**TASK 1**

**(1a)**

**Manual versus Code-Based Execution of the Algorithm**

**Selected Data Structure and Algorithm:**

* Algorithm: Dijkstra’s Algorithm (W3Schools, n.d.)
* Data Structure: Graph

**Justification:**

Dijkstra's Algorithm and the graph data format effectively address pathfinding issues in networks such as a tube system, they are perfect for this coursework:

* Optimal Pathfinding: Dijkstra's Algorithm is ideal for cutting down on travel time or stops between stations because it is made to discover the shortest path in weighted graphs.
* Natural Network Representation: The tube network is intuitively represented by a graph structure, particularly an adjacency list, in which stations are nodes and routes are edges. This configuration scales effectively as the network expands and allows for fast lookups of connected stations.

Simple Dataset:

Small artificial tube network with stations and journey between them in minutes.

5 stations: A, B, C, D, E

Distance between them:

A to B: 8 minutes, B to C: 3 minutes, A to D: 7 minutes, D to E: 1 minute, C to E: 5 minutes

**Manual Algorithm Execution:**

A diagram of a diagram

Description automatically generated

Shortest Path from B to D is B to C, C to E, E to D with a total journey time of 9 minutes.

**Code Implementation:**

The part of your Python code that implements the dataset

This section outlines the graph (tube network) applying an adjacency list form. The graph shows the stations and the corresponding travel times between them.

# Step 1: The first step is to create the graph (tube network) using AdjencyListGraph class  
vertices = ['A', 'B', 'C', 'D', 'E'] # Available stations  
edges = [  
 ('A', 'B', 8), # A to B- 8 minutes  
 ('A', 'D', 7), # A to D- 7 minutes  
 ('B', 'C', 3), # B to C- 3 minutes  
 ('C', 'E', 5), # C to E- 5 minutes  
 ('D', 'E', 1) # D to E- 1 minute  
]  
  
# Set the graph with 5 vertices, directed=False, weighted=True  
graph = AdjacencyListGraph(len(vertices), directed=False, weighted=True)  
  
# Put the edges into the graph  
for edge in edges:  
 graph.insert\_edge(vertices.index(edge[0]), vertices.index(edge[1]), edge[2])

* Vertices: Defines the stations
* Edges: Defines the journeys between stations along with travel times (weights)
* AdjacencyListGraph: Initializes the graph and inserts each journey as an undirected, weighted edge. (Geeks, n.d.)

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

This fragment shows the code calling the Dijkstra function (imported from the dijkstra.py library file), which computes the shortest path from the source station.

# Step 4: Determine the shortest route from source to destination  
source\_vertex = vertices.index(source\_station) # Obtain the index of the source station  
d, pi = dijkstra(graph, source\_vertex) # Call the imported Dijkstra function.

* **Dijkstra** (graph, source\_vertex): Calls the **Dijkstra** function from the imported library, passing the graph and source station index
* D: It stores the shortest distances from the source station to all other stations
* Pi: Stores the predecessors to help reconstruct paths if needed

The output showing the shortest route and journey duration

This section of the code calculates the shortest route from station B to station D with the Dijkstra algorithm, eventually displaying the travel duration.

# Step 5: Rebuild the shortest path  
target\_vertex = vertices.index(destination\_station) # Get the index of the destination station  
path = []  
current\_vertex = target\_vertex  
  
while current\_vertex is not None:  
 path.insert(0, vertices[current\_vertex]) # Insert each station at the beginning of the path list  
 current\_vertex = pi[current\_vertex] # Go to the predecessor  
  
# Step 6: Show the shortest route and journey duration  
print(f"Shortest path from {source\_station} to {destination\_station}: {' -> '.join(path)}")  
print(f"Journey duration: {d[target\_vertex]} minutes")

**Dijkstra’s Execution**: Once the user inputs valid stations, the source station is found using vertices. Index (), and Dijkstra’s algorithm is run.

**Output**: The program prints the shortest journey time (in minutes) from the source station to the destination station by looking up the distance in the d dictionary.

Code Output:

**Shortest Path from B to D: 9 minutes**

The output shows that the shortest path from station **B** to station **D** is **9 minutes**, via the path: B to C, C to E and E to D.

B to C: 3 minutes, C to E: 5 minutes, E to D: 1 minute

Total= 9 minutes

Comparison:

1. Manual Calculation:

Shortest path from B to D=9 minutes, via B to C, C to E, E to D

1. Code-Generated Result:

Shortest path from B to D=9 minutes, Via Dijkstra’s algorithm.

The results obtained through **manual execution** and **code generation** are identical, validating the accurate implementation of Dijkstra’s algorithm, with both producing the optimum route of **9 minutes.**

**(1b)**

**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]

This dataset represents a graph with n stations and random journey durations (edge weights) between pairs of adjacent stations.

def generate\_random\_tube\_network(n, edge\_probability=0.1, min\_weight=1, max\_weight=15):  
 *"""  
 Generate a random tube network with n stations and random journey durations.  
  
 Arguments:  
 n -- number of stations (vertices)  
 edge\_probability -- probability that an edge (journey) exists between stations  
 min\_weight, max\_weight -- range for random journey durations (weights)  
  
 Returns:  
 graph -- adjacency list representation of the tube network  
 """* graph = AdjacencyListGraph(n, directed=False, weighted=True)  
 for u in range(n):  
 for v in range(u + 1, n):  
 if random() < edge\_probability: # Create edge with probability  
 weight = randint(min\_weight, max\_weight)  
 graph.insert\_edge(u, v, weight) # Add edge with weight  
 return graph

**Execution Time Measurement:**

def measure\_execution\_time(graph, source, repetitions=5):  
 *"""  
 Measure the execution time for Dijkstra's algorithm on the tube network.  
 Arguments:  
 graph -- the tube network (AdjacencyListGraph)  
 source -- the source station index  
 repetitions -- number of times to repeat for averaging  
 Returns:  
 average\_time -- average execution time in milliseconds  
 """* total\_time = 0  
 for \_ in range(repetitions):  
 start\_time = time.time()  
 dijkstra(graph, source) # Use Dijkstra's algorithm  
 end\_time = time.time()  
 total\_time += (end\_time - start\_time) \* 1000 # Change the time to milliseconds  
 return total\_time / repetitions

* This function calculates the execution time of Dijkstra’s algorithm on the generated network
* A source station (source\_vertex) is selected ( it is usually the first station, index 0) for running Dijkstra’s algorithm
* The execution time is the average of several repetitions to give trusted timing

Steps:

* **Randomly selecting a pair of stations:** In each run of Dijkstra’s Algorithm, the source station is fixed at 0. It could be further enhanced by randomly choosing the source each time
* **Computing the Journey:** The Dijkstra’s function computes journey durations from the source to all reachable stations.
* **Average Execution Time:** When this procedure is repeated many times, it provides an average execution time for more stable results.

**Time Complexity Graph:**

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

# Step 4: Plot results for empirical time complexity  
plt.figure(figsize=(10, 6))  
plt.plot(network\_sizes, execution\_times, marker='o', label='Execution Time')  
plt.xlabel('Network Size (Number of Stations)')  
plt.ylabel('Average Execution Time (ms)')  
plt.title('Average Execution Time vs Network Size')  
plt.grid(True)  
plt.legend()  
plt.show()

* This code plots the average execution time against the network size (number of stations) to show the time complexity.
* For each network size n, the number of line sections (edges) is controlled by the edge\_probability in generate\_random\_tube\_network. This determines the expected density of edges.
* The Plotted graph helps visualize how the execution time of Dijkstra’s algorithm grows as the network size increases, aligning with the theoretical time complexity for Dijkstra’s algorithm in sparse or dense graphs.

State the tool used for graph generation

* Graph Library: AdjacencyListGraph from “Introduction to Algorithms”
* Plotting: matplotlib for visualization

**Analysis: Theoretical Time Complexity of Dijkstra’s Algorithm**

The theoretical time complexity of Dijkstra's algorithm is dependent upon the data structure employed for the priority queue implementation. In this instance, given that we are utilising a min-heap priority queue (MinHeapPriorityQueue), the time complexity is:

**Time Complexity**: O ((V + E) \* log V) (Earth, n.d.), where:

* **V**: Number of vertices (stations).
* **E**: Number of edges (line sections between stations).

Comparison:

**Alignments**

* As the network size expands, both V (vertices) and E (edges) increase. The time complexity is O ((V + E) \* log V), indicating that the execution time is anticipated to increase approximately in proportion to V \* log V. If your graph displays a non-linear growth in execution time (first slower and then accelerating more rapidly as V increases), this corresponds with the theoretical time complexity.

**Discrepancies**

* **Edge Probability and Density of graph:** If the graph possesses a relatively high edge probability, indicating an increased number of connections between stations, this will increase the total number of edges, hence expanding E. The complexity of time is dependent upon both V and E; thus, an increase in edges leads to extended execution durations. Elevated execution times in your empirical graph may be attributed to an increased density of edges inside the network.

**TASK 2**

**(2a)**

**Manual versus Code-Based Execution of the Algorithm**

Selected Data Structure and Algorithm:

* Algorithm: Dijkstra’s Algorithm
* Data Structure: Graph

Justification:

Dijkstra's Algorithm and the graph data format effectively address pathfinding issues in networks such as a tube system, they are perfect for this coursework:

* Optimal Pathfinding: Dijkstra's Algorithm is ideal for cutting down on travel time or stops between stations because it is made to discover the shortest path in weighted graphs.

**Simple Dataset:**

Small artificial tube network with stations and journey between them in minutes.

5 stations: A, B, C, D, E

Distance between them:A to B: 1 stop, B to C: 1 stop, A to D: 1 stop, D to E: 1 stop, C to E: 1 stop

**Manual Algorithm Execution: [**

A diagram of a manual execution

Description automatically generated

Shortest Path from B to D is B to C, C to E, E to D with 2 stops in total

**Code implementation with key difference:**

Journey duration is measured by the number of stops rather than minutes

Graph Initialization for stop-based calculation:

* Instead of using the duration of the journey in minutes as weights, the graph is set 1 so that it calculates the paths based on the number of stops.

# this graph is for stop based calculation  
graph\_stops = AdjacencyListGraph(len(vertices), directed=False, weighted=True)  
  
# inserting the edges with stop count (weight as 1) into the stop-based graph  
for edge in edges\_time:  
 graph\_stops.insert\_edge(vertices.index(edge[0]), vertices.index(edge[1]), 1)

Dijkstra’s Algorithm applied for stops:

* The Dijkstra’s function is used to the stop constructed graph to calculate the shortest route in terms of the number of stops.

# calculating the shortest path based on stops  
d\_stops, pi\_stops = dijkstra(graph\_stops, source\_vertex)

Path reconstruction and output for Stops:

* The reconstruct\_path function is used to rebuild the route, and the output is shown in relation with the number of stops.

# Shortest path by stops  
if d\_stops[target\_vertex] != float('inf'):  
 path\_stops = reconstruct\_path(pi\_stops, source\_vertex, target\_vertex)  
 path\_stops\_named = [vertices[v] for v in path\_stops]  
 print(  
 f"Shortest path from {source\_station} to {destination\_station} (by stops): {d\_stops[target\_vertex]} stops. Path: {' -> '.join(path\_stops\_named)}"  
  
 )  
  
else:  
  
 print(f"There is no path from {source\_station} to {destination\_station} (by stops).")

Summary of the differences:

* Graph weights: The graph is changed to use 1 as weights for each edge, representing one stop.
* **Shortest Path calculation:** The shortest path is calculated in terms of the number of stops rather than the time in minutes.
* **Output:** The output specifies the number of stops and the corresponding path instead of journey duration in minutes.

**(2b)**

**Difference in Empirical Measurement of Time Complexity Between Task 1b and 2b**

The key difference between both the tasks is what the **graph represents** and how the **Dijkstra’s algorithm** operates on it, which directly influences the nature of the **time complexity analysis.**

**Representation of the Graph**

*Returns:  
 graph -- the generated tube network as an adjacency list graph  
 """* graph = AdjacencyListGraph(n, directed=False, weighted=True)  
 for u in range(n):  
 for v in range(u + 1, n):  
 if random.random() < edge\_probability:  
 weight = random.randint(min\_weight, max\_weight)  
 graph.insert\_edge(u, v, weight)  
 return graph

**In Task 2b:**

* The graph edges have **constant weights** (all weights=1) to **represent the stops.**
* Dijkstra’s algorithm reduces the **number of edges** (stops), not their total weights
* The computational cost depends solely on the graph topology (connections between stations), with no variations in edge weights.

def generate\_random\_tube\_network(n, edge\_probability=0.1):  
 *"""  
 Generate an artificial tube network with n stations.  
 Each edge weight is 1, representing one stop.  
  
 Arguments:  
 n -- number of stations (vertices)  
 edge\_probability -- probability of an edge (journey) existing between two stations  
  
 Returns:  
 graph -- the generated tube network as an adjacency list graph  
 """* graph = AdjacencyListGraph(n, directed=False, weighted=True)  
 for u in range(n):  
 for v in range(u + 1, n):  
 if random.random() < edge\_probability:  
 weight = 1 # Weight is set o 1 to show one stop  
 graph.insert\_edge(u, v, weight)  
 return graph.

**In Task 2b:**

* The time complexity is measured for **stop-based calculations** using a graph with uniform edge weights.
* Empirical time complexity reflects a simpler case where all edges are treated equally

**Code Difference in Empirical Measurement**

**Graph Generation:**

**Task 2b:**

* Graph edges have constant weights (weight=1), representing one stop:

weight = 1 # Weight is set o 1 to show one stop  
graph.insert\_edge(u, v, weight)

**Dijkstra’s Algorithm Objective**

**In Task 2b:**

* Objective: Find the shortest path based on the **number of stops** (sum of uniform edge weights).
* The algorithm only counts the number of edges traversed, making priority queue operations simpler and faster.

start\_time = time.time()  
dijkstra(graph, source) # Apply Dijkstra's algorithm  
end\_time = time.time()

**Execution Time Measurement**

Task (2b): Impact of variable weights

Variable edge weights need more comparisons in the algorithm’s priority que operations, reducing execution time.

def measure\_execution\_time(graph, source, repetitions=5):  
 total\_time = 0  
 for \_ in range(repetitions):  
 start\_time = time.time()  
 dijkstra(graph, source) # Apply Dijkstra's algorithm  
 end\_time = time.time()  
 total\_time += (end\_time - start\_time) \* 1000 # Convert to milliseconds  
 return total\_time / repetitions

**Network Sizes**

Task 2(b):

* The network size ranges from 1100 to 2000
* Larger network size reflects the simplified computation due to uniform weights.

network\_sizes = [1100, 1200, 1300, 1400, 1500, 1600, 1700, 1800, 1900, 2000] # As per the specification

**Time Complexity Graph**

Task (2b):

The graph shows a linear graph, as constant weights simplify the algorithm.

plt.plot(network\_sizes, execution\_times, marker='o', label='Execution Time')

plt.title('Average Execution Time vs Network Size (Number of Stops)')

Graph for 2(b) (PyPi, n.d.):

A graph with a line going up

Description automatically generated

**TASK 3**

**(3a).**

**Journey Duration Histograms and Longest Path (in Minutes)**

**Data Import Method:** Briefly describe how you imported the 'London Underground Data.xlsx' into your Python code.

* The data import is from an Excel file named “London Underground Data.xlsx” using the pandas library (W3Schools, n.d.).
* The Columns are named [“Line”, “Station1”, “Station2”, “Duration”] to represent line name, two of the connected stations, and journey duration in minutes.

data = pd.read\_excel("London Underground Data.xlsx", header=None)  
data.columns = ["Line", "Station1", "Station2", "Duration"] # Coloumn name based on the file format

**Journey Duration Calculations:**

Total number of journey durations calculated:

The code calculates journey durations for all different station pairs. Using a nested loop over stations and excluding duplicates. (A->B vs B<-A). It stores the durations in a duration list.

for source in range(n): # For every station  
 distances, predecessors = dijkstra(graph, source) # Use Dijkstra’s algorithm  
 for target in range(source + 1, n): # Only use unique pairs (A->B, not B->A)  
 if distances[target] != float('inf'): # Remove unapproachable pairs  
 durations.append(distances[target]) # stock the journey duration  
 paths[(source, target)] = (distances[target], predecessors) # Store the path information

Duplicate journeys included/excluded:

The duplicate journeys have been excluded. Only the unique pairs are taken into consideration.

Histogram:

Method used to plot the histogram:

The matplotlib library is used plot the histogram. The histogram visualizes the distribution of journey durations in minutes across the entire network.

plt.hist(durations, bins=20, edgecolor='black')  
plt.xlabel("Journey Duration (minutes)")  
plt.ylabel("Frequency")  
plt.title("Distribution of Journey Durations in the London Underground Network")  
plt.show()

Interpretation:

The histogram will show the frequency of journey durations across different intervals. This helps identify common journey times and dissident

Longest Journey:

Duration: The maximum duration found in the duration list

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

**Code:**

longest\_duration = max(durations)  
longest\_pair = max(paths, key=lambda pair: paths[pair][0]) # Get the pair of stations with the longest duration

longest\_path\_indices = reconstruct\_path(predecessors, longest\_pair[0], longest\_pair[1])  
longest\_path\_stations = [stations[i] for i in longest\_path\_indices]

print(f"Longest journey duration: {longest\_duration} minutes")  
print("Path for the longest journey (in order):")  
print(" -> ".join(longest\_path\_stations))

**Histogram** (Schafer, n.d.)**:**

**A graph of a number of blue bars

Description automatically generated**

Code Implementation:

* Data Import:

data = pd.read\_excel("London Underground Data.xlsx", header=None)  
data.columns = ["Line", "Station1", "Station2", "Duration"] # Coloumn name based on the file format

* Graph Initialization and Edge Insertion:

for \_, row in data.iterrows():  
 source = station\_to\_index[row['Station1']]  
 target = station\_to\_index[row['Station2']]  
 duration = row['Duration']  
  
 # Only add the edge if has not been added before  
 if (source, target) not in added\_edges and (target, source) not in added\_edges:  
 graph.insert\_edge(source, target, duration)  
 added\_edges.add((source, target)) # Mark this edge as added

* Journey Duration Calculations:

for source in range(n): # For every station  
 distances, predecessors = dijkstra(graph, source) # Use Dijkstra’s algorithm  
 for target in range(source + 1, n): # Only use unique pairs (A->B, not B->A)  
 if distances[target] != float('inf'): # Leave unreachable pairs  
 durations.append(distances[target]) # Stock the journey duration  
 paths[(source, target)] = (distances[target], predecessors) # Store path info

* Histogram Plotting

plt.hist(durations, bins=20, edgecolor='black')

* Longest Path Duration

longest\_duration = max(durations)  
longest\_pair = max(paths, key=lambda pair: paths[pair][0]) # Get the pair with the longest duration  
longest\_path\_duration, predecessors = paths[longest\_pair]

**Analysis:**

By emphasising frequent and uncommon route times, the histogram offers insights into typical journey durations within the network. The longest trip time indicates possible areas for route optimisation or network connectivity enhancement.

**(3b)**

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

Journey Duration Calculations:

* Total number of journey durations calculated: 2500 journeys were calculated
* Duplicate journeys included/excluded: Duplicate Journeys were excluded

Histogram:

Method used to plot the histogram: To plot the histogram, the matplotlib library was used

Image of the histogram:

A graph of a number of steps

Description automatically generated

Longest Journey:

Number of stops: 38

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 -> Hammersmith -> Barons Court -> Earl's Court -> Gloucester Road -> South Kensington -> Sloane Square -> Victoria -> Green Park -> Westminster -> Waterloo -> Bank -> Liverpool Street -> Bethnal Green -> Mile End -> Stratford -> West Ham -> Plaistow -> Upton Park -> East Ham -> Barking -> Upney -> Becontree -> Dagenham Heathway -> Dagenham East -> Elm Park -> Hornchurch -> Upminster Bridge -> Upminster

Code Implementation:

if (source, target) not in added\_edges and (target, source) not in added\_edges:  
 graph.insert\_edge(source, target, duration)  
 added\_edges.add((source, target)) # Mark this edge as added

The code 3a uses duration as an edge weight as only time is being measured between stops.

if (source, target) not in added\_edges and (target, source) not in added\_edges:  
 graph\_stops.insert\_edge(source, target, 1)  
 added\_edges.add((source, target))

On the other hand, 3b uses 1 as a fixed weight of 1 representing the number of stops.

for target in range(source + 1, n): # Only use unique pairs (A->B, not B->A)  
 if distances[target] != float('inf'): # Exclude unapproachable pairs  
 durations.append(distances[target]) # Stock the journey duration  
 paths[(source, target)] = (distances[target], predecessors) # Store path information

In 3a durations are stored as time (in minutes)

for target in range(source + 1, n): # Only consider unique pairs (A->B, not B->A)  
 if distances[target] != float('inf'): # Exclude unreachable pairs  
 durations\_stops.append(distances[target]) # Store the journey duration  
 paths\_stops[(source, target)] = (distances[target], predecessors)

For 3b, the durations are stored in stops

plt.xlabel("Journey Duration (minutes)")

X-axis for histogram for 3a is assigned for duration in minutes

plt.xlabel("Journey Duration (Number of Stops)")

X-axis for histogram for 3b is assigned for duration in stops.

print (f"Longest journey duration: {longest\_duration} minutes")

For 3a the console outputs the longest duration in minutes

print(f"Longest journey duration (by stops): {longest\_duration\_stops} stops")

For 3b console outputs the longest duration in stops

data = data.dropna(subset=["Station1", "Station2", "Duration"])  
data['Station1'] = data['Station1'].astype(str).str.strip()  
data['Station2'] = data['Station2'].astype(str).str.strip()

Empty spaces were present in the data set, so we used this function to make sure the interpreter did not acknowledge them to minimise errors.

Comparison with 3a:

The longest journey in 3a (measured in minutes) takes 111.0 minutes, whereas the longest journey in 3b (measured in stops) takes 38 minutes. For the longest journey, the routes are very different; 3a covers stations such as Chesham and Upminster, whereas 3b shows a route from Heathrow Terminal 5 to Upminster. This variation illustrates how network optimisation varies according on the metric (time vs. stops) and shows that the longest route by stops does not always equal the longest journey by time. This demonstrates the London Underground network's varied connection, where trip duration and stop count are not always exactly proportional.

**TASK 4**

**(4a)**

**Line Section Closure Analysis**

Selected Algorithm:

Kruskal's Minimum Spanning Tree (MST) algorithm (Geeks, n.d.) is used to identify redundant line sections in the London Underground network. These are sections that are not part of the Minimum Spanning Tree, meaning they can be removed without disconnecting the network.

Library Code Implementation:

This section contains the necessary libraries and the setup for custom modules.

import pandas as pd  
from adjacency\_list\_graph import AdjacencyListGraph  
from mst import kruskal # Using the Kruskal's MST algorithm from the library ("Introduction to Algorithms" 4th edition)

Creating the Graph:

This part of the code converts the journey data into a graph representation, where stations are the vertices, and journey durations are the edge weights.

def create\_graph(data):  
 *"""Initialize the graph with vertices and edges based on the data."""* # Get unique stations from both Start\_Station and End\_Station columns  
 unique\_stations = pd.concat([data['Start\_Station'], data['End\_Station']]).unique()  
 graph = AdjacencyListGraph(len(unique\_stations), directed=False, weighted=True)  
  
 # Map each station to a unique index  
 station\_to\_index = {station: idx for idx, station in enumerate(unique\_stations)}  
  
 # Add edges to the graph  
 for \_, row in data.iterrows():  
 u = station\_to\_index[row['Start\_Station']]  
 v = station\_to\_index[row['End\_Station']]  
 weight = row['Duration']  
 if not graph.has\_edge(u, v): # Avoid duplicate edges  
 graph.insert\_edge(u, v, weight)  
  
 return graph, station\_to\_index, unique\_stations

Finding redundant edges using Kruskal’s algorithms

def find\_redundant\_edges(graph):  
 *"""Identify edges that are not part of the MST."""* mst\_graph = kruskal(graph) # Call the Kruskal's algorithm from the imported module  
 original\_edges = set(graph.get\_edge\_list())  
 mst\_edges = set(mst\_graph.get\_edge\_list())  
 redundant\_edges = original\_edges - mst\_edges  
 return redundant\_edges

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

Connectivity Verification:

* The MST made sure that the connectivity is maintained by including the minimum number of edges needed to maintain a connected graph
* After finding out the MST using kruskal’s algorithm, only edges not in the MST are marked for closure. The makes sure that every station remains available from an alternate route.

Analysis:

This analysis helps to optimize the London Underground network by identifying sections that are not necessary for connectivity. By removing redundant sections, the network could be streamlined, potentially reducing operational costs or identifying areas for improvements. The goal is to ensure that the transportation system remains connected while minimizing unnecessary connections.

Code Implementation:

Relevant Code demonstrating closure Determination:

* Redundant Edge Identification:

original\_edges = set(graph.get\_edge\_list())  
mst\_edges = set(mst\_graph.get\_edge\_list())  
redundant\_edges = original\_edges - mst\_edges

* Output for closed line sections:

closed\_sections = list\_closed\_sections(redundant\_edges, station\_to\_index, vertices)  
print("Closed line sections:")  
for section in closed\_sections:  
 print(section)

**(4b)**

**Impact Analysis of Line Section Closures**

Histogram Comparison:

The histogram shows the overall journey time distributions for all the station pairs within the TFL London Underground Network. After the removal of the redundant edges using the Kruskal’s MST, the histogram shows a up to date journey duration for the reduced network. Overall, the histogram proves a movement in journey durations between the original and the reduced network. The original network was quick and efficient which was unlike the widespread increase in duration times for the reduced network. Line closures caused longer average journey times and slowed down the network drastically.

Longest Path Comparison:

Original Network (from 3a):

Duration: 111 minutes

Path: Chesham -> Chalfont & Latimer -> Chorleywood -> Rickmansworth -> Moor Park -> Harrow-on-the-Hill -> Finchley 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 -> Upminster

Reduced Network:

Duration: 121 minutes

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

Analysis: The original network times for the journey duration between the given stations were 14 minutes, whereas when there was a reduced network, the time drastically increased to 32 minutes, this was an increase of 16 minutes proving line closures paths caused major delays for travellers who experienced a major increase in journey durations

Impact Analysis: The closure for the lines had a major impact on the network. The total distribution of the journey duration moved with a major increase in longer trip times. By removing the redundant edges it provided longer path durations, proving a decrease in connectivity. The closed edges represent the journey routes that used to be vital to provide an efficient and quicker journey travel within the network, but in terms of MST were redundant. This overall proves that a slower reduced network can be optimal for cost bases for TFL but is causes major disruptions and a poor service for travellers.

Code Implementation:

Kruskal’s Algorithm is used for the identification and closure of redundant edges, this allows the graph to remain overall connected without having an increase in weight totals.

mst\_graph = kruskal(graph) # Call the Kruskal's algorithm from the imported module

redundant\_edges = original\_edges - mst\_edges

provided graph updates, by removing redundant edges from the original graph

for u, v in redundant\_edges:  
 graph.delete\_edge(u, v)

Histogram generation, to compare the distribution of journey times 2 histograms were plotted to visually compare to the original and then reduced network times.

plot\_histogram(durations\_original, "Original Network: Journey Durations")  
plot\_histogram(durations\_reduced, "Reduced Network: Journey Durations")

The path to reconstruct for the longest journey duration, I used the reconstruct\_path again to track the station sequence for the longest duration times in both networks for a very precise comparison

original\_longest\_path = [unique\_stations[i] for i in original\_longest\_path\_indices]

reduced\_longest\_path = [unique\_stations[i] for i in reduced\_longest\_path\_indices]

# Works Cited

Earth, H., n.d. *Hacker Earth Shortest Path Algorithms.* [Online]   
Available at: https://www.hackerearth.com/practice/algorithms/graphs/shortest-path-algorithms/tutorial/#:~:text=Time%20Complexity%20of%20Dijkstra's%20Algorithm,E%20l%20o%20g%20V%20)%20.

Geeks, G. f., n.d. *Geeks For Geeks (Graph and its representation).* [Online]   
Available at: https://www.geeksforgeeks.org/graph-and-its-representations/

Geeks, G. f., n.d. *Geeks for Geeks Kruskal's Algorithm and MST.* [Online]   
Available at: https://www.geeksforgeeks.org/kruskals-minimum-spanning-tree-algorithm-greedy-algo-2/

PyPi, n.d. *Pypi Matplotlib.* [Online]   
Available at: https://pypi.org/project/matplotlib/

Schafer, C., n.d. *Matplot Lib Tutorials (Histograms).* [Online]   
Available at: https://www.youtube.com/watch?v=XDv6T4a0RNc

W3Schools, n.d. *W3 Schools DSA Dijkstra's Algorithm.* [Online]   
Available at: https://www.w3schools.com/dsa/dsa\_algo\_graphs\_dijkstra.php

W3Schools, n.d. *W3schools DSA Dijkstra's Algorithm.* [Online]   
Available at: https://www.w3schools.com/dsa/dsa\_algo\_graphs\_dijkstra.php

W3Schools, n.d. *W3Schools Pandas Tutorials.* [Online]   
Available at: https://www.w3schools.com/python/pandas/default.asp