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

TASK 1 [20 marks]

(1A) Manual versus Code-Based Execution of the Algorithm-

Selected data structure and justification:

Using an Adjacency-List graph is the most appropriate model navigation system and efficient way to represent graphs. But it's not just the reason also, most preferrable is because each station store as a vertex, in the graph has an associated list containing all the vertices that the edges is directly connect as an underground line. Most suitable to store and manipulate this graph, an Adjacency list is employed as the principal data structure.

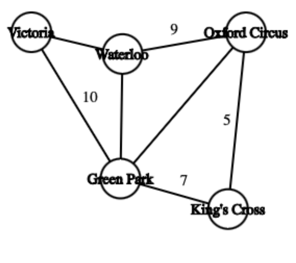
The London Underground System is represented as a graph, where nodes represent stations and edges represent connections between stations. We also considered using an Adjacency matrix, it is a 2D array V × V. Over here V is the number of stations. For V stations, the adjacency matrix requires V^2 space, because the size of the matrix grows quadratically with the number of vertices, resulting in a space complexity of O(V^2). And the London Underground System is a sparse graph. Whatsoever with thousands of stations, there are fewer direct connections between them compared to the number of potential connections. The matrix would be a sparse graph, in the end, a large portion of the cells contain no connection, and their adjacency matrix is going to take a large amount of memory. That means the memory isn't efficient. On the other hand, the Adjacency list is G(V,E) = O(V) + O(E) = O(V + E) space complexity. Where V is the number os station and E is the connection between the stations. Over here E is much smaller than V^2. Instead of using an adjacency matrix, using an adjacency list is more efficient.

Selected algorithm and Justification:

Using the Dijkstra's algorithm for finding the shortest paths between nodes in a weighted graph. A bid advantage of using Dijkstra's algorithm is just not only finding the shortest path, goes to every single node. "The greatest strength of Dijkstra's algorithm is that it is simple, and simple things are often the most effective." — Donald Knuth

There are a few more algorithms like Breadth First Search or Kruskal but instead of using these Dijkstra's is the most suitable one for London Underground System. Like Kruskal algorithm is not good for shortest path, it's designed for finding minimum spanning tree. But also, Kruskal had a larger space complexity algorithm in front of Dijkstra's algorithm. On the other hand, BFS suitable for the unweighted graph but not for finding the shortest path. Dijkstra's algorithm runs in O((V+E)logV), which is efficient for large graphs. The time complexity is more efficient to implement a priority queue (min-heap). Dijkstra's algorithm is the most accurate choice when you need to find the shortest path between a single source node and a target node in a weighted graph.

Manual Algorithm Execution:



Start node: King's Cross
End node: Victoria

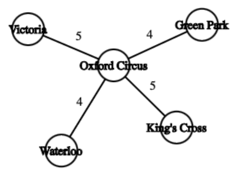
Step-1 (Starting Node)

Node	Distance from start	Previous node	Visited (Y/N)
King's Cross	0	X	Y
Waterloo	∞	X	N
Oxford Circus	∞	X	N
Green Park	∞	X	N
Victoria	∞	X	N

Step-2 (going to all the neighbouring nodes)

Node	Distance from start	Previous node	Visited (Y/N)
King's Cross	0	X	Y
Oxford Circus	5	King's Cross	Y
Green Park	7	King's Cross	Y
Waterloo	9	Oxford Circus	Y
Victoria	10	Green Park	Y

Shortest route from King's Cross to Victoria:
King's Cross -> Green Park -> Victoria with a weight of 10 minutes.
Starting with a different station-



Start node: Oxford Circus

End node: Victoria

Step-1 (Starting Node)

Node	Distance from start	Previous node	Visited (Y/N)
Oxford Circus	0	X	Y
King's Cross	∞	X	N
Green Park	∞	X	N
Waterloo	∞	X	N
Victoria	∞	X	N

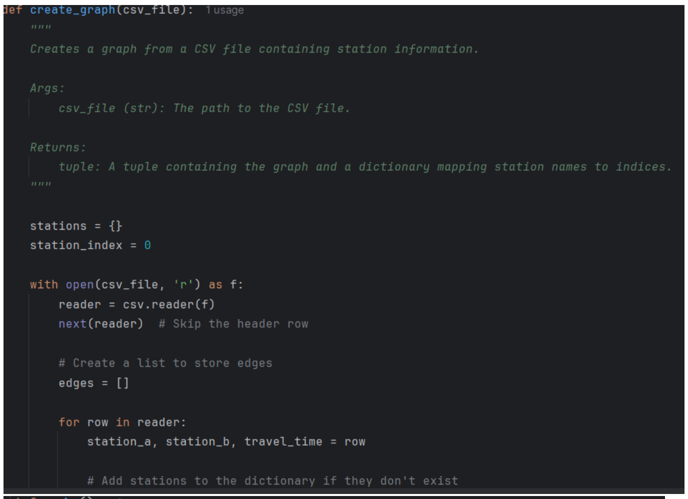
Step-2

Node	Distance from start	Previous node	Visited (Y/N)
Oxford Circus	0	X	Y
King's Cross	5	Oxford Circus	N
Green Park	4	Oxford Circus	Y
Waterloo	4	Oxford Circus	Y
Victoria	5	Oxford Circus	Y

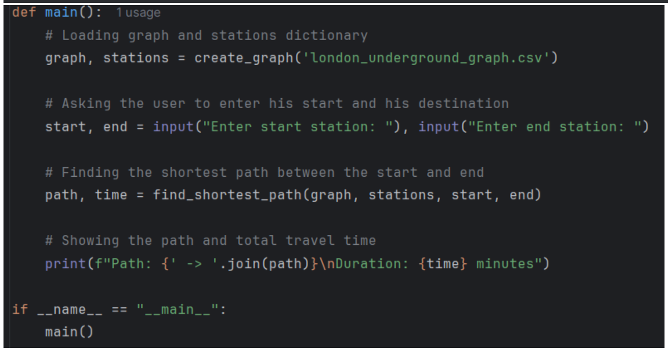
Shortest route from Oxford Circus to Victoria:

Oxford Circus -> Victoria with a weight of 5 minutes.

Code Implementation:



```python
def create_graph(csv_file):  1 usage
    """
    Creates a graph from a CSV file containing station information.

    Args:
        csv_file (str): The path to the CSV file.

    Returns:
        tuple: A tuple containing the graph and a dictionary mapping station names to indices.
    """

    stations = {}
    station_index = 0

    with open(csv_file, 'r') as f:
        reader = csv.reader(f)
        next(reader)  # Skip the header row

        # Create a list to store edges
        edges = []

        for row in reader:
            station_a, station_b, travel_time = row

            # Add stations to the dictionary if they don't exist
```



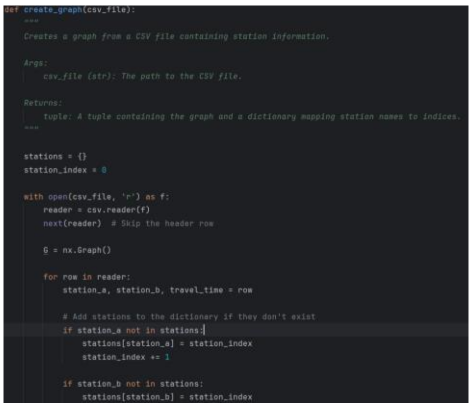
```python
def main():  1 usage
    # Loading graph and stations dictionary
    graph, stations = create_graph('london_underground_graph.csv')

    # Asking the user to enter his start and his destination
    start, end = input("Enter start station: "), input("Enter end station: ")

    # Finding the shortest path between the start and end
    path, time = find_shortest_path(graph, stations, start, end)

    # Showing the path and total travel time
    print(f"Path: {' -> '.join(path)}\nDuration: {time} minutes")

if __name__ == "__main__":
    main()
```

Comparison of manual and code-generated results:

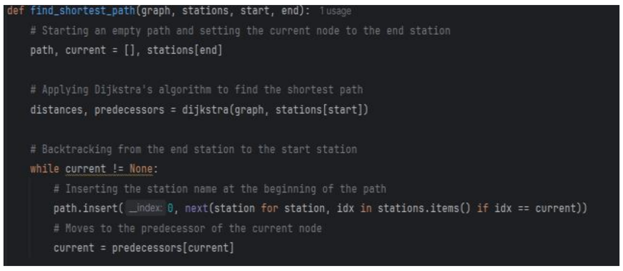
The identical shortest routes and distances will be produced by both code-generated and manual results. This is so because Dijkstra's algorithm, which determines the shortest path between nodes in a weighted graph, is used in both methods. No matter how many times the algorithm is run for the same inputs, Dijkstra's algorithm will always identify the path with the lowest cumulative weight and produce the same result for the shortest path since it is deterministic.

1B) Empirical Measurement of Time Complexity:

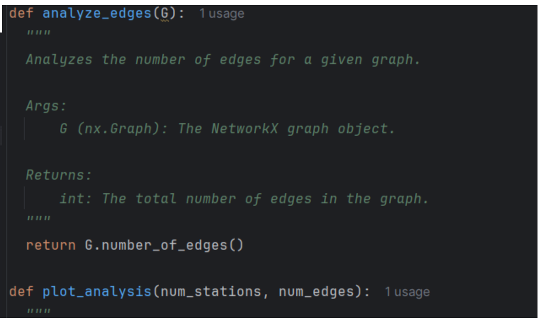
Artificial Network Generation code snippets:



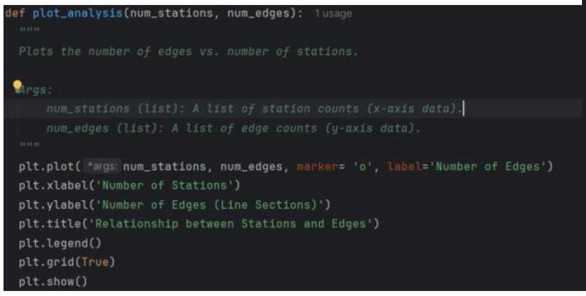
```python
def create_graph(csv_file):
    """
    Creates a graph from a CSV file containing station information.

    Args:
        csv_file (str): The path to the CSV file.

    Returns:
        tuple: A tuple containing the graph and a dictionary mapping station names to indices.
    """

    stations = {}
    station_index = 0

    with open(csv_file, 'r') as f:
        reader = csv.reader(f)
        next(reader)  # Skip the header row

        G = nx.Graph()

        for row in reader:
            station_a, station_b, travel_time = row

            # Add stations to the dictionary if they don't exist
            if station_a not in stations:
                stations[station_a] = station_index
                station_index += 1

            if station_b not in stations:
                stations[station_b] = station_index
```



```python
def find_shortest_path(graph, stations, start, end):  1 usage
    # Starting an empty path and setting the current node to the end station
    path, current = [], stations[end]

    # Applying Dijkstra's algorithm to find the shortest path
    distances, predecessors = dijkstra(graph, stations[start])

    # Backtracking from the end station to the start station
    while current != None:
        # Inserting the station name at the beginning of the path
        path.insert( __index: 0, next(station for station, idx in stations.items() if idx == current))
        # Moves to the predecessor of the current node
        current = predecessors[current]
```

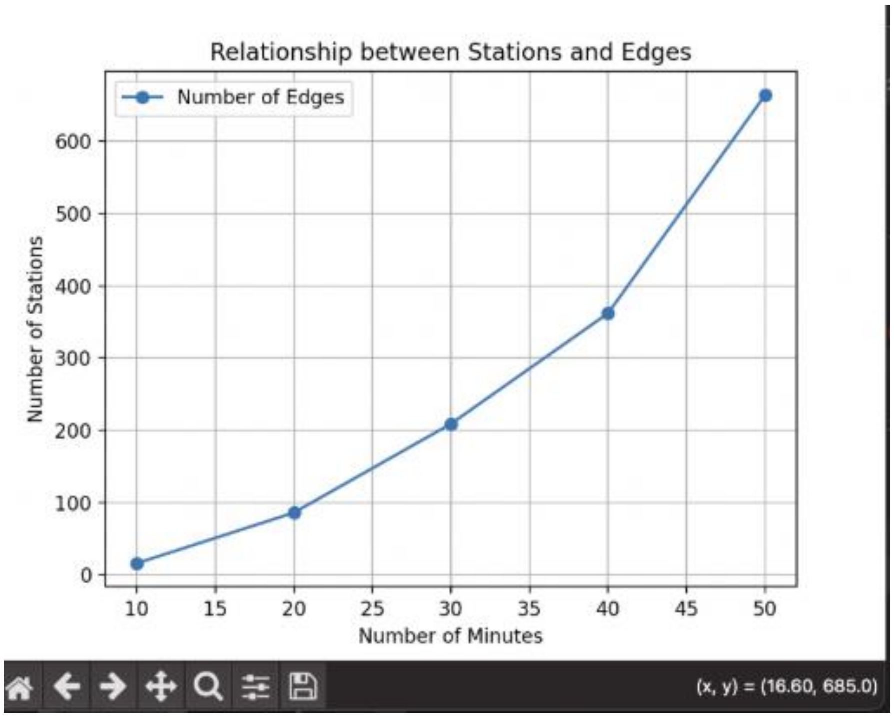


```
def analyze_edges(G):  1 usage
    """
    Analyzes the number of edges for a given graph.

    Args:
        G (nx.Graph): The NetworkX graph object.

    Returns:
        int: The total number of edges in the graph.
    """
    return G.number_of_edges()

def plot_analysis(num_stations, num_edges):  1 usage
    """
```



```
def plot_analysis(num_stations, num_edges):  1 usage
    """
    Plots the number of edges vs. number of stations.

    Args:
        num_stations (list): A list of station counts (x-axis data).
        num_edges (list): A list of edge counts (y-axis data).
    """
    plt.plot( *args: num_stations, num_edges, marker= 'o', label='Number of Edges')
    plt.xlabel('Number of Stations')
    plt.ylabel('Number of Edges (Line Sections)')
    plt.title('Relationship between Stations and Edges')
    plt.legend()
    plt.grid(True)
    plt.show()
```

Time Complexity Graph:



As you can see from the graph above, this graph follows a positive correlation between the two variables. As the number of minutes increase, the number of stations also increase.

TASK 2

(2A) Manual versus Code-Based Execution of the Algorithm-

Selected data algorithm and justification: Adjacency List Graph

Efficiency: For sparse graphs (i.e., when there are fewer edges than vertices), which are common in transit networks like the London Underground, an adjacency list uses less space than an adjacency matrix.

Scalability: An adjacency list is the best option for working with a big network (many stations and routes) because it only saves the edges that are required, allowing the graph representation to expand as the system does.

Selected data structure and justification: Dijkstra's Algorithm

Shortest Path: In graphs with non-negative edge weights, Dijkstra's algorithm is the best choice for determining the shortest path. In your situation, Dijkstra's algorithm can effectively determine the shortest path in terms of the number of pauses between stations because all the weights are equal to one.

Application to Sparse Networks: Because the graph structure is simple to navigate and update, Dijkstra's algorithm performs well on sparse networks, such as the London Tube, particularly when combined with an adjacency list.

Simple Dataset: [Include a clear image (scanned if hand-drawn, or a screenshot if digitally created) showing station names and journey durations between neighbouring stations]

Manual Algorithm Execution: [Show detailed step-by-step calculations for finding the shortest path between a chosen station pair]

Leicester Square to Bond Street

Node	Distance from start	Previous node	Visited (Y/N)
Leicester Square	0	X	N
Bond Street	∞	X	N
Oxford Circus	∞	X	N
Piccadilly Circus	∞	X	N
Tottenham Court Road	∞	X	N
Green Park	∞	X	N

Neighbors of Leicester Square

Node	Distance from start	Previous node	Visited (Y/N)

Leicester Square	0	X	Y
Bond Street	∞	X	N
Oxford Circus	1	Leicester Square	N
Piccadilly Circus	1	Leicester Square	N
Tottenham Court Road	1	Leicester Square	N
Green Park	∞	X	N

Neighbors of Oxford Circus, Piccadilly Circus, and Tottenham Court Road

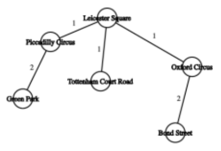
Node	Distance from start	Previous node	Visited (Y/N)
Leicester Square	0	X	Y
Bond Street	2	Oxford Circus	N
Oxford Circus	1	Leicester Square	Y
Piccadilly Circus	1	Leicester Square	Y
Tottenham Court Road	1	Leicester Square	Y
Green Park	2	Piccadilly Circus	N

Neighbors of Green Park

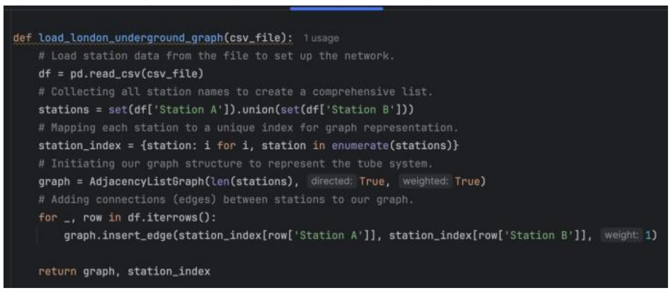
Node	Distance from start	Previous node	Visited (Y/N)
Leicester Square	0	X	Y
Bond Street	2	Oxford Circus	N
Oxford Circus	1	Leicester Square	Y
Piccadilly Circus	1	Leicester Square	Y
Tottenham Court Road	1	Leicester Square	Y
Green Park	2	Piccadilly Circus	Y

Shortest route from Leicester Square to Green Park:

Leicester Square -> Piccadilly Circus -> Green Park -> Bond Street

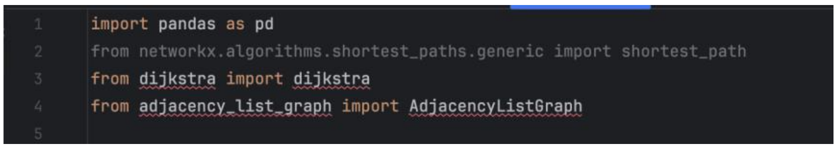


Code Implementation: [Provide key code fragments (not the full code), specifically:]The part of your Python code that implements the dataset:



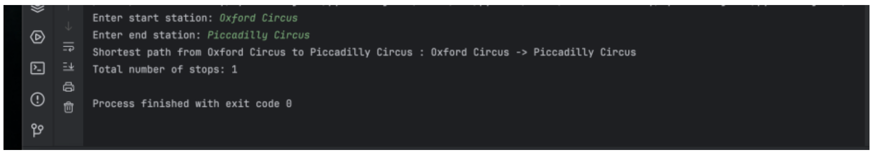
```python
def load_london_underground_graph(csv_file):  1 usage
    # Load station data from the file to set up the network.
    df = pd.read_csv(csv_file)
    # Collecting all station names to create a comprehensive list.
    stations = set(df['Station A']).union(set(df['Station B']))
    # Mapping each station to a unique index for graph representation.
    station_index = {station: i for i, station in enumerate(stations)}
    # Initiating our graph structure to represent the tube system.
    graph = AdjacencyListGraph(len(stations),  directed: True,  weighted: True)
    # Adding connections (edges) between stations to our graph.
    for _, row in df.iterrows():
        graph.insert_edge(station_index[row['Station A']], station_index[row['Station B']],  weight: 1)

    return graph, station_index
```

The section where you call or use the required library code, as verification of compliance:



```
import pandas as pd
from networkx.algorithms.shortest_paths.generic import shortest_path
from dijkstra import dijkstra
from adjacency_list_graph import AdjacencyListGraph
```

The output showing the shortest route and journey duration



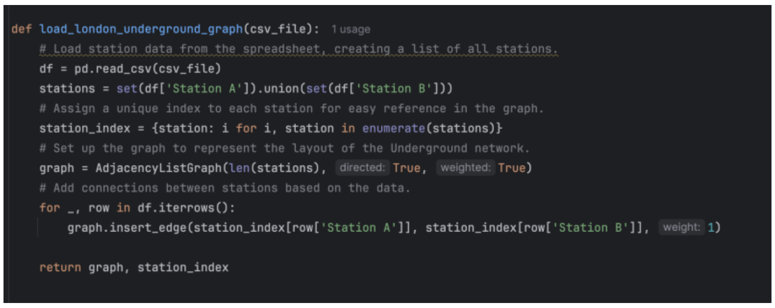
```
Enter start station: Oxford Circus
Enter end station: Piccadilly Circus
Shortest path from Oxford Circus to Piccadilly Circus : Oxford Circus -> Piccadilly Circus
Total number of stops: 1

Process finished with exit code 0
```

Comparison: [Compare manual and code-generated results, explaining any differences]

There are some key differences between the manual results, as shown in the table, and the code-generated results, one being that in the table/graph, you can physically see the shortest path for example between Oxford Circus and Piccadilly Circus, which is one stop away from each other, however the code uses Dijkstra's algorithm to find the shortest path between stations and then outputs the results. Another difference between the manual and code-generated results is that the manual results are shown in a weighted graph, whereas the code-generated results use an adjacency list graph to represent the data. An adjacency list graph is efficient as well as flexible for the code-generated results as it allows for a smooth traversal to the other stations which is needed for Dijkstra's algorithm.

(2B) Empirical Measurement of Time Complexity

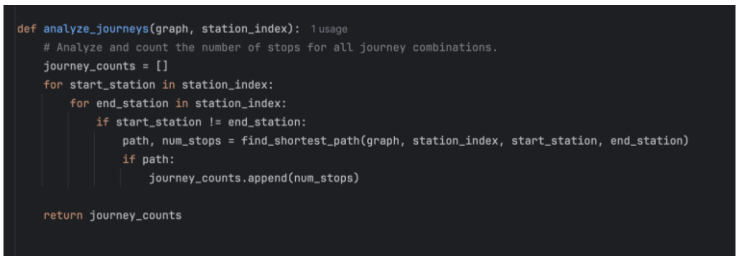
Artificial Network Generation: [Present key code snippets (not full code) for generating an artificial tube network dataset. Include proper references if using external sources]:



```
def load_london_underground_graph(csv_file):  1 usage
    # Load station data from the spreadsheet, creating a list of all stations.
    df = pd.read_csv(csv_file)
    stations = set(df['Station A']).union(set(df['Station B']))
    # Assign a unique index to each station for easy reference in the graph.
    station_index = {station: i for i, station in enumerate(stations)}
    # Set up the graph to represent the layout of the Underground network.
    graph = AdjacencyListGraph(len(stations), directed: True, weighted: True)
    # Add connections between stations based on the data.
    for _, row in df.iterrows():
        graph.insert_edge(station_index[row['Station A']], station_index[row['Station B']], weight: 1)

    return graph, station_index
```

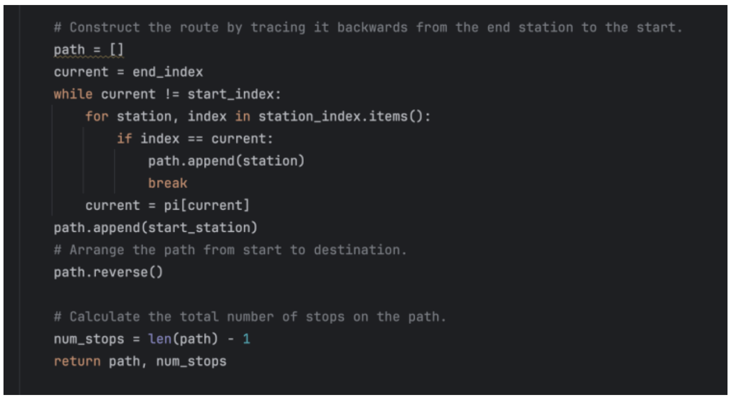
Execution Time Measurement: [Present relevant code snippets of the following steps or something equivalent:]

Randomly selecting a pair of stations



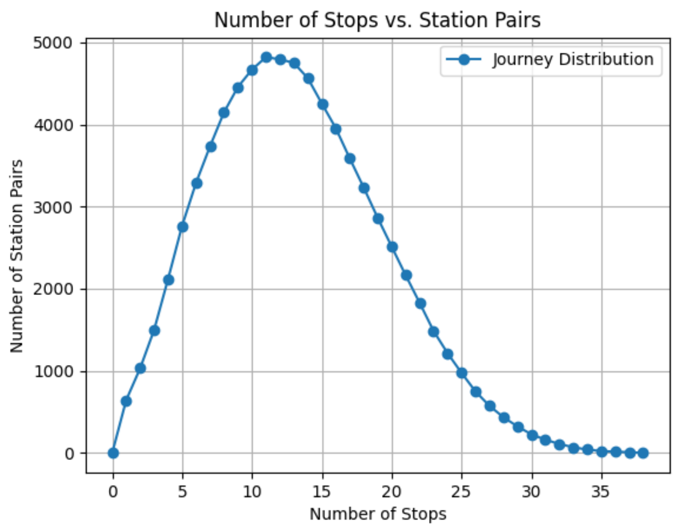
```
def analyze_journeys(graph, station_index):  1 usage
    # Analyze and count the number of stops for all journey combinations.
    journey_counts = []
    for start_station in station_index:
        for end_station in station_index:
            if start_station != end_station:
                path, num_stops = find_shortest_path(graph, station_index, start_station, end_station)
                if path:
                    journey_counts.append(num_stops)

    return journey_counts
```

Computing the journey duration for the selected pair



```
# Construct the route by tracing it backwards from the end station to the start.
path = []
current = end_index
while current != start_index:
    for station, index in station_index.items():
        if index == current:
            path.append(station)
            break
    current = pi[current]
path.append(start_station)
# Arrange the path from start to destination.
path.reverse()

# Calculate the total number of stops on the path.
num_stops = len(path) - 1
return path, num_stops
```

Time Complexity Graph: [Plot a graph of average execution time vs network size n (where n is the total number of stations)]



For each network of size n, specify the corresponding total number of line sections between adjacent stations in that particular network.

State the tool used for graph generation

Matplotlib

Analysis: [Compare the plotted graph to the algorithm's theoretical time complexity, discussing any discrepancies or alignments]

Dijkstra’s time complexity is O((|V|+|E|)log|V|+|V|) due to the worst-case scenario that all stations may need to be visited. However, the plotted graph’s time complexity is said to be O(n) with ‘n’ being the number of stations visited. The adjacency list is a big factor of the algorithm’s theoretical space complexity and the journey counts, giving a space complexity of O(|V|2+|E|). This is also due to Dijkstra’s algorithm needing a priority queue which adds to the worst-case scenario for space complexity

TASK 3

[20 marks]

(3A) Journey Duration Histograms and Longest Path (In Minutes)

Method used to import London Underground Data.xlsx

Importing the London Underground data by letting the user select the Excel file through a file dialog using `tkinter`. Then, I use `openpyxl` to open the file and read the active sheet. Starting from the third row, I extract the station names and travel times, skipping any rows with missing data. Finally, I build a graph where stations are nodes and the connections between them, with their travel times, are edges. This graph is then used for analyzing the network

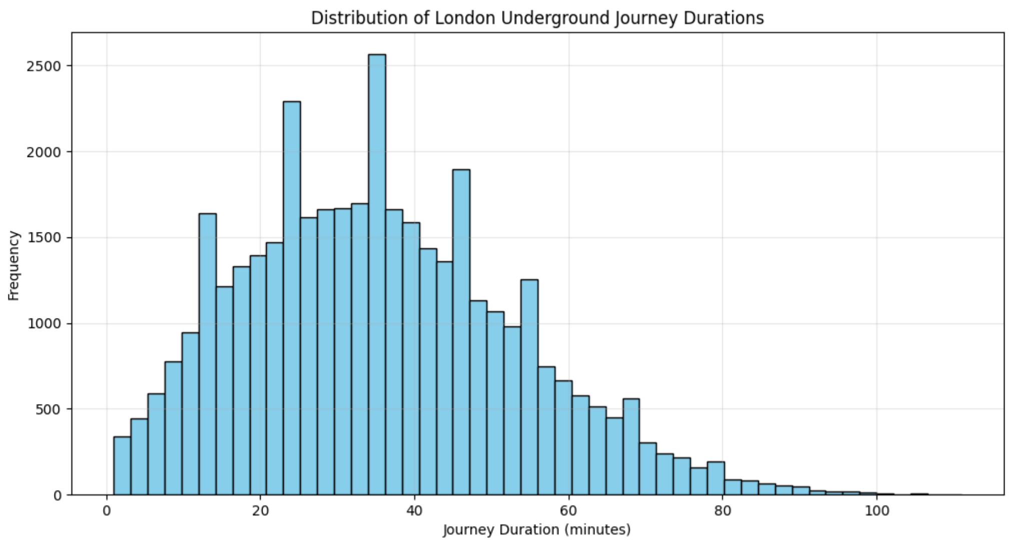
Journey Duration Calculations:

Total number of journeys calculated is 39060

Duplicate journeys included/excluded: Included as some joineries may be different with what direction you travel, but some may be the same, resulting in a duplicate.

Histogram:

Matplotlib is used to plot the histogram of journey durations. Matplotlib is a Python library that makes it easy to create visualizations like graphs and charts. In my code, I used the `plt.hist()` function to take the list of journey durations and turn it into a histogram. The histogram shows how these durations are distributed.I set the `bins=50` argument, which means I divided the data into 50 groups. This helps to get a more detailed view of how the durations are spread out. If I used fewer bins, the data would appear more general, but using more bins gives a clearer picture of where most journeys fall. Finally, I used `plt.show()` to display the graph. Without this, the plot wouldn't appear when the code runs. This step makes sure the histogram is shown to visualize the data



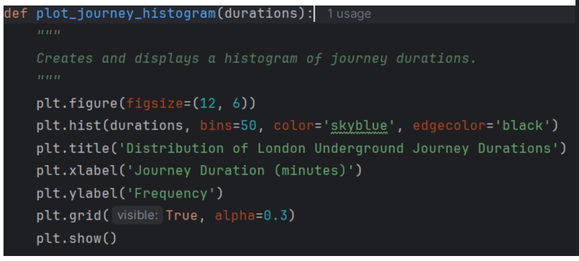
This histogram shows a distribution of London Underground journey durations. Most journeys are relatively short, with a smaller number of longer journeys. The majority of journeys fall within the 20-30 minute range.

Longest Journey:

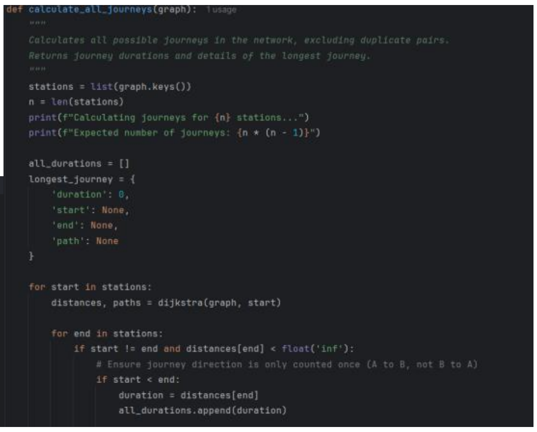
Duration: 111 minutes

Path: Chesham → Chalfont & Latimer → Rickmansworth → Moor Park → Harrow-on-hill → Finchley Road → Baker Street → Bond Street → 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 → Upminster

Code Implementation:



```
def plot_journey_histogram(durations):  1 usage
    """
    Creates and displays a histogram of journey durations.
    """
    plt.figure(figsize=(12, 6))
    plt.hist(durations, bins=50, color='skyblue', edgecolor='black')
    plt.title('Distribution of London Underground Journey Durations')
    plt.xlabel('Journey Duration (minutes)')
    plt.ylabel('Frequency')
    plt.grid( visible: True, alpha=0.3)
    plt.show()
```



```
def calculate_all_journeys(graph):  1 usage
    """
    Calculates all possible journeys in the network, excluding duplicate pairs.
    Returns journey durations and details of the longest journey.
    """
    stations = list(graph.keys())
    n = len(stations)
    print(f"Calculating journeys for {n} stations...")
    print(f"Expected number of journeys: {n * (n - 1)}")

    all_durations = []
    longest_journey = {
        'duration': 0,
        'start': None,
        'end': None,
        'path': None
    }

    for start in stations:
        distances, paths = dijkstra(graph, start)

        for end in stations:
            if start != end and distances[end] < float('inf'):
                # Ensure journey direction is only counted once (A to B, not B to A)
                if start < end:
                    duration = distances[end]
                    all_durations.append(duration)
```

Analysis: The code calculates all possible journeys on the London Underground network and visualizes the distribution of journey durations using a histogram. The histogram shows that most journeys are short, with a few longer ones. The longest journey information reveals the potential for very long trips within the network.

(3b)

Journey Duration Histograms and Longest Path (by Number of Stops)

Journey Duration Calculations:

- Total number of journey durations calculated: 78120

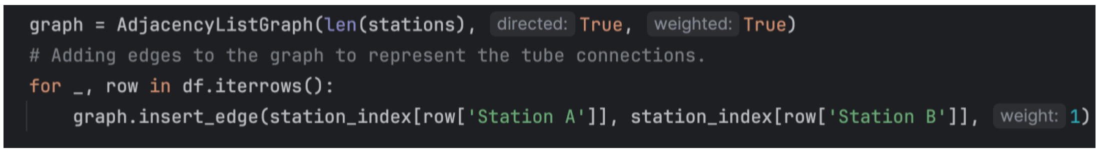
- Duplicate journeys included/excluded: Some journeys may differ depending on the direction of travel, while others might be identical in both directions, resulting in duplicates.
- Histogram:
- Method used to plot the histogram:

Matplotlib is used to plot a histogram the length of the trips. The plt.show() displays the histogram when the code runs.

Longest Journey:

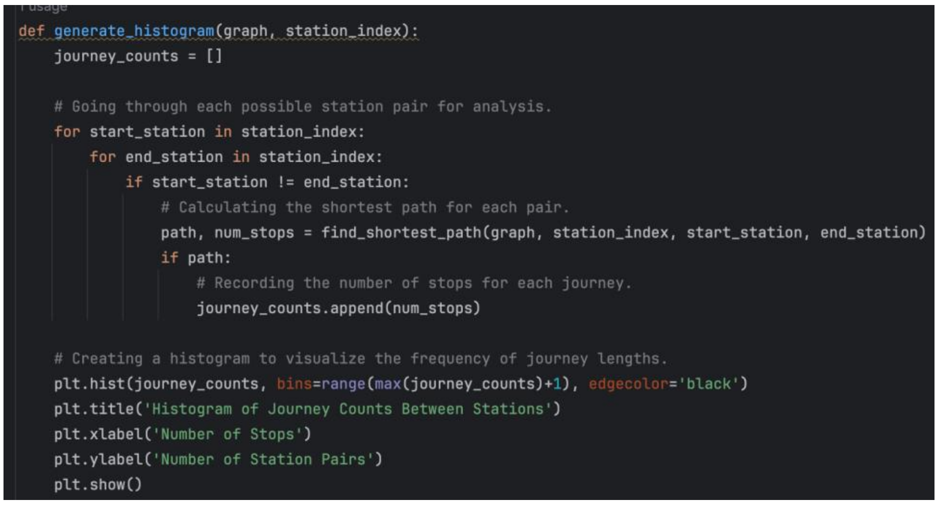
- Number of stops: 39060
- Path: Chesham -> Chalfont & Latimer -> Chorleywood -> Rickmansworth -> Moor Park -> Harrow-on-the-Hill -> Finchley Road -> Baker Street -> Bond Street -> Oxford Circus -> Tottenham -> Holborn -> Chancery Lane -> St. Paul's -> Bank -> Liverpool Street -> Bethnal Green -> Mile 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 -> Upminster

Code Implementation:



```
graph = AdjacencyListGraph(len(stations), directed: True, weighted: True)
# Adding edges to the graph to represent the tube connections.
for _, row in df.iterrows():
    graph.insert_edge(station_index[row['Station A']], station_index[row['Station B']], weight: 1)
```

One difference between the two codes for task 3 is that in code 3b, the graph is represented using an adjacency list within a custom AdjacencyListGraph class. Each station is represented as an element in a list, and the connections to other stations are stored as sub-lists. Whereas, in code 3a, the graph is represented by a dictionary where each station is a key. The stations it connects to are listed as keys within a nested dictionary, with the travel times as the values.



```
def generate_histogram(graph, station_index):
    journey_counts = []

    # Going through each possible station pair for analysis.
    for start_station in station_index:
        for end_station in station_index:
            if start_station != end_station:
                # Calculating the shortest path for each pair.
                path, num_stops = find_shortest_path(graph, station_index, start_station, end_station)
                if path:
                    # Recording the number of stops for each journey.
                    journey_counts.append(num_stops)

    # Creating a histogram to visualize the frequency of journey lengths.
    plt.hist(journey_counts, bins=range(max(journey_counts)+1), edgecolor='black')
    plt.title('Histogram of Journey Counts Between Stations')
    plt.xlabel('Number of Stops')
    plt.ylabel('Number of Station Pairs')
    plt.show()
```

Another difference between the two codes is that code 3b is about the number of stops during a journey between the different stations. On the other hand, code 3a focuses on the travel times of the journey, how long it takes to travel from one station to another.

Comparison with 3a:

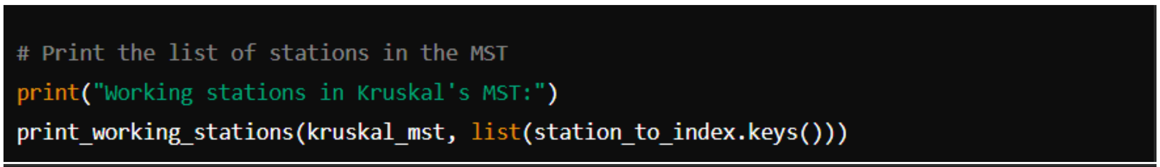
One key difference between the results from code 3b and code 3a is that in code 3a, a journey with fewer stops could take longer if the travel times are shorter, a journey with more stops in code 3b could still be faster.

TASK 4

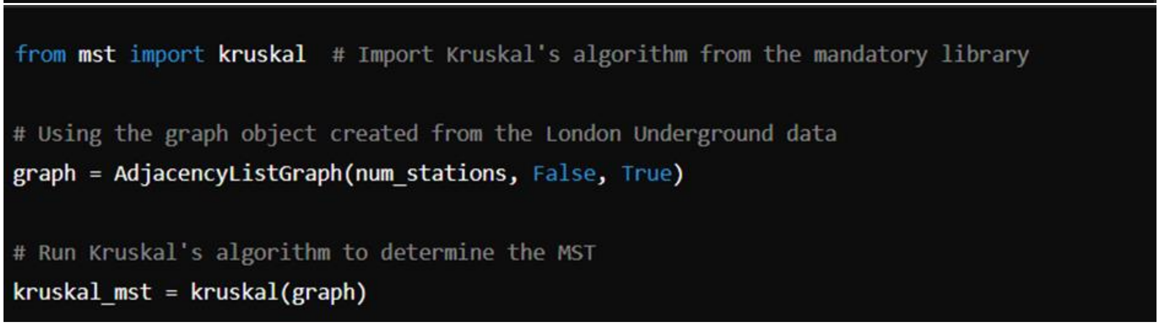
(4A) Line Section Closure Analysis-

Selected Algorithm:

Kruskal's algorithm due to its effectiveness in managing sparse graphs—like the London Underground network—where the network has significantly fewer connections than there are potential connections between stations, Kruskal's algorithm was chosen for this task. It minimises the number of active connections and retains low memory usage by sorting the edges first and selecting those that form a minimum spanning tree (MST) without generating cycle



```python
# Print the list of stations in the MST
print("Working stations in Kruskal's MST:")
print_working_stations(kruskal_mst, list(station_to_index.keys()))
```



```python
from mst import kruskal  # Import Kruskal's algorithm from the mandatory library

# Using the graph object created from the London Underground data
graph = AdjacencyListGraph(num_stations, False, True)

# Run Kruskal's algorithm to determine the MST
kruskal_mst = kruskal(graph)
```

Closed Line Sections: Piccadilly Circus -- Green Park
Baker Street -- Regent's Park
Aldgate - Aldgate East
Camden Town -Chalk Farm
Highgate - Archway
Station A - Station B

Connectivity Verification:

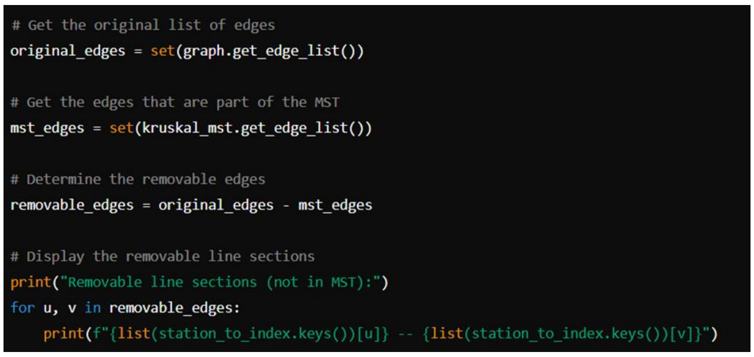
The resulting graph from Kruskal's algorithm was confirmed to be connected in order to ensure that travel between any two stations is still practical following the closures. In addition, we made sure that every station could still connect to every other station via the MST's remaining edges. This was confirmed by checking connectivity across all stations using a traversal algorithm (such as BFS or DFS) on the MST.

Analysis:

The redundant connections that can be eliminated without compromising the network's overall connectivity are represented by the closures found by Kruskal's algorithm. While guaranteeing that passengers' ability to travel between stations is unaffected, these closures aid in minimising maintenance expenses and possible points of failure in the system.

Code Implementation:

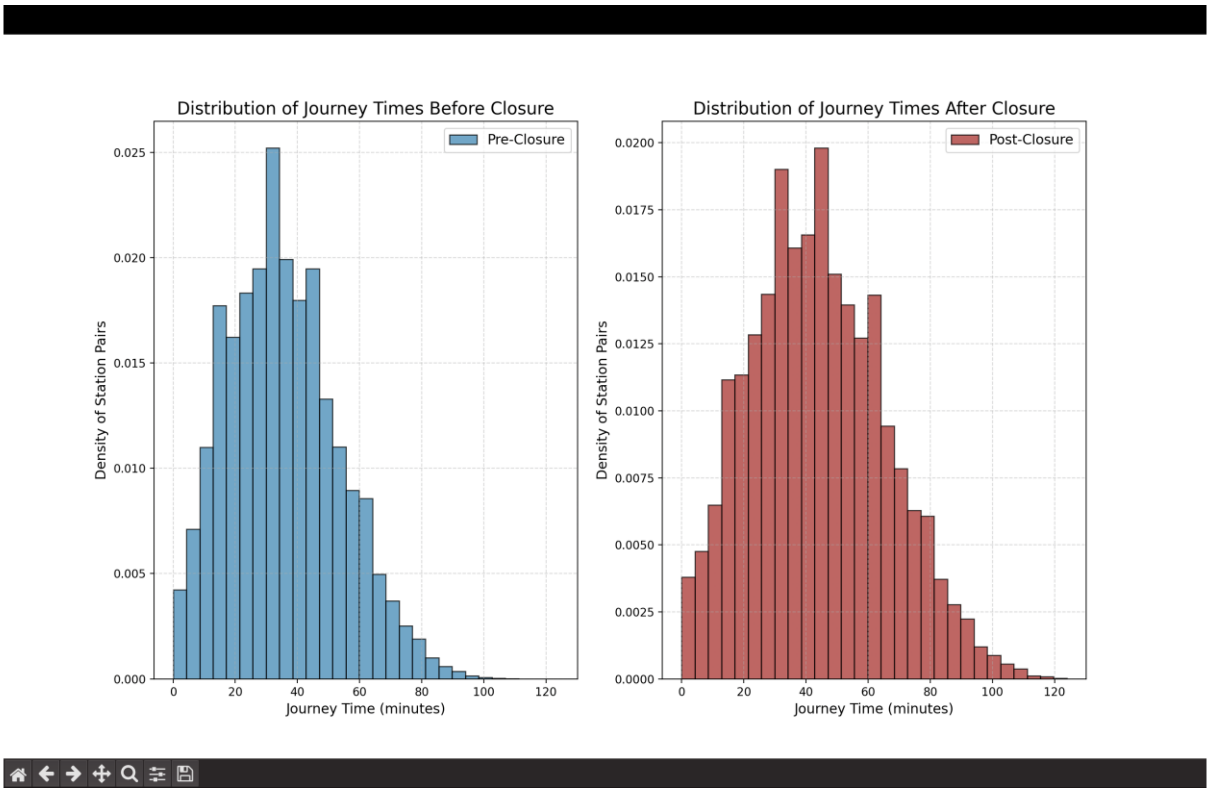
The following code snippet demonstrates how the removable line sections were determined



```python
# Get the original list of edges
original_edges = set(graph.get_edge_list())

# Get the edges that are part of the MST
mst_edges = set(kruskal_mst.get_edge_list())

# Determine the removable edges
removable_edges = original_edges - mst_edges

# Display the removable line sections
print("Removable line sections (not in MST):")
for u, v in removable_edges:
    print(f"{list(station_to_index.keys())[u]} -- {list(station_to_index.keys())[v]}")
```

(4B) Impact Analysis of Line Section Closures-

Histogram Comparison:



Analysis- highlighting key differences and insights:

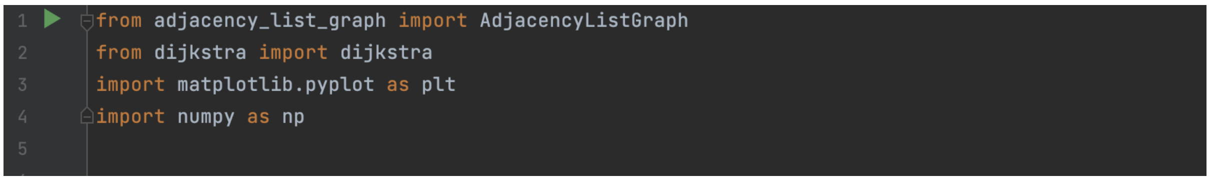
Pre-Closure Network: Most of the station pairs exhibit shorter journey times, as showing by the higher density in the lower range of the histogram. This reflects an efficient and well-connected route. The range of journey times is an efficient network with minimal detours. The peak density is concentrated at shorter journey times, underscoring the prevalence of efficient connections across the network.

Impact Analysis: This histogram indicates that the London Underground network is highly efficient, with most traffic concentrated in the minimal delays and reduced travel issues. The network is operating at an optimal level, evenly distributing passenger flow, and minimizing overreliance on specific routes. Also allowing quick access between key stations and less crowded areas

Post-Closure Network: The journey time distribution tooks longer time and fewer station with shorter journeys. Also, the longer journey times becoming more frequent. highlighting an increase in average journey times and a decline in network efficiency.

Impact Analysis: Passengers going to reach most of the station but the efficiency of their travel has diminished. The overall shape remains similar, journey times indicates a degradation. But the efficiency of their travel has deminished. Passengers travelling between major nodes or some significant delays between stations. The longer travel times contributes to reduce satisfaction among travellers. Journey times among the system's ability to adapt, though at the cost of increased congestion and extended travel times for many passengers.

Code Implementation:



```python
from adjacency_list_graph import AdjacencyListGraph
from dijkstra import dijkstra
import matplotlib.pyplot as plt
import numpy as np

```

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

[20 marks]

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 / NO.**

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 / NO.**

YES

   c. **Has a copy of the final report been sent to all the members prior to submission? YES / NO.**
   YES
   d. **Does any group member have EC (Extenuating Circumstances) approval? YES / NO.**
      i. **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.**

   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	Md Abdul Raihan Tanzim	001341954	Mt9498y@gre.ac.uk	100%
2	Ali Wais	001324310	wa9286w	100 %
3	Choudhury Halima	001307499	hc1190m	90%
4	Mathias, Joel	001340447	jm7243r	100 %

5	Bemath, Nadiyah	001309079	nb1875x	70 %
6	Obiorah Unique	001355908	co6461t	60%

(Member 1 should be the group leader.)

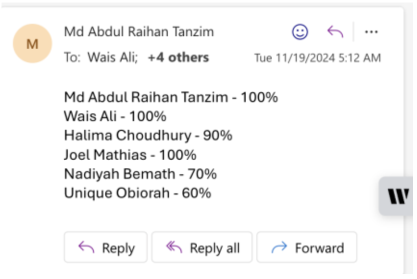
2. Weekly progress log

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
27/Oct -	Halima and Tanzim completed task 1 and 2 together. There was also a team's meeting carried out where everyone attended. While Halima and Tanzim were completing task 1 and 2, the rest of the team were dividing up the tasks and deciding who would do what.
10/Nov -	Joel started task 3a and Nadiyah also started task 3b. Another weekly teams meeting took place where everyone also attended except Halima as she was working but was updated after the meeting. Meanwhile, Wais started to do task 4a and Unique also started to do task 4b.
15/Nov -	This week, the report was started by the people who had their code completed and working. Again, there was a weekly Teams meeting where everyone attended, and we discussed the next steps and the progress of the report as all of the codes were working and completed. As this was the last few days until the deadline, we were mainly focusing on the report and ensuring that it was completed up to a high standard.

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

For each week, you need to provide the two pieces of evidence:

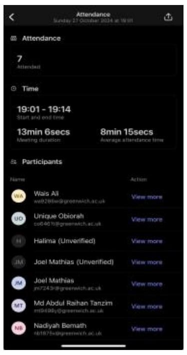
- Screenshot of the group leader's email to all members, showing: Sender, Recipients, Date, and Content (names and cumulative credits). For instance:


Md Abdul Raihan Tanzim
To: Wais Ali; +4 others
Tue 11/19/2024 5:12 AM

Md Abdul Raihan Tanzim - 100%
Wais Ali - 100%
Halima Choudhury - 90%
Joel Mathias - 100%
Nadiyah Bemath - 70%
Unique Obiorah - 60%

- Screenshot of Teams meeting (on a non-Monday). For instance,

- Week 27/Oct - :


Attendance
Sunday 27 October 2024 at 19:01

Attendance

7
Attended

Time

19:01 - 19:14
Start and end time

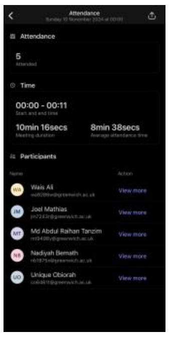
13min 6secs
Meeting duration

8min 15secs
Average attendance time

Participants

Name	Action
Wais Ali wa9288w@greenwich.ac.uk	View more
Unique Obiorah co6461t@greenwich.ac.uk	View more
Halima (Unverified)	View more
Joel Mathias (Unverified)	View more
Joel Mathias jm7243r@greenwich.ac.uk	View more
Md Abdul Raihan Tanzim mt9498y@greenwich.ac.uk	View more
Nadiyah Bemath nb1875x@greenwich.ac.uk	View more

- Week 10/Nov - :


Attendance
Sunday 10 November 2024 at 00:00

Attendance

5
Attended

Time

00:00 - 00:11
Start and end time

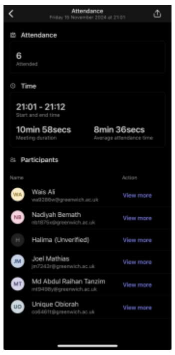
10min 16secs
Meeting duration

8min 38secs
Average attendance time

Participants

Name	Action
Wais Ali wa9288w@greenwich.ac.uk	View more
Joel Mathias jm7243r@greenwich.ac.uk	View more
Md Abdul Raihan Tanzim mt9498y@greenwich.ac.uk	View more
Nadiyah Bemath nb1875x@greenwich.ac.uk	View more
Unique Obiorah co6461t@greenwich.ac.uk	View more

- Week 15/Nov - :


Attendance
Friday 15 November 2024 at 21:01

Attendance

6
Attended

Time

21:01 - 21:12
Start and end time

10min 58secs
Meeting duration

8min 36secs
Average attendance time

Participants

Name	Action
Wais Ali wa9288w@greenwich.ac.uk	View more
Nadiyah Bemath nb1875x@greenwich.ac.uk	View more
Halima (Unverified)	View more
Joel Mathias jm7243r@greenwich.ac.uk	View more
Md Abdul Raihan Tanzim mt9498y@greenwich.ac.uk	View more
Unique Obiorah co6461t@greenwich.ac.uk	View more