuring the time taken to calculate the shortest path for random station pairs
        # (used 100 for more accuracy in the answers)
        st_st = vertices[randint( a: 0, n - 1)]
        dest_st = vertices[randint( a: 0, n - 1)]
        shortest_path(tube_graph, st_st, dest_st, vertices)
    end_time = time()

    edges = 0  # counting the number of edges in the graph using method from DLLSentinel
    for nSt in tube_graph.adj_lists:
        for _ in nSt.iterator():
            edges += 1

    edges //= 2  # dividing by 2 for undirected graph
    avg_time = (end_time - st_time) / 100  # Average time for one path

    return avg_time, edges
```

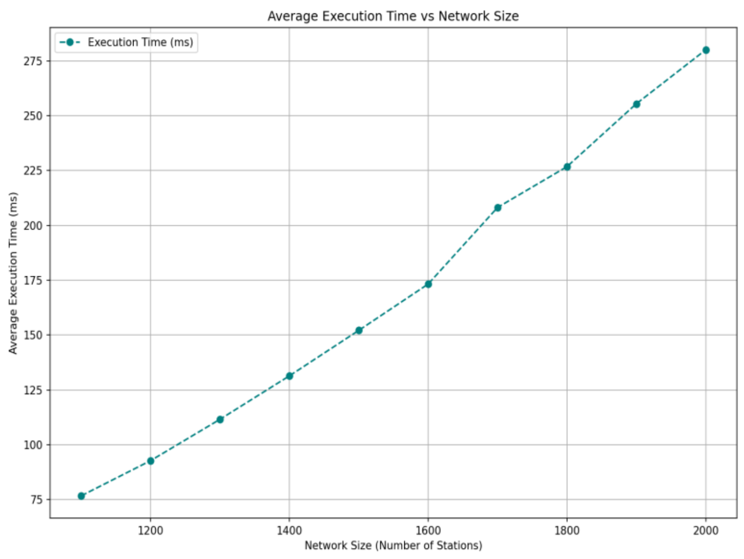
Example of Generated Graph & Artificial Network Generation

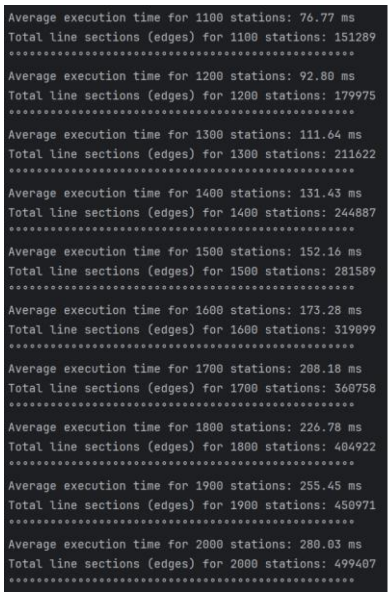




```
---------------------------------------------
Path from 4 to 9: 2 stops
Path: 4 -> 0 -> 9
---------------------------------------------
Path from 2 to 7: 4 stops
Path: 2 -> 3 -> 8 -> 1 -> 7
---------------------------------------------
Path from 9 to 1: 2 stops
Path: 9 -> 8 -> 1
---------------------------------------------
Average execution time for 10 stations: 0.12 ms
Total line sections (edges) for 10 stations: 14
ooooooooooooooooooooooooooooooooooooooooooooooo
```

Time Complexity Graph & Total Line Sections





```
Average execution time for 1100 stations: 76.77 ms
Total line sections (edges) for 1100 stations: 151289
oooooooooooooooooooooooooooooooooooooooooooooooooo
Average execution time for 1200 stations: 92.80 ms
Total line sections (edges) for 1200 stations: 179975
oooooooooooooooooooooooooooooooooooooooooooooooooo
Average execution time for 1300 stations: 111.64 ms
Total line sections (edges) for 1300 stations: 211622
oooooooooooooooooooooooooooooooooooooooooooooooooo
Average execution time for 1400 stations: 131.43 ms
Total line sections (edges) for 1400 stations: 244887
oooooooooooooooooooooooooooooooooooooooooooooooooo
Average execution time for 1500 stations: 152.16 ms
Total line sections (edges) for 1500 stations: 281589
oooooooooooooooooooooooooooooooooooooooooooooooooo
Average execution time for 1600 stations: 173.28 ms
Total line sections (edges) for 1600 stations: 319099
oooooooooooooooooooooooooooooooooooooooooooooooooo
Average execution time for 1700 stations: 208.18 ms
Total line sections (edges) for 1700 stations: 360758
oooooooooooooooooooooooooooooooooooooooooooooooooo
Average execution time for 1800 stations: 226.78 ms
Total line sections (edges) for 1800 stations: 404922
oooooooooooooooooooooooooooooooooooooooooooooooooo
Average execution time for 1900 stations: 255.45 ms
Total line sections (edges) for 1900 stations: 450971
oooooooooooooooooooooooooooooooooooooooooooooooooo
Average execution time for 2000 stations: 280.03 ms
Total line sections (edges) for 2000 stations: 499407
oooooooooooooooooooooooooooooooooooooooooooooooooo
```

The tool used for graph generation is **Matplotlib.pyplot.**

Theoretical Time Complexity

Dijkstra's algorithm has a time complexity of O((V + E) log V), where:

- V is the number of vertices (stations).
- E is the number of edges (connections between stations).

Dijkstra’s algorithm is suitable here as it calculates the shortest path based on the number of stops, even if distances in minutes are not used.
In the plot we can see how with the increase of the numbers of stations, there's a clear near-linear increase in the execution time, which aligns well with the theoretical complexity of O((V + E)log V), highlighting the fact that even with the additional overhead from priority queue operations, the algorithm scales efficiently within a network like the London Underground’s one.

Key Observations

1. The observed growth in the execution time is consistent with the theoretical time complexity of O((V + E) log V);
2. There are no major discrepancies as there aren't any drastic deviations in the plotted graph and the clear near-linear increase in execution time means how efficiently the algorithm is handling larger graphs, even with the priority queue overhead;
3. There could be a possible discrepancy for larger graphs, but we are talking about thousands and thousands of stations, and we can notice this as a deviation in the plotted graph.
4. While Breadth-First Search (BFS) algorithm might be a faster choice for focusing purely on unweighted pathfinding, Dijkstra's algorithm it's more a versatile choice for both weighted and unweighted pathfinding.

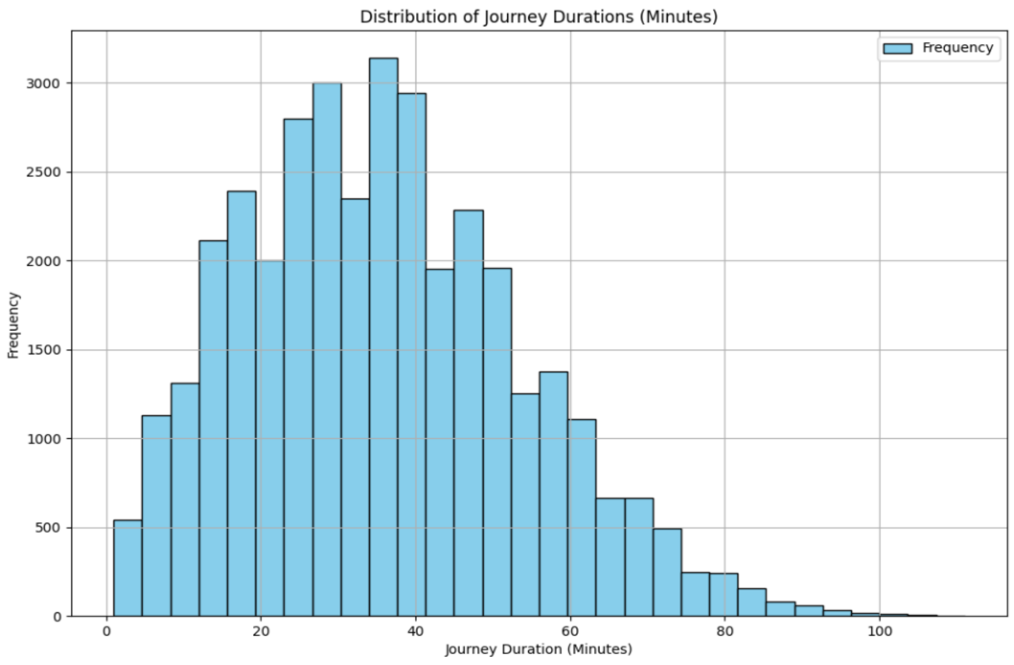
TASK 3

Journey Duration Histograms and Longest Path [mins]

The data from 'London Underground data.xlsx' has been manually extracted and translated into Python language using a dictionary named 'lines' to store all the information. The dictionary it's organised into several line names and each line has an ordered list of stations and a tuple with the format ("Station A", "Station B", distance [min]), which describes direct connections between stations (ex. A -- B) on that line. 'Lines' is then imported from 'underground_lines.py', which allows the data to be accessed and used in the main file.

```
Total number of journey durations calculated: 36315
Duplicate journeys included/excluded: Excluded
```

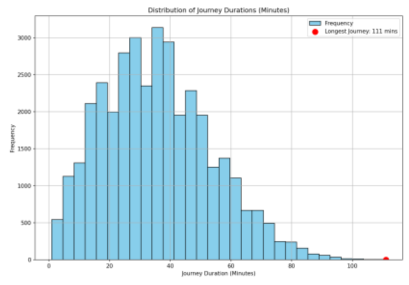
Histogram

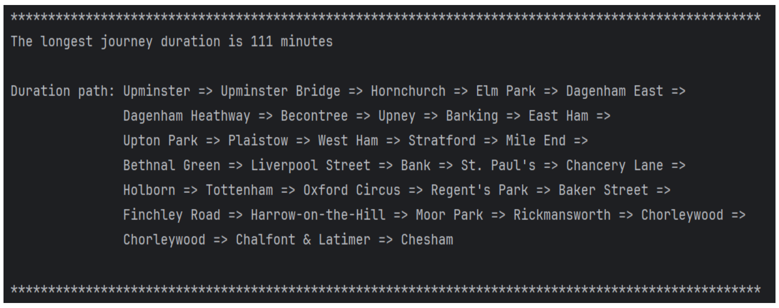


The tool used for Histogram generation is **Matplotlib.pyplot.**

The histogram displays the distribution of journey durations, with most journeys grouped between 10 and 50 minutes and it’s notable a gradual decrease in frequency for journeys longer than 56/57 minutes.

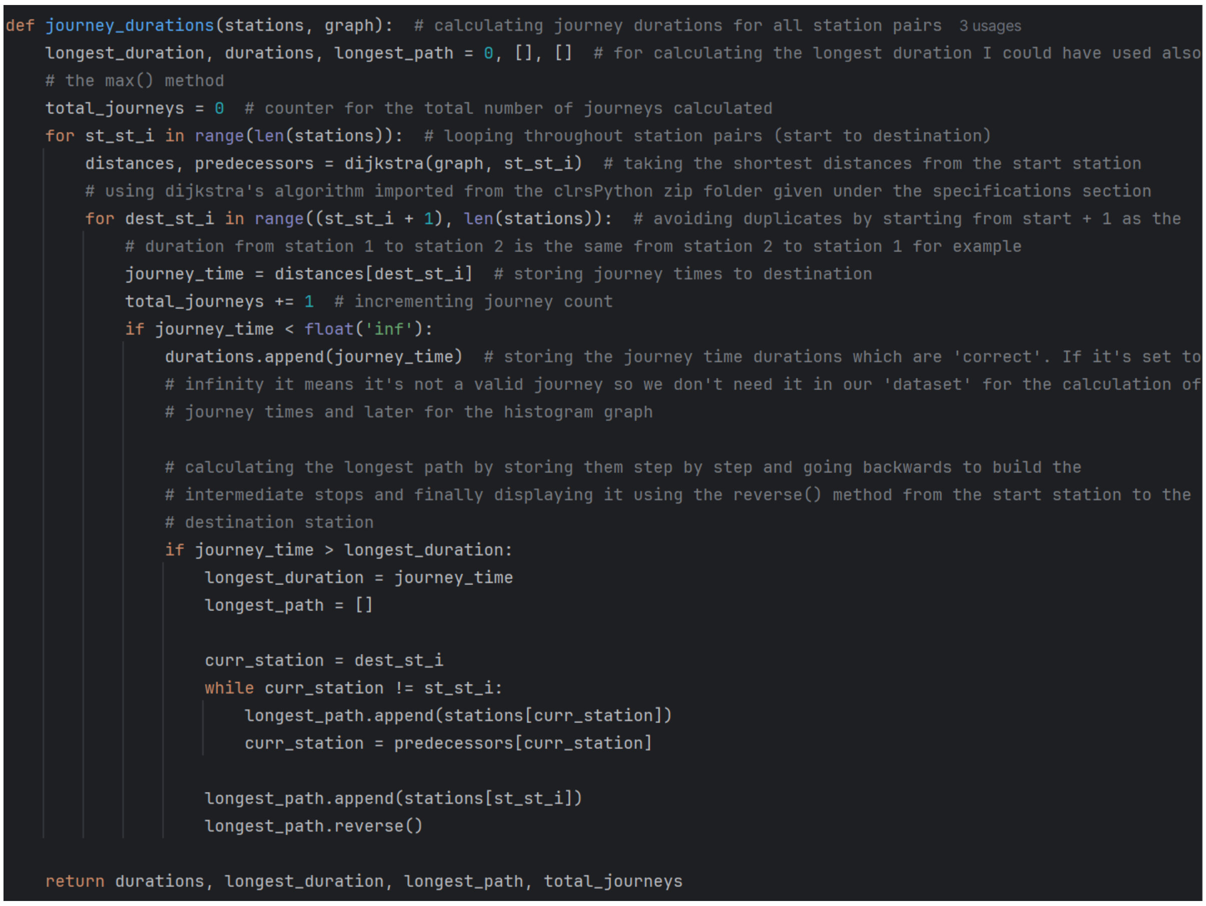
Longest Journey Duration [mins]





```
****************************************************************************************************
The longest journey duration is 111 minutes

Duration path: Upminster => Upminster Bridge => Hornchurch => Elm Park => Dagenham East =>
               Dagenham Heathway => Becontree => Upney => Barking => East Ham =>
               Upton Park => Plaistow => West Ham => Stratford => Mile End =>
               Bethnal Green => Liverpool Street => Bank => St. Paul's => Chancery Lane =>
               Holborn => Tottenham => Oxford Circus => Regent's Park => Baker Street =>
               Finchley Road => Harrow-on-the-Hill => Moor Park => Rickmansworth => Chorleywood =>
               Chorleywood => Chalfont & Latimer => Chesham

****************************************************************************************************
```

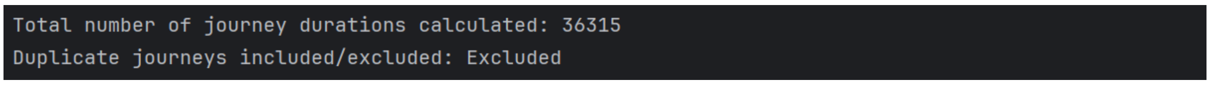
Code Implementation



```
def journey_durations(stations, graph):  # calculating journey durations for all station pairs  3 usages
    longest_duration, durations, longest_path = 0, [], []  # for calculating the longest duration I could have used also
    # the max() method
    total_journeys = 0  # counter for the total number of journeys calculated
    for st_st_i in range(len(stations)):  # looping throughout station pairs (start to destination)
        distances, predecessors = dijkstra(graph, st_st_i)  # taking the shortest distances from the start station
        # using dijkstra's algorithm imported from the clrsPython zip folder given under the specifications section
        for dest_st_i in range((st_st_i + 1), len(stations)):  # avoiding duplicates by starting from start + 1 as the
            # duration from station 1 to station 2 is the same from station 2 to station 1 for example
            journey_time = distances[dest_st_i]  # storing journey times to destination
            total_journeys += 1  # incrementing journey count
            if journey_time < float('inf'):
                durations.append(journey_time)  # storing the journey time durations which are 'correct'. If it's set to
                # infinity it means it's not a valid journey so we don't need it in our 'dataset' for the calculation of
                # journey times and later for the histogram graph

                # calculating the longest path by storing them step by step and going backwards to build the
                # intermediate stops and finally displaying it using the reverse() method from the start station to the
                # destination station
                if journey_time > longest_duration:
                    longest_duration = journey_time
                    longest_path = []

                    curr_station = dest_st_i
                    while curr_station != st_st_i:
                        longest_path.append(stations[curr_station])
                        curr_station = predecessors[curr_station]

                    longest_path.append(stations[st_st_i])
                    longest_path.reverse()

    return durations, longest_duration, longest_path, total_journeys
```

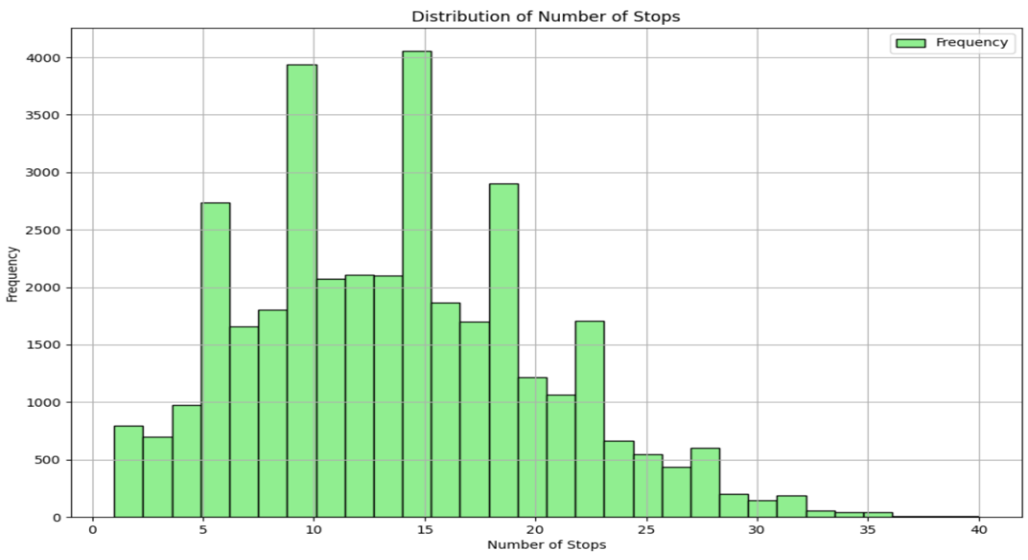
The histogram highlights that most trips are under 50 minutes of duration, with shorter journeys being the most common ones. However, the longest journey goes from Upminster to Chesham for a total of 111 minutes, involving several transfers across the network and highlighting the different travel demands within the London Underground system.

Journey Duration Histograms and Longest Path [Stops]



```
Total number of journey durations calculated: 36315
Duplicate journeys included/excluded: Excluded
```

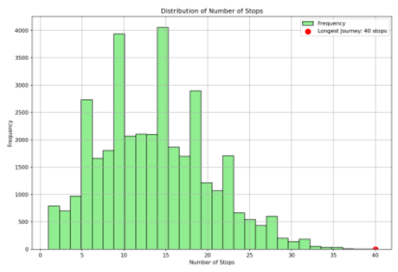
Histogram

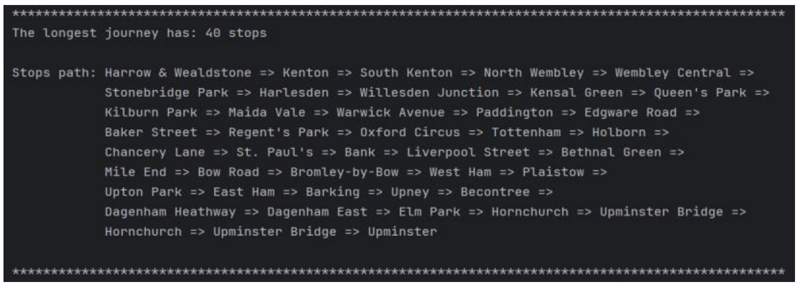


The tool used for Histogram generation is **Matplotlib.pyplot.**

The histogram of journey durations based on number of stops highlights the fact of most trips being quite short (under 20 stops). However, there are longer trips, with the longest being 40 stops.

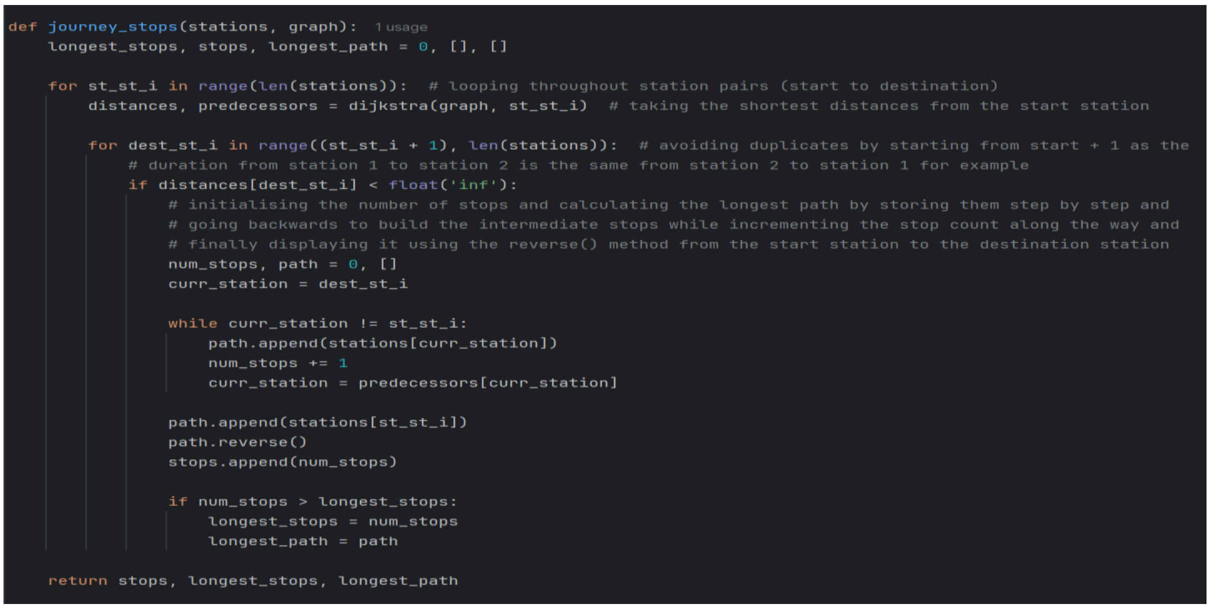
Longest Journey Duration [stops]





```
***************************************************************************************************
The longest journey has: 40 stops

Stops path: Harrow & Wealdstone => Kenton => South Kenton => North Wembley => Wembley Central =>
            Stonebridge Park => Harlesden => Willesden Junction => Kensal Green => Queen's Park =>
            Kilburn Park => Maida Vale => Warwick Avenue => Paddington => Edgware Road =>
            Baker Street => Regent's Park => Oxford Circus => Tottenham => Holborn =>
            Chancery Lane => St. Paul's => Bank => Liverpool Street => Bethnal Green =>
            Mile End => Bow Road => Bromley-by-Bow => West Ham => Plaistow =>
            Upton Park => East Ham => Barking => Upney => Becontree =>
            Dagenham Heathway => Dagenham East => Elm Park => Hornchurch => Upminster Bridge =>
            Hornchurch => Upminster Bridge => Upminster

***************************************************************************************************
```

Code Implementation



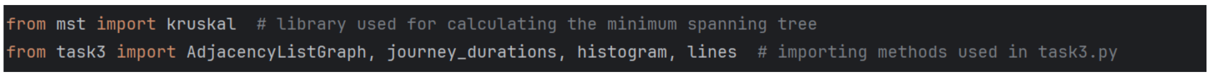
```python
def journey_stops(stations, graph):  1 usage
    longest_stops, stops, longest_path = 0, [], []

    for st_st_i in range(len(stations)):  # looping throughout station pairs (start to destination)
        distances, predecessors = dijkstra(graph, st_st_i)  # taking the shortest distances from the start station

        for dest_st_i in range((st_st_i + 1), len(stations)):  # avoiding duplicates by starting from start + 1 as the
            # duration from station 1 to station 2 is the same from station 2 to station 1 for example
            if distances[dest_st_i] < float('inf'):
                # initialising the number of stops and calculating the longest path by storing them step by step and
                # going backwards to build the intermediate stops while incrementing the stop count along the way and
                # finally displaying it using the reverse() method from the start station to the destination station
                num_stops, path = 0, []
                curr_station = dest_st_i

                while curr_station != st_st_i:
                    path.append(stations[curr_station])
                    num_stops += 1
                    curr_station = predecessors[curr_station]

                path.append(stations[st_st_i])
                path.reverse()
                stops.append(num_stops)

                if num_stops > longest_stops:
                    longest_stops = num_stops
                    longest_path = path

    return stops, longest_stops, longest_path
```

Comparing the two histograms (Subtask 3a & 3b), the distributions are similar towards shorter trips, with most journeys under 50 minutes or 20 stops. However, the longest journeys do not fully align with each other, as longer travel times can result from factors other than just the number of stops, such as for example waiting times.

TASK 4

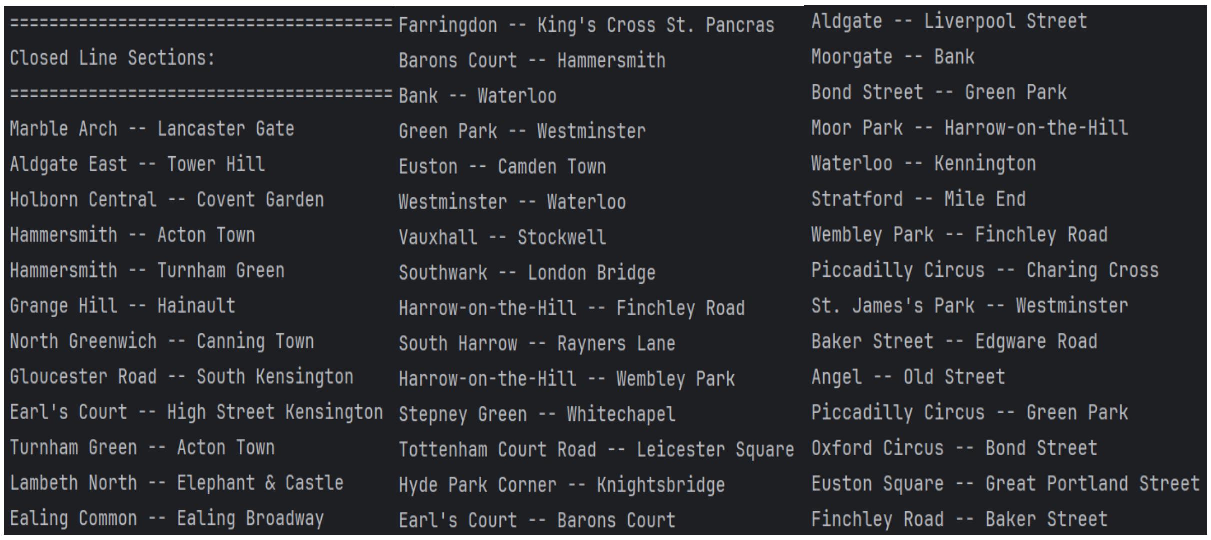
Line Section Closure Analysis

Kruskal’s algorithm is an optimal choice for this task as it maintains network connectivity by constructing a Minimum Spanning Tree and the edges not in the MST are flagged as ‘closed’, achieving maximum closures without disconnecting any stations.



```
from mst import kruskal  # library used for calculating the minimum spanning tree
from task3 import AdjacencyListGraph, journey_durations, histogram, lines  # importing methods used in task3.py
```

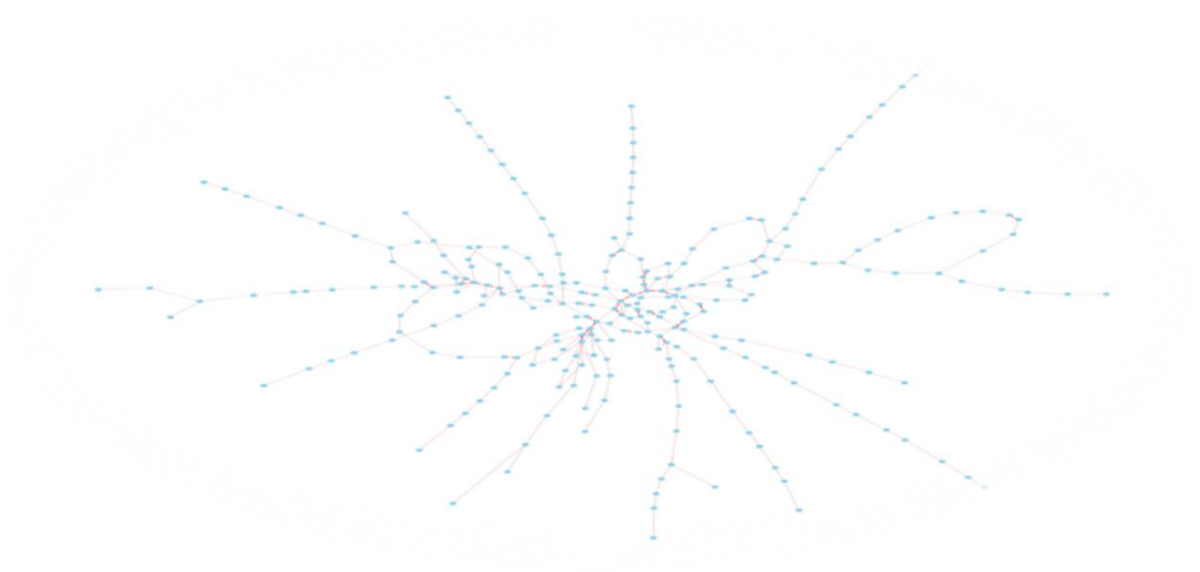
Closed Line Sections



```
=======================================
Closed Line Sections:
=======================================
Marble Arch -- Lancaster Gate
Aldgate East -- Tower Hill
Holborn Central -- Covent Garden
Hammersmith -- Acton Town
Hammersmith -- Turnham Green
Grange Hill -- Hainault
North Greenwich -- Canning Town
Gloucester Road -- South Kensington
Earl's Court -- High Street Kensington
Turnham Green -- Acton Town
Lambeth North -- Elephant & Castle
Ealing Common -- Ealing Broadway
Farringdon -- King's Cross St. Pancras
Barons Court -- Hammersmith
Bank -- Waterloo
Green Park -- Westminster
Euston -- Camden Town
Westminster -- Waterloo
Vauxhall -- Stockwell
Southwark -- London Bridge
Harrow-on-the-Hill -- Finchley Road
South Harrow -- Rayners Lane
Harrow-on-the-Hill -- Wembley Park
Stepney Green -- Whitechapel
Tottenham Court Road -- Leicester Square
Hyde Park Corner -- Knightsbridge
Earl's Court -- Barons Court
Aldgate -- Liverpool Street
Moorgate -- Bank
Bond Street -- Green Park
Moor Park -- Harrow-on-the-Hill
Waterloo -- Kennington
Stratford -- Mile End
Wembley Park -- Finchley Road
Piccadilly Circus -- Charing Cross
St. James's Park -- Westminster
Baker Street -- Edgware Road
Angel -- Old Street
Piccadilly Circus -- Green Park
Oxford Circus -- Bond Street
Euston Square -- Great Portland Street
Finchley Road -- Baker Street
```

Connectivity Verification

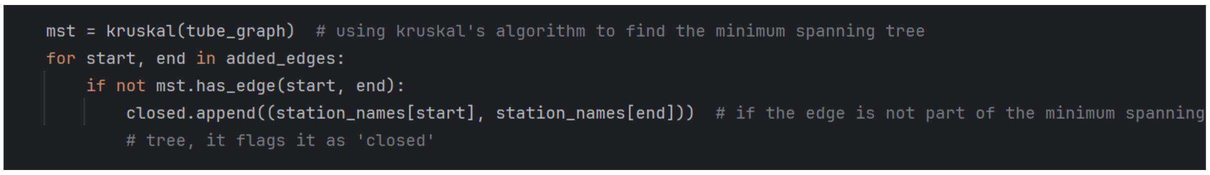
Kruskal’s algorithm has been used to ensure connectivity across the London Underground network after closures. By generating the Minimum Spanning Tree, we ensured that each station remains accessible from every other station, even if many sections are flagged as ‘closed’. The MST includes only the essential edges required to connect all stations, while the rest are marked as closed sections, maintaining the full connectivity.



The closures have an impact on the London Underground network in two ways:

1. In terms of connectivity, removing redundant connections makes the network more efficient [positive impact];
2. Longer travel times for certain trips due to the elimination of direct paths, resulting in passengers' inconvenience [negative impact].

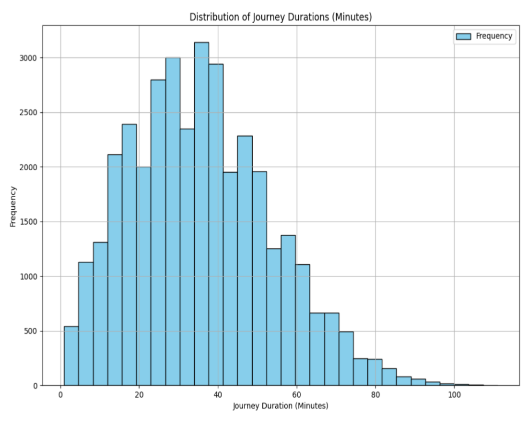
Code Implementation

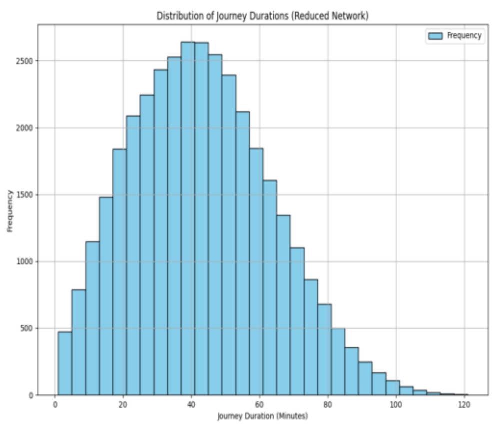


```
mst = kruskal(tube_graph)  # using kruskal's algorithm to find the minimum spanning tree
for start, end in added_edges:
    if not mst.has_edge(start, end):
        closed.append((station_names[start], station_names[end]))  # if the edge is not part of the minimum spanning
        # tree, it flags it as 'closed'
```

Impact Analysis of Line Section Closures

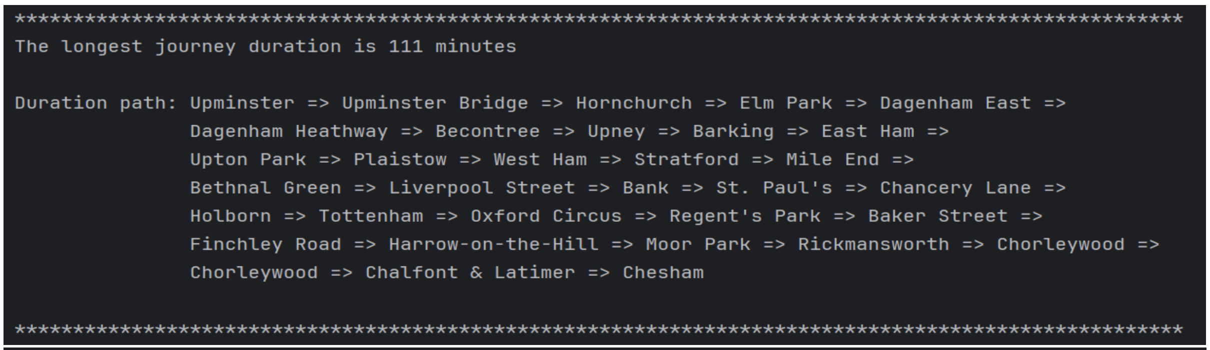
Histogram Comparison



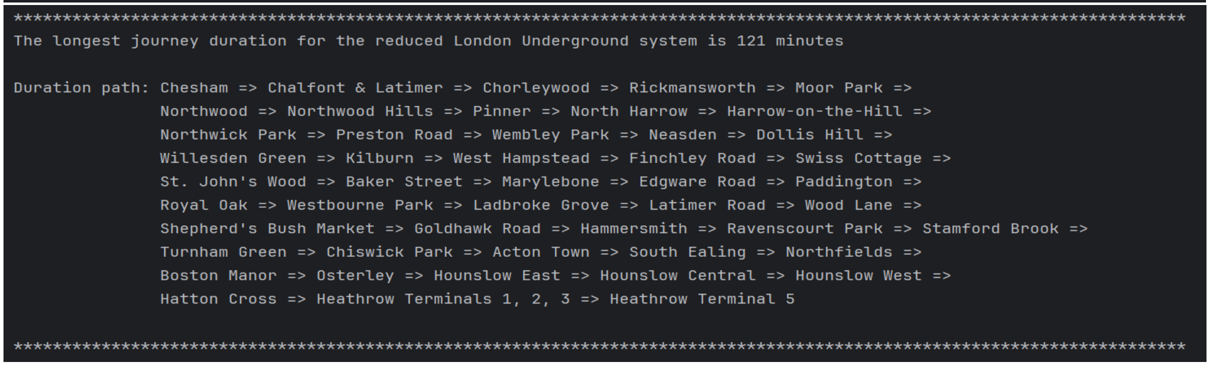


The histograms of journey durations before and after the closures highlight how much longer are trips in the reduced network. On the left-hand side, we can see how durations are more evenly distributed with a concentration around shorter journey times, while on the right-hand side we can see an increase in mid-to-longer journey durations, resulting in a negative impact on passengers' satisfaction.

Longest Path Comparison



```
****************************************************************************************************
The longest journey duration is 111 minutes

Duration path: Upminster => Upminster Bridge => Hornchurch => Elm Park => Dagenham East =>
               Dagenham Heathway => Becontree => Upney => Barking => East Ham =>
               Upton Park => Plaistow => West Ham => Stratford => Mile End =>
               Bethnal Green => Liverpool Street => Bank => St. Paul's => Chancery Lane =>
               Holborn => Tottenham => Oxford Circus => Regent's Park => Baker Street =>
               Finchley Road => Harrow-on-the-Hill => Moor Park => Rickmansworth => Chorleywood =>
               Chorleywood => Chalfont & Latimer => Chesham

****************************************************************************************************
```



```
************************************************************************************************************************
The longest journey duration for the reduced London Underground system is 121 minutes

Duration path: Chesham => Chalfont & Latimer => Chorleywood => Rickmansworth => Moor Park =>
               Northwood => Northwood Hills => Pinner => North Harrow => Harrow-on-the-Hill =>
               Northwick Park => Preston Road => Wembley Park => Neasden => Dollis Hill =>
               Willesden Green => Kilburn => West Hampstead => Finchley Road => Swiss Cottage =>
               St. John's Wood => Baker Street => Marylebone => Edgware Road => Paddington =>
               Royal Oak => Westbourne Park => Ladbroke Grove => Latimer Road => Wood Lane =>
               Shepherd's Bush Market => Goldhawk Road => Hammersmith => Ravenscourt Park => Stamford Brook =>
               Turnham Green => Chiswick Park => Acton Town => South Ealing => Northfields =>
               Boston Manor => Osterley => Hounslow East => Hounslow Central => Hounslow West =>
               Hatton Cross => Heathrow Terminals 1, 2, 3 => Heathrow Terminal 5

************************************************************************************************************************
```

The longest duration path in the reduced network is significantly longer than it was before in the original network with more transfers from a station to another, highlighting how reducing direct connections impacts the efficiency of travel and resulting in greater delays for passengers to reach their destinations.

The closures increase journey durations and due to fewer direct connections, there is more congestion on remaining routes, causing potential delays. Even if connectivity throughout the network is maintained using the Minimum Spanning Tree (MST), passengers might face longer and less convenient trips.

Progress Journal, Compliance with Instructions, Clarity of Language, etc.

0. Answer the following questions by keeping only one of the options (YES or NO) and removing the other.

a. Are the group members identical to those listed in (https://moodlecurrent.gre.ac.uk/pluginfile.php/4919652/mod_resource/content/12/CW_group_allocation_2024-25.pdf)? YES.
b. Have the final cumulative credits, which will appear in the final report, been sent to all members in good time to allow for any potential discussion (i.e., not left until the last minute near submission time)? YES.
c. Has a copy of the final report been sent to all the members prior to submission? YES.
d. Does any group member have EC (Extenuating Circumstances) approval? NO.
e. If you answered YES to the EC question above, please answer the following: Has the member's name been specified, and has evidence of the EC approval been included in this report (e.g., at the end)? YES / NO.

1. A summary of final credits of member contribution

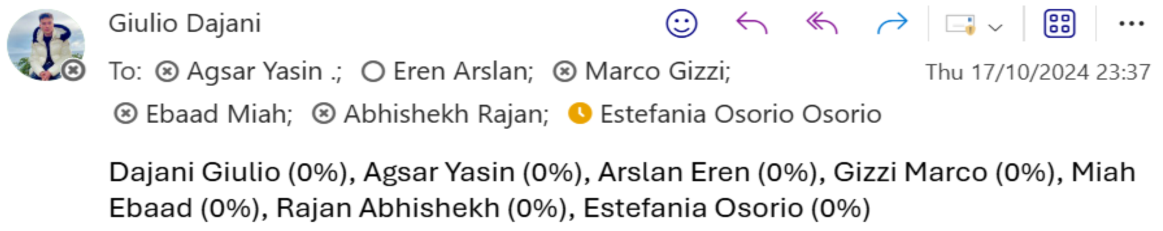
#	Name	ID in 9 digits	Login ID or Email (e.g. jk7492y)	Cumulative credit (0 - 100%)
1	Dajani Giulio	001343717	gd2482b@gre.ac.uk	% 100
2	Agsar Yasin	001328612	aa6632a@gre.ac.uk	% 78
3	Arslan Eren	001355719	ea7749d@gre.ac.uk	% 0
4	Gizzi Marco	001327359	mg1232y@gre.ac.uk	% 100
5	Miah Ebaad Ebaad	001294130	mm4879k@gre.ac.uk	% 82
6	Rajan Abhishekh	001319475	ar7811t@gre.ac.uk	% 80
7	Estefania Osorio Osorio	001413489	eo0139n@greenwich.ac.uk	% 82

2. Weekly progress log

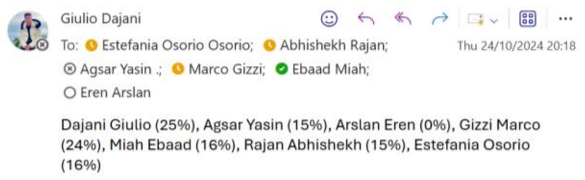
Week	**Brief description of each member's contributions; Confirmation of weekly email sent by each member; Attendance at weekly Teams meeting; Cumulative credit earned up to that week**
14/10/24	**Marco added all the members who replied to an email he sent us before to a WhatsApp group and we all have decided who will be the team leader. Giulio nominated also two co-leaders to help him mostly in emergency cases when he would be not available for any reason (Estefania and Marco). Giulio has in addition created a poll on WhatsApp for choosing the best day of the week to do our Teams call in addition of the lab meetings every Monday. The poll got the most votes on each Thursday of the week. We have started talking about the coursework specifications highlighting the tasks that should be done and we split them on a weekly basis. Each of us gave his/her opinion about the tasks and how can we tackle them in the best way possible. Everyone attended the Teams meeting besides Arslan Eren who hasn't communicated with us yet. Marco and Giulio are the first ones to implement their solutions for task1.**
21/10/24	**We have met on Monday the 21st to share ideas on how to do/implement task 1. Yasin, Ebaad, Abhishekh and Estefania produced their scripts using different approaches and explained them during the Teams call held on Thursday and Giulio shared feedback to each team member about any implementations that could have improved their solution. Someone has used Dijkstra algorithm. Everyone attended the Teams meeting besides of Arslan Eren which we haven't heard of him yet. After 2 hours-meeting, we agreed which algorithm to choose for the first task as the 'best' one which was a mix between Giulio's and Marco's solution and shared ideas on how to adjust the first task solution and prepare for doing the second one (2a + 2b).**
28/10/24	**We have met on Monday the 28th to share ideas on how to do/implement task 2. All of us have produced a solution for the current task and shared ideas on how to implement a solution which can be better in terms of performance and correctness. Everyone used a different approach, but the final algorithm used for this task is a mix between Giulio's and Marco's again and has been chosen during the meeting hold on Thursday 31st. Everyone attended the Teams meeting scheduled at 6pm besides of Arslan Eren which we still haven't heard of him yet. After 1 hour meeting, we started to think which approach to use for the subtask 3a.**

04/11/24	We have met on Monday the 4th to share ideas on how to do/implement task 3. Each of us has created his/her own solution by referring to the specifications for the subtasks 3a and 3b. Giulio's and Marco's solutions were the favourite ones for this task too because there have been some problems with plotting the graphs for the rest of the group, but everyone tried using different approaches. During the Teams call hold on Thursday 7th we have shared constructive feedback between us and understood how our solutions could still be implemented. Everyone attended the meeting scheduled at 6pm besides Arslan Eren and after more and less 1 hour and a half we have chosen the best algorithm and started thinking about the last task.
11/11/24	We have met on Monday the 11th to share ideas on how to do/implement task 4. After we have started creating our solutions, we sent them through WhatsApp and email. We shared opinions and tips about each of them for trying to implement them and make them better in terms of performance and correctness. For someone has been hard to understand the problem and follow the specifications but at the end everyone tried to make his own solution using a unique approach and that's the most important thing. The best one that we chose is again a mix between Giulio's and Marco's solution. Everyone attended the meeting besides of Arslan Eren, and it lasted 2 hours because we also talked and shared ideas on structuring the report.

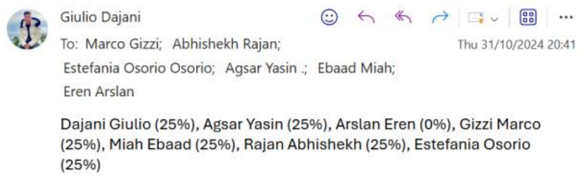
3. Evidence of a weekly email on cumulative credits and a Teams online meeting


Giulio Dajani
To: Agsar Yasin .; Eren Arslan; Marco Gizzi; Ebaad Miah; Abhishekh Rajan; Estefania Osorio Osorio
Thu 17/10/2024 23:37

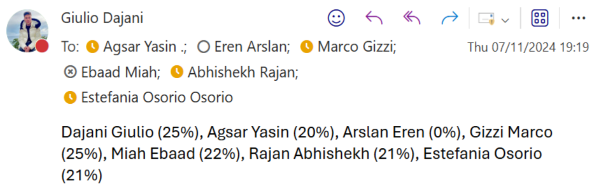
Dajani Giulio (0%), Agsar Yasin (0%), Arslan Eren (0%), Gizzi Marco (0%), Miah Ebaad (0%), Rajan Abhishekh (0%), Estefania Osorio (0%)


Giulio Dajani
To: Estefania Osorio Osorio; Abhishekh Rajan; Agsar Yasin .; Marco Gizzi; Ebaad Miah; Eren Arslan
Thu 24/10/2024 20:18

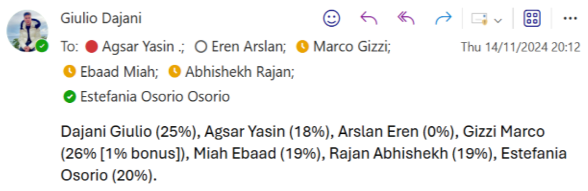
Dajani Giulio (25%), Agsar Yasin (15%), Arslan Eren (0%), Gizzi Marco (24%), Miah Ebaad (16%), Rajan Abhishekh (15%), Estefania Osorio (16%)


Giulio Dajani
To: Marco Gizzi; Abhishekh Rajan; Estefania Osorio Osorio; Agsar Yasin .; Ebaad Miah; Eren Arslan
Thu 31/10/2024 20:41

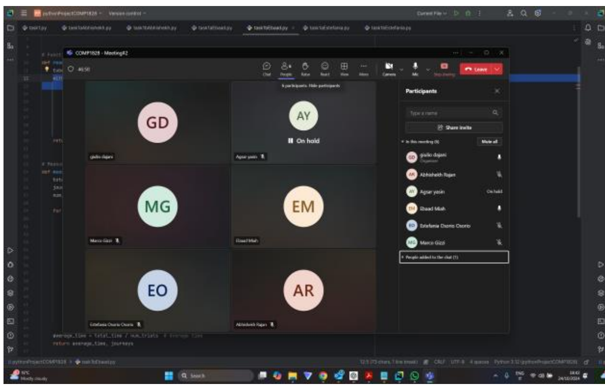
Dajani Giulio (25%), Agsar Yasin (25%), Arslan Eren (0%), Gizzi Marco (25%), Miah Ebaad (25%), Rajan Abhishekh (25%), Estefania Osorio (25%)

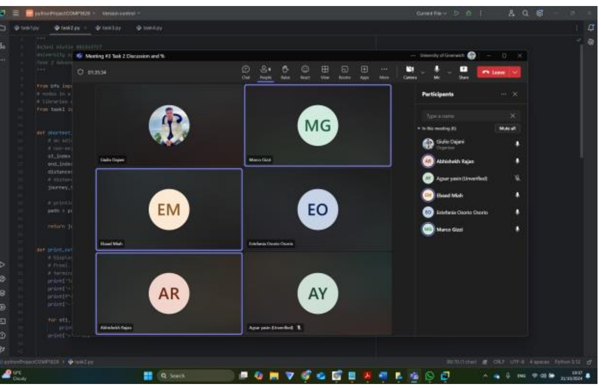

Giulio Dajani
To: Agsar Yasin .; Eren Arslan; Marco Gizzi; Ebaad Miah; Abhishekh Rajan; Estefania Osorio Osorio
Thu 07/11/2024 19:19

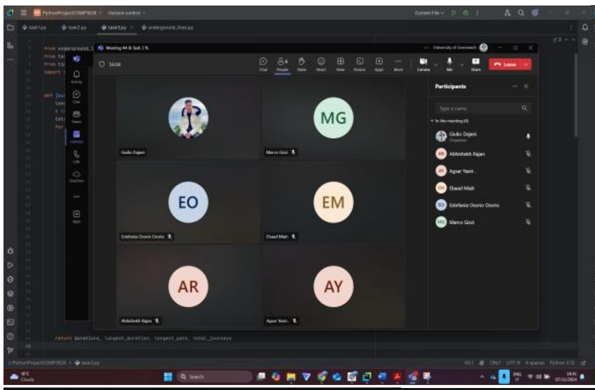
Dajani Giulio (25%), Agsar Yasin (20%), Arslan Eren (0%), Gizzi Marco (25%), Miah Ebaad (22%), Rajan Abhishekh (21%), Estefania Osorio (21%)

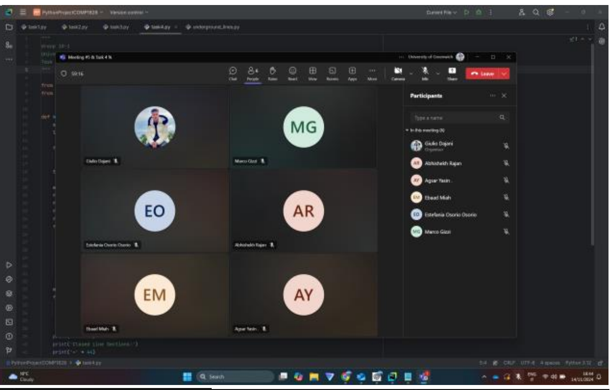

Giulio Dajani
To: Agsar Yasin .; Eren Arslan; Marco Gizzi; Ebaad Miah; Abhishekh Rajan; Estefania Osorio Osorio
Thu 14/11/2024 20:12

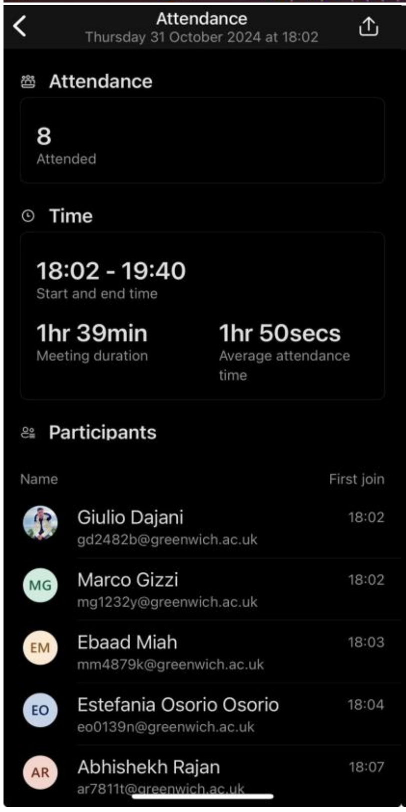
Dajani Giulio (25%), Agsar Yasin (18%), Arslan Eren (0%), Gizzi Marco (26% [1% bonus]), Miah Ebaad (19%), Rajan Abhishekh (19%), Estefania Osorio (20%).











Attendance
Thursday 31 October 2024 at 18:02

Attendance

8
Attended

Time

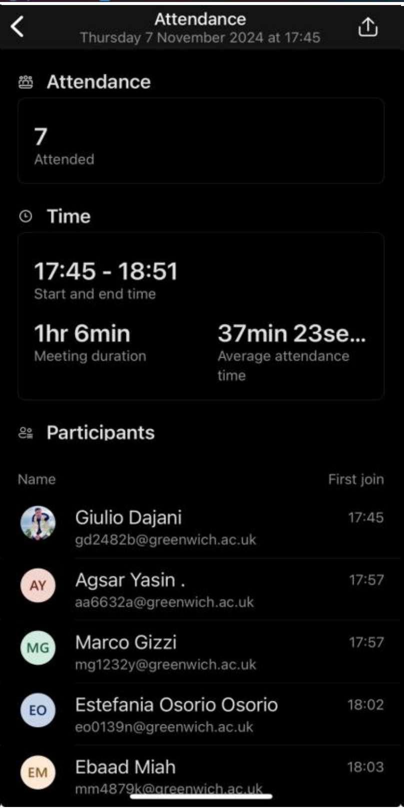
18:02 - 19:40
Start and end time

1hr 39min
Meeting duration

1hr 50secs
Average attendance time

Participants

Name	First join
Giulio Dajani gd2482b@greenwich.ac.uk	18:02
Marco Gizzi mg1232y@greenwich.ac.uk	18:02
Ebaad Miah mm4879k@greenwich.ac.uk	18:03
Estefania Osorio Osorio eo0139n@greenwich.ac.uk	18:04
Abhishekh Rajan ar7811t@greenwich.ac.uk	18:07



Attendance
Thursday 7 November 2024 at 17:45

Attendance

7
Attended

Time

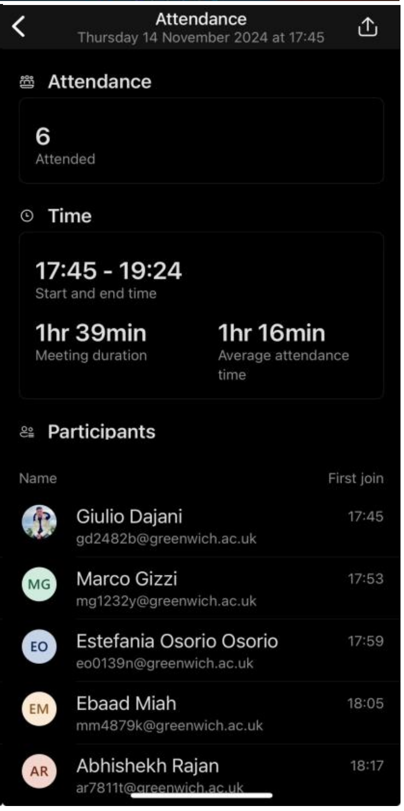
17:45 - 18:51
Start and end time

1hr 6min
Meeting duration

37min 23se...
Average attendance time

Participants

Name	First join
Giulio Dajani gd2482b@greenwich.ac.uk	17:45
Agsar Yasin . aa6632a@greenwich.ac.uk	17:57
Marco Gizzi mg1232y@greenwich.ac.uk	17:57
Estefania Osorio Osorio eo0139n@greenwich.ac.uk	18:02
Ebaad Miah mm4879k@greenwich.ac.uk	18:03



Attendance
Thursday 14 November 2024 at 17:45

Attendance

6
Attended

Time

17:45 - 19:24
Start and end time

1hr 39min
Meeting duration

1hr 16min
Average attendance time

Participants

Name	First join
Giulio Dajani gd2482b@greenwich.ac.uk	17:45
Marco Gizzi mg1232y@greenwich.ac.uk	17:53
Estefania Osorio Osorio eo0139n@greenwich.ac.uk	17:59
Ebaad Miah mm4879k@greenwich.ac.uk	18:05
Abhishekh Rajan ar7811t@greenwich.ac.uk	18:17