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 Algorithm:

  Data Structure: NumPy 2D Array

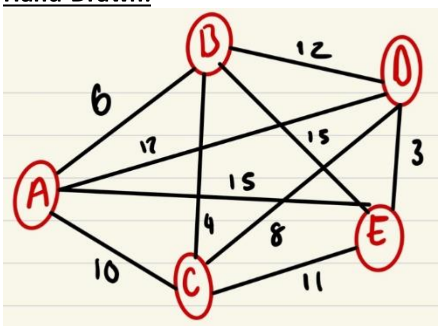
  The reason behind using NumPy 2D Arrays as our Data structure to hold our stations is that it allows us to use mathematical operations such as matrices which directly correlates with 2D Arrays. This allows for faster and simpler matrix operations within the Floyd-Warshall function.

  Simple Dataset:

  Digital Screenshot:

```
"""
-----------------------------
Train Station Connections   -
-----------------------------
    A     B     C     D     E -
A   0     6     ∞     12    ∞ -
B   6     0     4     ∞     ∞ -
C   ∞     4     0     8     ∞ -
D   12    ∞     8     0     3 -
E   ∞     ∞     ∞     3     0 -
-----------------------------
"""
```

- **Hand-Drawn:**



- **Manual Algorithm Execution:**

	A	B	C	D	E
A	0	6	10	12	15
B	6	0	4	12	15
C	10	4	0	8	11
D	12	12	8	0	3
E	15	15	11	3	0

$A \to C$: $A \xrightarrow{6} B \xrightarrow{4} C$: 10 minutes
$A \to E$: $A \xrightarrow{12} D \xrightarrow{3} E$: 15 minutes
$B \to E$: $B \xrightarrow{4} C \xrightarrow{8} D \xrightarrow{3} E$: 15 minutes
$C \to E$: $C \xrightarrow{8} D \xrightarrow{3} E$: 11 minutes
$D \to B$: $D \xrightarrow{8} C \xrightarrow{4} B$: 12 minutes

- **Code Implementation:**
  - **Implementing the data set:**

```python
train_stations = np.array([ # Create an array that holds the different stations.
    [0,6,NA,12,NA], # Station: A
    [6,0,4,NA,NA], # Station: B
    [NA,4,0,8,NA], # Station: C
    [12,NA,8,0,3], # Station: D
    [NA,NA,NA,3,0] # Station: E
])
```

- **Required libraries used:**

```python
from Coursework.Task1CLRS.floyd_warshall import floyd_warshall
import numpy as np
```

```
UndergroundSystem F:\ADSCoursework\
  Coursework
    Task1CLRS
      all_pairs_shortest_paths.py
      floyd_warshall.py
      johnson.py
      print_all_pairs_shortest_path.py
    main.py
```

(Included file path for verification)

- **Output showing the shortest route and journey duration:**

```
F:\ADSCoursework\.venv\Scripts\python.exe F:\ADSCoursework\UndergroundSystem\Coursework\main.py
[[ 0.  6. 10. 12. 15.]
 [ 6.  0.  4. 12. 15.]
 [10.  4.  0.  8. 11.]
 [12. 12.  8.  0.  3.]
 [15. 15. 11.  3.  0.]]
```

- **Comparison:**

Manual Approach:

- The disadvantage of this approach is that it is time-consuming, as you would need to go through each station. This also means that when using this approach for a larger network, it would take a significant amount of time and effort.

Code-generated Approach:

- In this approach, we use the Floyd-Warshall algorithm to automatically calculate the shortest path between all stations. Using the algorithm makes it much easier and quicker to find the shortest path between stations. The algorithm returns a matrix of shortest-path weights for all pairs of vertices. (Cormen, et al., 2022)

Differences:

- This approach also allows for larger datasets to be processed more efficiently, with its time complexity of O(n^3), because it involves three nested loops that go through the nodes of the graph. (Singh, 2024). However, the manual approach is almost impractical for large datasets, such as the London Underground system.

(1b) Empirical Measurement of Time Complexity

- **Artificial Network Generation:**

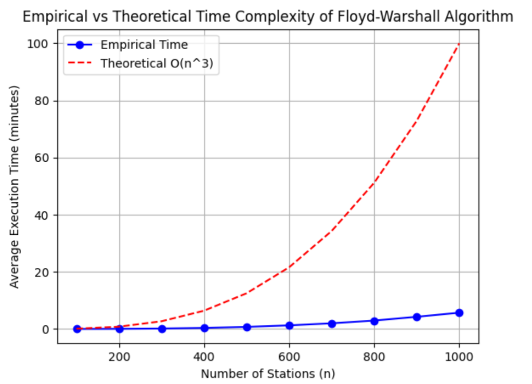
```python
# Function to generate a matrix representing the stations
def generate_matrix(number_of_stations):
    # Create an n x n matrix filled with NA (infinity) representing disconnected stations
    default_station = np.full((number_of_stations, number_of_stations), NA)
    # Set the diagonal to 0 (distance from a station to itself is 0)
    np.fill_diagonal(default_station, 0)
    return default_station  # Return the generated matrix


# Function to populate the station matrix with random journey times
def populate_stations(matrix):
    # Iterate over all pairs of stations to assign journey times
    for i in range(len(matrix)):  # Iterate through each station
        for j in range(i + 1, len(matrix)):  # Iterate through stations after station i
            station_weight = random.randint(1, MAX_DURATION)  # Generate a random journey time
            matrix[i][j] = station_weight  # Set journey time between station i and station j
            matrix[j][i] = station_weight  # Ensure symmetry: station j to station i has the same journey time
    return matrix  # Return the populated matrix
```

- **Execution Time Measurement:**

```python
# Function to measure the execution time of the Floyd-Warshall algorithm
def measure_execution_time(network_size, runs=10):
    total_time = 0  # Initialize the total time variable

    for _ in range(runs):
        # Generate the matrix representing the tube network for the given number of stations
        matrix_maker = generate_matrix(network_size)
        # Populate the matrix with random journey times
        station_creation = populate_stations(matrix_maker)

        # Measure the start time before running the Floyd-Warshall algorithm
        start_time = time.time()
        # Run Floyd-Warshall algorithm to calculate shortest paths
        shortest_paths = floyd_warshall(station_creation, len(station_creation))
        # Measure the time taken and accumulate it
        total_time += (time.time() - start_time)

    # Calculate the average time per run in seconds, then convert to minutes
    avg_time = total_time / runs
    avg_time_minutes = avg_time / 60  # Convert to minutes
    return round(avg_time_minutes, 5)  # Return average time rounded to 5 decimal places
```

```python
if __name__ == "__main__":
    execution_times = []  # List to store the average execution times
    test_cases = [100, 200, 300, 400, 500, 600, 700, 800, 900, 1000]  # Define test cases for different networ

    # Measure execution times for each network size (empirical data)
    for test_case in test_cases:
        avg_time = measure_execution_time(test_case)  # Measure time for Floyd-Warshall
        execution_times.append(avg_time)  # Append the measured time to the execution_times list

        # Print the average execution time for this network size
        print(f"Average execution time for station sizes of {test_case}: {avg_time} minutes")

    # Theoretical O(n^3) time complexity
    # This is the expected time complexity for Floyd-Warshall algorithm. Scaling for comparison
    theoretical_times = [(n ** 3) / 10 ** 7 for n in test_cases]

    # Plot the empirical data
    plt.plot(test_cases, execution_times, marker='o', linestyle='-', color='b', label='Empirical Time')

    # Plot the theoretical O(n^3) curve
    plt.plot(test_cases, theoretical_times, linestyle='--', color='r', label="Theoretical O(n^3)")

    # Add labels, title, and legend to the plot
    plt.xlabel('Number of Stations (n)')  # Set the x-axis label
    plt.ylabel('Average Execution Time (minutes)')  # Set the y-axis label
    plt.title('Empirical vs Theoretical Time Complexity of Floyd-Warshall Algorithm')  # Set the plot title
    plt.grid(True)  # Add grid to the plot
    plt.legend()  # Display the legend

    # Display the plot
    plt.show()
```

- **Time Complexity Graph:**



The tool used for the graphing was matplotlib. Matplotlib allows you to generate plots, histograms, bar charts, scatter plots, etc., with just a few lines of code. (Babitz, 2023)

Analysis:

- The red dashed line represents the theoretical time complexity of the Floyd-Warshall algorithm, growing as $O(n^3)$. The curve rises steeply as the number of stations increases, indicating that the complexity increases significantly with larger networks.

- The blue line with markers shows the empirical execution times for various station sizes. While it also increases as the network grows, it rises more gradually compared to the theoretical curve, indicating better real-world performance.

- In conclusion, both theoretical and empirical results reflect cubic growth. However, the empirical times increase more slowly in the tested range but may align more closely with the theoretical curve for networks beyond 1000 stations. This highlights how worst-case complexity predictions overestimate execution time in practical scenarios.

TASK 2 [20 marks]

```python
# Updated matrix where each value represents the number of stops (1 for direct connections)
train_stations_stops = np.array([  # Create an array that holds the number of stops between stations
    [0, 1, NA, 1, NA],  # Station A
    [1, 0, 1, NA, NA],  # Station B
    [NA, 1, 0, 1, NA],  # Station C
    [1, NA, 1, 0, 1],  # Station D
    [NA, NA, NA, 1, 0]  # Station E
])

# Use Floyd-Warshall to calculate the shortest path in terms of number of stops
shortest_paths_stops = floyd_warshall(train_stations_stops, len(train_stations_stops))

# Output the shortest paths between stations in terms of number of stops
print("Shortest paths in terms of number of stops:")
print(shortest_paths_stops)
```

```
Shortest paths in terms of number of stops:
[[0. 1. 2. 1. 2.]
 [1. 0. 1. 2. 3.]
 [2. 1. 0. 1. 2.]
 [1. 2. 1. 0. 1.]
 [2. 3. 2. 1. 0.]]
```

```
Shortest path in terms of minutes:
[[ 0.  6. 10. 12. 15.]
 [ 6.  0.  4. 12. 15.]
 [10.  4.  0.  8. 11.]
 [12. 12.  8.  0.  3.]
 [15. 15. 11.  3.  0.]]
```

Analysis

To achieve the goal of finding the shortest path through number of stops rather than time, we had to change the matrix to show the direct connections between stations rather than time. For example, traveling from Station A to Station C requires 2 stops. This focuses on how many stations changes or transfers a passenger must make, rather than how long the trip takes. By abstracting away time, we can focus purely on the structure of the network and number of connections. When comparing both solutions we can see that the direct routes are often similar. Station A to B is 1 stop and 6 minutes. For routes that have more stops such as A to C the shortest path requires 2 stops but in terms of time takes 10 minutes. In this case, minimizing stops and minimizing time led to the same route, since both measures result in a direct connection. However, when we consider routes with more stops, such as traveling from station A to station C, the differences become clearer.

In conclusion, the shortest paths prioritise minimising stops whilst shortest path based on minutes aim to reduce travel time. However, there can be discrepancies between the two. For instance, B to E shows 3 stops with a travel time of 15 minutes showing how a route may have fewer stops but a much longer journey duration.

Code Changed for task 2B:

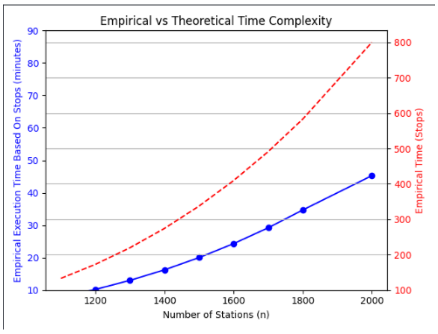
```python
def generate_stops_matrix(number_of_stations):  1 usage  Aman
    # Create an n x n matrix filled with NA (infinity) representing disconnected stations
    stops_matrix = np.full( shape: (number_of_stations, number_of_stations), NA)
    # Set the diagonal to 0 (no stops within the same station)
    np.fill_diagonal(stops_matrix,  val: 0)
    return stops_matrix  # Return the generated matrix

# Function to populate the station matrix with random stops
# 1 stop will be added between randomly connected stations
def populate_stops_matrix(matrix, connection_density=0.3):  1 usage  Aman
    for i in range(len(matrix)):
        for j in range(i + 1, len(matrix)):
            if random.random() < connection_density:  # Randomly connect stations with a certain probability
                matrix[i][j] = 1  # Add 1 stop for connected stations
                matrix[j][i] = 1  # Ensure symmetry
    return matrix
```

For Task 2, we adapted the code from Task 1 to calculate the shortest path based on the number of stops instead of travel time in minutes.

```python
if __name__ == "__main__":
    execution_times = []  # List to store the average execution times
    test_cases = [1100, 1200, 1300, 1400, 1500, 1600, 1700, 1800, 1900, 2000]  # Network sizes for Task 2B

    # Measure execution times for each network size (empirical data)
    for test_case in test_cases:
        avg_time = measure_execution_time_stops(test_case)  # Measure time based on number of stops
        execution_times.append(avg_time)  # Append the measured time to the execution_times list

        # Print the average execution time for this network size based on number of stops
        print(f"Average execution time for station size of {test_case} (based on stops): {avg_time} minutes")

    # Theoretical O(n^3) time complexity
    # This is the expected time complexity for Floyd-Warshall algorithm. Scaling for comparison
    theoretical_times = [(n ** 3) / 10 ** 7 for n in test_cases]

    plt.plot(test_cases, execution_times, marker='o', linestyle='-', color='b', label='Empirical Time (Stops)')

    # Plot the theoretical O(n^3) curve
    plt.plot(test_cases, theoretical_times, linestyle='--', color='r', label="Theoretical O(n^3)")

    # Add labels, title, and legend to the plot
    plt.xlabel('Number of Stations (n)')  # Set the x-axis label
    plt.ylabel('Average Execution Time (minutes)')  # Keep this as we are measuring execution time
    plt.title('Empirical vs Theoretical Time Complexity of Floyd-Warshall (Shortest Path by Stops)')  # Update the title
    plt.grid(True)  # Add grid to the plot
    plt.legend()  # Display the legend

    # Display the plot
    plt.show()
```

Updated test cases and x and y values to display time it takes for the shortest path through stations.

Graph for 2b



Analysis

When comparing both graphs we can see that the empirical time grows with increasing network size. Task 1 (Duration) results in lower empirical times compared to Task 2 (Stops). This shows that calculating shortest paths based on the number of stops takes longer.

However, the scale on in Task 2 is much larger ranging from 1200 to 2000 stations which would naturally take longer than task 1s scale of 100 to 1000. Therefore, both graphs show how empirical times are significantly lower than the theoretical curve of O(n^3) but as Task 2 gets closer to 2000 stations we can see the time taken is much longer.

In conclusion, in a real-world scenario, users may prefer shortest duration through minutes rather than stops as it would often be shorter than stops, proven by Task 2A.

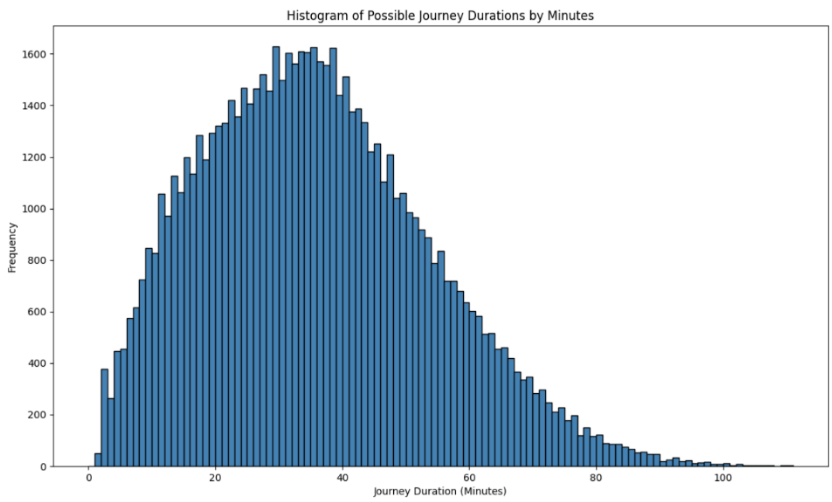
TASK 3 [20 marks]

(3a) Journey Duration Histograms and Longest Path (in Minutes)

- **Data Import Method:**
  I have chosen to use the (pandas, n.d.) library as it allows for ease of accessibility as well as conducting changes to the file.

- **Journey Duration Calculations:**
  - Total number of journey durations calculated: 353
  - Duplicate journeys included/excluded: Included
- **Histogram:**

  I used the library matplotlib assisted with pandas to create the histogram. The histogram shows the different journey durations and how many journeys also share that duration.
- **Longest Journey:**
  - Duration: 111.0 minutes
  - 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 → Chalfont & Latimer → Chesham



- **Code Implementation:**

```python
# Add connections (edges) to the graph based on journey data
for _, row in underground_data.iterrows():
    start_station_index = station_to_index[row["Starting_Station"]]
    end_station_index = station_to_index[row["Ending_Station"]]
    travel_duration = row["Journey_Duration"]

    # Only add the connection if it doesn't already exist
    if not station_graph.has_edge(start_station_index, end_station_index):
        station_graph.insert_edge(start_station_index, end_station_index, travel_duration)

# Function to reconstruct the path from the predecessors list
1 usage  new *
def reconstruct_path(predecessors_list, source_station, destination_station):
    path_trace = []
    current_station = destination_station
    while current_station != source_station:
        path_trace.insert( __index: 0, current_station)
        current_station = predecessors_list[current_station]
    path_trace.insert(0, source_station)  # Add the source station to the path
    return path_trace

# Function to calculate the number of stops between every pair of stations
1 usage  new *
def get_stops_between_all_stations(station_graph, unique_stations):
    stops_count = []
    total_stations = len(unique_stations)

    for source_station in range(total_stations):
        # Get the list of predecessors for the current source station using Dijkstra
        predecessors_list = dijkstra(station_graph, source_station)[1]

        # For each destination station, calculate the number of stops
        for destination_station in range(total_stations):
            if predecessors_list[destination_station] and source_station != destination_station:
                reconstructed_path = reconstruct_path(predecessors_list, source_station, destination_station)
                stops_count.append(len(reconstructed_path) - 1)  # Number of stops is path length - 1

    return stops_count
```

- **Analysis:**

  From the histogram we can see that most journey durations take between 20 and 40 minutes. But we can see that as the journey durations increase there are fewer journeys who have longer durations. However, this shows that the network is well connected.

(3b) Journey Duration Histograms and Longest Path (by Number of Stops)

- Journey Duration Calculations:
  - Total number of journey durations calculated: 77,839
  - Duplicate journeys included/excluded: Included

(Task 3B Histogram)

- Longest Journey:
  - Number of stops: 38
  - 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 → Waterloo → Westminster → St. James's Park → Victoria → Sloane Square → South Kensington → Gloucester Road → Earl's Court → Barons Court → Hammersmith → Acton Town → South Ealing → Northfields → Boston Manor → Osterley → Hounslow East → Hounslow Central → Hounslow West → Hatton Cross → Heathrow Terminals 1, 2, 3 → Heathrow Terminal 5

```python
# Function to get the journey length (for finding max)
def get_journey_length(journey):  1 usage  iTrqzy *
    return journey[0]


# Find the longest journey (in terms of stops) and the stations involved
longest_duration, start_station_idx, end_station_idx = max(journey_durations, key=get_journey_length)

# Reconstruct the path for the longest journey using the predecessors from Dijkstra's algorithm
distances, predecessors = dijkstra(graph, start_station_idx)  # Run Dijkstra again for the longest journey's start
path = []  # List to store the longest journey path
current = end_station_idx  # Start from the end station

# Trace back from the end station to the start using the predecessor list
while current is not None:
    path.append(stations[current])
    current = predecessors[current]

# Reverse the path so it's in the correct order (from start to end)
path.reverse()

# Output the longest journey details
print(f"Longest Journey (in stops): {longest_duration} stops")
print(f"Path: {' → '.join(path)}")

# Find the shortest path from the first station
if __name__ == "__main__":
    first_station = stations[0]  # Take the first station as the starting point
    first_station_idx = station_to_index[first_station]

    # Run Dijkstra's algorithm to find the shortest paths from the first station
    distances, predecessors = dijkstra(graph, first_station_idx)
```

In terms of what we had to change for code, instead of updating the list as we find the new longest path (minutes) we are looking at stops per station. If a station has no stops connected to it, we will skip that station and look at which stations have a greater number of connections.

The longest journey based on the number of stops (47 stops) differs significantly from the journey time histogram, which shows most journeys are clustered around 2-3 minutes. This suggests that while many direct connections are short in duration, the network is vast and interconnected, allowing for extended multi-stop journeys across numerous stations.

- **<u>Comparison:</u>**

The longest journey based on minutes shows as 16 minutes but when looking at stops we can see that the longest is 47 stops. Which suggests that the underground is large and interconnected.

TASK 4 [20 marks]

<u>(4a) Line Section Closure Analysis</u>

<u>Selected Algorithm: Kruskal</u>

We have decided to use the Kruskal algorithm to determine which lines can be closed without effecting the network links. In Kruskal's algorithm, sort all edges of the given graph in increasing order. Then it keeps on adding new edges and nodes in the MST if the newly added edge does not form a cycle. It picks the minimum weighted edge at first and the maximum weighted edge at last. Thus, we can say that it makes a locally optimal choice in each step to find the optimal solution. (geeksforgeeks, 2023)

<u>Library Code Implementation:</u>

```python
def build_graph(data):  # Create a graph where each station is a node and each connection is an edge
    stations = list(set(data['Start Station']).union(set(data['End Station'])))  # Create a unique list of stations

    station_to_index = {}  # Map each station to an index
    for index, station in enumerate(stations):
        station_to_index[station] = index

    graph = AdjacencyListGraph(len(stations), weighted=True, directed=False)  # Initialize the graph
    edges = []  # List to hold edges

    for _, row in data.iterrows():  # Add each connection (edge) to the graph
        start = station_to_index[row['Start Station']]
        end = station_to_index[row['End Station']]
        journey_time = row['Journey Time']

        if graph.has_edge(start, end) == False:  # Add the edge only if it does not already exist
            graph.insert_edge(start, end, journey_time)
            edges.append((start, end, journey_time))

    return graph, edges, stations

1 usage  new *
def find_redundant_connections(graph, edges, stations):  # Identify essential connections using Kruskal's algorithm
    mst = kruskal(graph)  # Generate the Minimum Spanning Tree (MST)

    removable_connections = []  # Initialize a list to hold removable connections

    for start, end, _ in edges:  # Check each edge in the original list of edges
        if not mst.has_edge(start, end):  # If the edge is not in the MST, it can be removed
            removable_connections.append((stations[start], stations[end]))

    return removable_connections, mst
```

```
Task4
  adjacency_list_graph.py
  adjacency_matrix_graph.py
  all_pairs_shortest_paths.py
  disjoint_set_forest.py
  dll_sentinel.py
  floyd_warshall.py
  heap.py
  heap_priority_queue.py
  johnson.py
  key_object.py
  London Underground data.xlsx
  mainA.py
  merge_sort.py
  min_heap_priority_queue.py
  mst.py
  print_all_pairs_shortest_path.py
```

Closed Line Sections:

```
Connections That Can Be Removed:
Oxford Circus - Piccadilly Circus
Piccadilly Circus - Charing Cross
Lambeth North - Elephant & Castle
Grange Hill - Hainault
Stratford - Mile End
Liverpool Street - Bank
Holborn - Tottenham
Oxford Circus - Bond Street
Marble Arch - Lancaster Gate
Edgware Road  - Paddington
High Street Kensington - Gloucester Road
Gloucester Road - South Kensington
South Kensington - Sloane Square
St. James's Park - Westminster
Aldgate - Liverpool Street
Farringdon - King's Cross St. Pancras
King's Cross St. Pancras - Euston Square
Aldgate East - Tower Hill
Earl's Court - High Street Kensington
Ealing Common - Ealing Broadway
Aldgate East - Liverpool Street
Baker Street - Edgware Road
Baker Street - Bond Street
Westminster - Waterloo
Canning Town - West Ham
Moor Park - Harrow-on-the-Hill
Harrow-on-the-Hill - Finchley Road
Harrow-on-the-Hill - Wembley Park
Wembley Park - Finchley Road
```

```
Finchley Road - Baker Street
Tottenham Court Road - Leicester Square
Waterloo - Kennington
Euston - Camden Town
King's Cross St. Pancras - Angel
King's Cross St. Pancras - Russell Square
Piccadilly Circus - Green Park
Earl's Court - Barons Court
Hammersmith - Acton Town
Hammersmith - Turnham Green
Turnham Green - Acton Town
South Harrow - Rayners Lane
Vauxhall - Stockwell
Bank - Waterloo
```

Connectivity Verification:

Using Kruskal algorithm, we ensure that the MST connects all stations with the minimum number of connections and that the stations that are not listed in the MST are seen as redundant.

Analysis:

Having redundant stations closed can have a positive and a negative impact on the network. The positive being that it can reduce journey durations as stations that don't need to be stopped at are no longer within the journey. However, the downside of removing active lines is the inconvenience it may add to journeys.

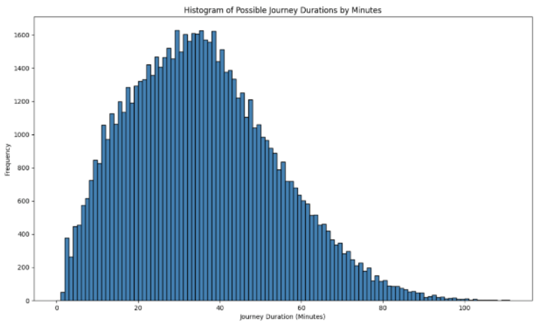
Code Implementation:

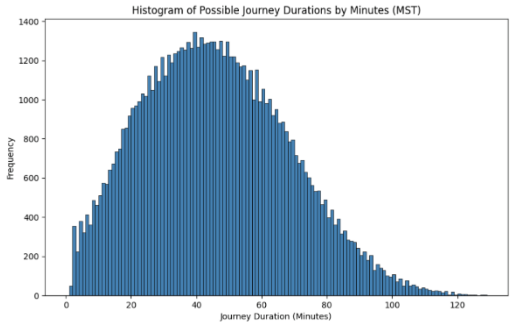
```python
1 usage  iTrqzy *
def build_graph(data):  # Create a graph where each station is a node and each connection is an edge
    stations = list(set(data['Start Station']).union(set(data['End Station'])))  # Create a unique list of stations

    station_to_index = {}  # Map each station to an index
    for index, station in enumerate(stations):
        station_to_index[station] = index

    graph = AdjacencyListGraph(len(stations), weighted=True, directed=False)  # Initialize the graph
    edges = []  # List to hold edges

    for _, row in data.iterrows():  # Add each connection (edge) to the graph
        start = station_to_index[row['Start Station']]
        end = station_to_index[row['End Station']]
        journey_time = row['Journey Time']

        if graph.has_edge(start, end) == False:  # Add the edge only if it does not already exist
            graph.insert_edge(start, end, journey_time)
            edges.append((start, end, journey_time))

    return graph, edges, stations
```

```python
def find_redundant_connections(graph, edges, stations):  # Identify essential connections using Kruskal's algorithm
    mst = kruskal(graph)  # Generate the Minimum Spanning Tree (MST)

    removable_connections = []  # Initialize a list to hold removable connections

    for start, end, _ in edges:  # Check each edge in the original list of edges
        if not mst.has_edge(start, end):  # If the edge is not in the MST, it can be removed
            removable_connections.append((stations[start], stations[end]))

    return removable_connections, mst
```

(4b) Impact Analysis of Line Section Closures

- **Histogram Comparison:**





Analysis:

From looking at both histograms, we can see that the network will all the connections reduces travel times as it becomes greater. However, the reduced network, we can see that there are more longer journey times on average. Showing that the MST makes stations more far apart and doesn't allow for intermediate stations to be taken, leading to a longer journey time all around.

- **Longest Path Comparison:**
  - **Original Network (from 3a):**

    Duration: 111.0 minutes

  - 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 → Chalfont & Latimer → Chesham

  - **Reduced Network:**

    Duration: 121 minutes

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

- **<u>Analysis:</u>**

  When comparing both paths we can see that the reduced networks longest path takes longer. This may be the case as because of fewer stations it would have to re-route some of its paths. Overall, leading to more stops than the original network.

- **<u>Impact Analysis:</u>**

  Focusing on essential station connections has its ups and downs. Firstly, though it has reduced paths the duration for each journey may be longer as seen above. However, from a real-life perspective it may prove to be efficient as it reduces costs and maintenance.

- **<u>Code Implementation:</u>**

```python
def find_longest_journey_mst(mst, stations, station_indices):  # Find the longest journey using the MST
    max_distance = -1  # Initialize to a very low number
    longest_path = None
    source_station = None
    target_station = None

    # Try Dijkstra's from every station and check the longest journey
    for start_idx in range(len(stations)):
        distances, predecessors = dijkstra(mst, start_idx)

        # Check all distances to find the farthest reachable station
        for end_idx, distance in enumerate(distances):
            if distance > max_distance and distance != float('inf'):
                max_distance = distance
                source_station = start_idx
                target_station = end_idx
                longest_path = print_path(predecessors, source_station, target_station, lambda idx: stations[idx])

    return max_distance, longest_path
```

```python
def build_graph(data):  # Build a graph where stations are nodes and connections are edges
    stations = list(set(data['Start Station']).union(set(data['End Station'])))  # Create a unique list of stations
    station_to_index = {station: index for index, station in enumerate(stations)}  # Map each station to an index
    graph = AdjacencyListGraph(len(stations), weighted=True, directed=False)  # Initialize the graph
    edges = []  # List to hold edges

    for _, row in data.iterrows():  # Add each connection (edge) to the graph
        start = station_to_index[row['Start Station']]
        end = station_to_index[row['End Station']]
        journey_time = row['Journey Time']

        if not graph.has_edge(start, end):  # Add the edge only if it does not already exist
            graph.insert_edge(start, end, journey_time)
            edges.append((start, end, journey_time))

    return graph, edges, stations
```

Progress Journal [20 marks]

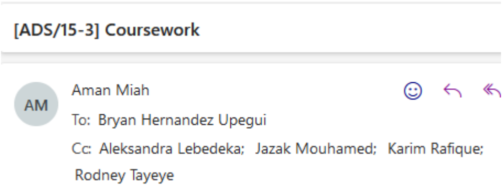
1. Weekly progress log

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**
07/Oct-11/Oct	Emails sent for communication between each group member. Teams' setup by Bryan, and summary of member details setup. Lebedeka, not part of the university anymore. Aman roughly summarised the coursework to each group member to gain an initial understanding. Also, completing Task 1A.
14/Oct - 20/Oct	Jazak contributed to Task 1B along with Aman. Email sent out by team leader for what needs to happen this week as well as what happened last week. Bryan and Karim contributed and completed 2A and 2B supported by Jazak. Aman completed task 3A along with task 3B. Team meeting was completed and attended by everyone.
21/Oct - 27/Oct	Email sent out by team leader for this week and teams meeting was scheduled. Task 4a was completed by Aman contributed by Bryan. Teams meeting was attended by group members. Task 4b was completed by Aman.
28/Oct - 03/Nov	Email sent out by team leader for this week outlining the credits for the task completion along with penultimate teams meeting scheduled.
04/Nov - 10/Nov	Email was sent out by team leader for this week outlining changes that need to be made to the coursework as final touches and clarifying credits for each task. Final teams meeting scheduled and completed. Rodney contributed to changes made for task 3A, 3B and 4B assisted by Aman.

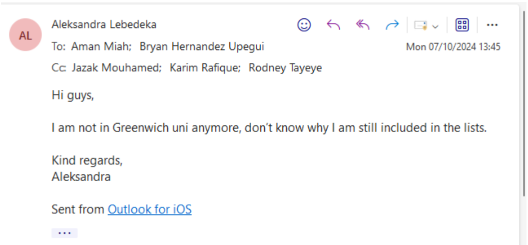
2. Evidence of a weekly email on cumulative credits and a Teams online meeting

- Week 7/Oct - 13:

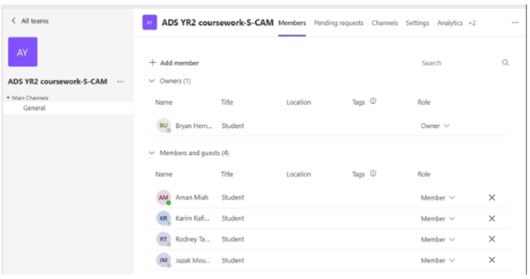
Screenshot of group leaders' email to all group members to establish communication:



Evidence of member, Lebedeka, Aleksandra no longer being part of the university:

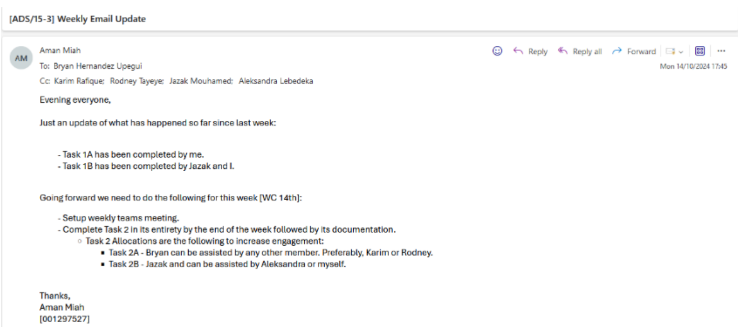


Screenshot group members to establish communication group in teams

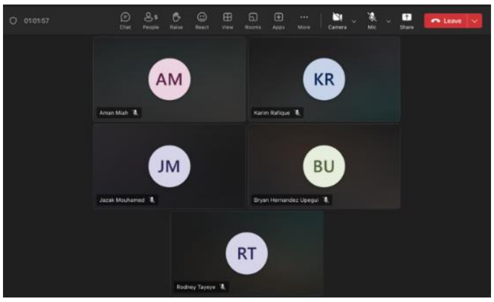


- Week 14/Oct - 20:

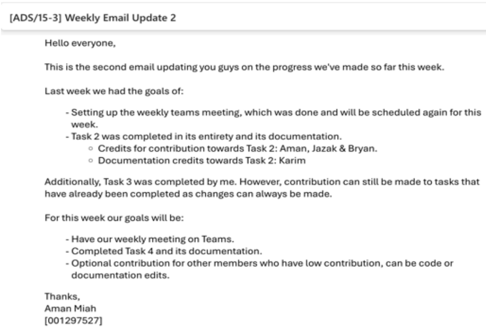
Evidence of team leader sending weekly email.



Evidence of team meeting on teams.



- Week 21/Oct - 27:



[ADS/15-3] Weekly Email Update 2

Hello everyone,

This is the second email updating you guys on the progress we've made so far this week.

Last week we had the goals of:

- Setting up the weekly teams meeting, which was done and will be scheduled again for this week.
- Task 2 was completed in its entirety and its documentation.
  - Credits for contribution towards Task 2: Aman, Jazak & Bryan.
  - Documentation credits towards Task 2: Karim

Additionally, Task 3 was completed by me. However, contribution can still be made to tasks that have already been completed as changes can always be made.

For this week our goals will be:

- Have our weekly meeting on Teams.
- Completed Task 4 and its documentation.
- Optional contribution for other members who have low contribution, can be code or documentation edits.

Thanks,
Aman Miah
[001297527]

Evidence of team meeting on teams.

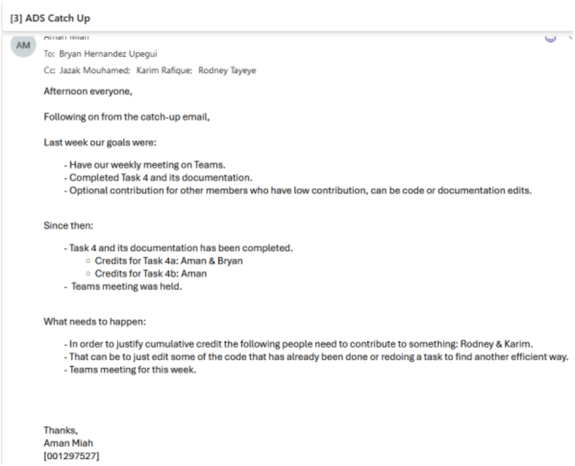
25 October 2024 14:48 - 16:04 Download

5 Attended | 14:48 - 16:04 Start and end time | 1h 16m 35s Meeting duration | 50m 6s Average attendance time

Participants

Name	First join	Last leave	In-meeting duration	Role
Aman Miah am7414s@greenwich.ac.uk	15:01	16:04	1h 2m 23s	Organiser
Karim Rafique ar1623n@greenwich.ac.uk	14:48	15:56	1h 8m 45s	Presenter
Jazak Mouhamed jm4379q@greenwich.ac.uk	15:01	15:46	44m 48s	Presenter
Rodney Tayeye rt6394r@greenwich.ac.uk	15:10	16:04	54m 32s	Presenter
Bryan Hernandez Upegui bh5510f@greenwich.ac.uk	15:35	15:55	20m 2s	Presenter

- Week 28/Nov - 1 :



[3] ADS Catch Up

Aman Miah
To: Bryan Hernandez Upegui
Cc: Jazak Mouhamed; Karim Rafique; Rodney Tayeye

Afternoon everyone,

Following on from the catch-up email,

Last week our goals were:

- Have our weekly meeting on Teams.
- Completed Task 4 and its documentation.
- Optional contribution for other members who have low contribution, can be code or documentation edits.

Since then:

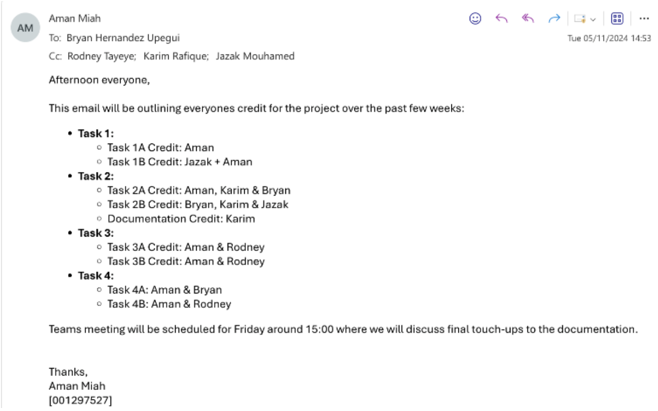
- Task 4 and its documentation has been completed.
  - Credits for Task 4a: Aman & Bryan
  - Credits for Task 4b: Aman
- Teams meeting was held.

What needs to happen:

- In order to justify cumulative credit the following people need to contribute to something: Rodney & Karim.
- That can be to just edit some of the code that has already been done or redoing a task to find another efficient way.
- Teams meeting for this week.

Thanks,
Aman Miah
[001297527]

- Week 04/Nov - 10 :


Aman Miah
To: Bryan Hernandez Upegui
Cc: Rodney Tayeye; Karim Rafique; Jazak Mouhamed
Tue 05/11/2024 14:53

Afternoon everyone,

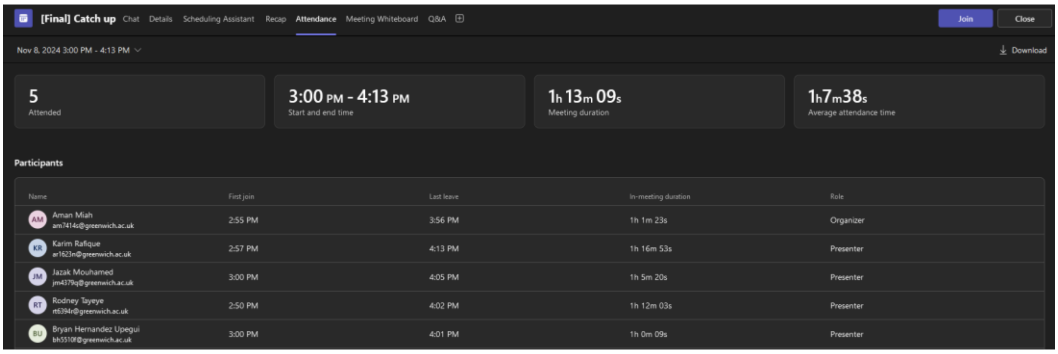
This email will be outlining everyones credit for the project over the past few weeks:

- **Task 1:**
  - Task 1A Credit: Aman
  - Task 1B Credit: Jazak + Aman
- **Task 2:**
  - Task 2A Credit: Aman, Karim & Bryan
  - Task 2B Credit: Bryan, Karim & Jazak
  - Documentation Credit: Karim
- **Task 3:**
  - Task 3A Credit: Aman & Rodney
  - Task 3B Credit: Aman & Rodney
- **Task 4:**
  - Task 4A: Aman & Bryan
  - Task 4B: Aman & Rodney

Teams meeting will be scheduled for Friday around 15:00 where we will discuss final touch-ups to the documentation.

Thanks,
Aman Miah
[001297527]

Evidence of teams meeting:



Name	First join	Last leave	In-meeting duration	Role
Aman Miah am7414s@greenwich.ac.uk	2:55 PM	3:56 PM	1h 1m 23s	Organizer
Karim Rafique ar1623n@greenwich.ac.uk	2:57 PM	4:13 PM	1h 16m 53s	Presenter
Jazak Mouhamed jm4379q@greenwich.ac.uk	3:00 PM	4:05 PM	1h 5m 20s	Presenter
Rodney Tayeye rt6394r@greenwich.ac.uk	2:50 PM	4:02 PM	1h 12m 03s	Presenter
Bryan Hernandez Upegui bh5510f@greenwich.ac.uk	3:00 PM	4:01 PM	1h 0m 09s	Presenter

Bibliography

Babitz, K., 2023. *matplotlib-tutorial-python.* [Online]
Available at: https://www.datacamp.com/tutorial/matplotlib-tutorial-python

Cormen, T. H., Leiserson, C. E., Rivest, R. L. & C. S., 2022. Introduction to Algorithms. In: s.l.:s.n., p. 662.

geeksforgeeks, 2023. *kruskal-mst.* [Online]
Available at: https://www.geeksforgeeks.org/kruskals-minimum-spanning-tree-algorithm-greedy-algo-2/

pandas, n.d. *pydata.* [Online]
Available at: https://pandas.pydata.org/docs/reference/api/pandas.read_excel.html

Singh, A., 2024. *Floyd-Warshall Algorithm.* [Online]
Available at: https://medium.com/@singhatul1155/depths-of-the-floyd-warshall-algorithm-most-effecient-for-shortest-path-ac790d888c07