UNIVERSITY OF GREENWICH

Report

COMP1828

Advance Algorithms and Data Structure

Executive Summary

This report summarizes our team's collaborative efforts in designing, developing, and testing solutions for the London Underground system. We were tasked with creating a software model for the London Underground tube system's route planner, with a focus on providing efficient travel information for the public. Our work included tasks related to journey planning and duration, station count evaluation, algorithm comparison and analysis of potential tube line closures. Throughout our project, efficient teamwork, communication, and resource management were essential.

Table of Contents

Introduction 5

Task 1: Journey Itinerary and Duration 6

1a. Route planer and duration in minutes 6

Graph creation 6

1. *Algorithm and Data Structure Selection*: 8

2. Test Data Selection 11

3. Result and Functionality Presentation 11

4. Final Remarks and Limitations 12

1b. Histogram of journey times in minutes 13

Task 2: Station Count Calculation 15

2a. The count of stations between the starting point and the destination 15

1. *Algorithm and Data Structure Selection*: 15

2. Test Data Selection 16

3. Result and Functionality Presentation 17

2b. Histogram of numbers of stops between each station pair. 17

Task 3: Alternative Algorithm Comparison 19

3a. Algorithm Selection 19

1. Algorithm and Data Structure Selection: 19

2. Test Data Selection 22

3. Result and Functionality Presentation 22

4. Final Remarks and Limitations 22

3b. Histogram of numbers of stops between each station pair. 23

Task 4: Tube Line Closure Analysis 24

4a. Algorithm Selection 24

1. Algorithm and Data Structures selection: 24

2. Test data selection: 28

3. Result and Functionality Presentation: 29

4. Final Remarks and Limitations 31

4b. 31

Impacts of the closure of adjacent stations: 31

Testing for correctness and efficiency 33

Test cases: 34

Conclusion 37

Teamwork and Communication 37
  Reflection and critical discussion 40
References 41
Appendices 42
  Additional Documentation 42

Introduction

The London Underground is the core of London's transportation system and, to meet public needs, our team embarked on an innovative solution search.

This report's primary goal is to present our team's efforts in designing a software model of London Underground tube system's route planner software, offering not only journey duration information, but also stop counts between stations for more efficient travel planning.

Our work has been driven by our dedication to careful design, robust implementation, and rigorous testing. Additionally, our coursework specifications required us to use Python library responsibly while adhering to its specifications governing resource use.

This report details our team's numerous tasks assigned, highlighting challenges encountered, solutions implemented, and insights gained during this project. From improving traveller experiences to analysing potential effects of line closures, our contributions to London Underground were both essential and innovative.

Task 1: Journey Itinerary and Duration

1a. Route planer and duration in minutes

Graph creation

The type of graph model is very important in graph theory and how it is used to show representations of networks in the real world. The AdjacencyListGraph class was chosen over the AdjacencyMatrixGraph class after careful consideration in order to describe the London Underground Data. There were several factors that went into this decision, including the way the underground network works and the need for computers and algorithms to complete the tasks.

The London Underground is a complex network that naturally looks like a sparse graph, where stations (vertices) are selectively interconnected(edges). Given this sparsity, the AdjacencyListGraph class becomes a more memory-efficient and flexible representation to show the data. It allows the addition of edge weights like travel times and makes it easier to use Dijkstra's algorithm for finding the single source shortest path.

key points from the document that influenced the decision to use the AdjacencyListGraph class for implementing the graph:

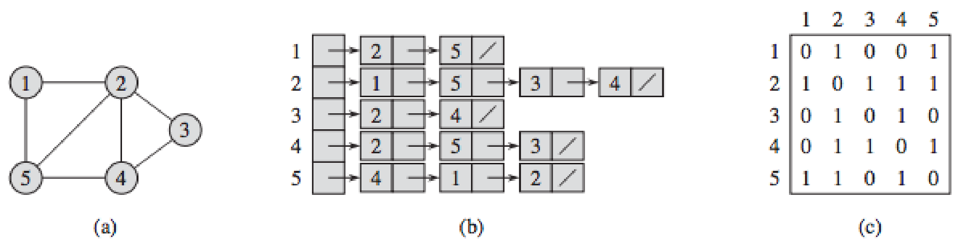
- Sparse Graphs: the adjacency-list representation is more suitable for sparse graphs where the number of edges $|E|$ is much less than the number of vertices squared $|V|^2$. If we think of the London Underground stations as nodes and the direct routes between them as edges, then the graph is sparse since not all of the stations are directly linked to each other.
- Memory Usage: the adjacency-list representation requires O(V+E) space to store the graph, making it more suitable for sparse graphs such as London Underground's limited direct connections between stations. This is beneficial in terms of memory-efficiency.
- Edge Presence: one drawback of adjacency-list representation is its time-consuming method for identifying edges; searching through this list requires time and in the worst case it has O(V) complexity. However, this limitation is not significant in our

case given that Dijkstra's main goal is locating shortest routes rather than frequently checking their existence.

- Weighted Graphs: adjacency-list representation can easily accommodate weighted graphs by including each edge's weight in its vertex in an adjacency list. Given that London Underground data includes durations as weights, this makes adjacency list representation suitable.
- Flexibility and Robustness: the adjacency-list representation is flexible enough to adapt to various graph variants, making it suitable for accommodating any requirements or changes necessary for representing and processing London Underground data.
- Algorithm Compatibility: many graph algorithms, including Dijkstra's algorithm used in this code, are compatible with an adjacency list representation, making it suitable for handling various graph operations required to process London Underground data.

Based on the characteristics of London Underground data and the given tasks (such as finding the shortest path), AdjacencyListGraph class with its adjacency-list representation was selected due to its efficiency in managing sparse, weighted graphs as well as compatibility with various graph algorithms despite slower edge presence checks.

Representation of the adjacency list adjacency and the adjacency matrix:



	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

Figure 1

Definitions and Key Notations used in our explanation:

v - specific vertex in the graph

V - total no. of vertices in the graph

N – Replace with V**

E – total number of edges in the graph

Deg(v) - is the number of nodes a specific Vertex is connected to.

Incident edges – edges that are connected to a specific vertex in the graph.

1. ***Algorithm and Data Structure Selection*:**

The adjacency list and adjacency matrix are the two main data structures used to implement a graph in a software. It is important to note that while graphs can be represented using various data structures, our library exclusively offers these two options, because for this reason the comparison is only between Adjacency List and Adjacency Matrix.

Each of these two data structures have some benefits or disadvantages depending on the operations we use to retrieve the data. Since we will use Dijkstra's algorithm to find the shortest path, we found that Dijkstra's key operation is finding the neighbors of each node or vertex. The algorithm will do retrieve the neighboring nodes for all the nodes in the graph, this means that this operation will happen O(V) times. Afte checking the time complexity for this operation in both data structures, we found that the adjacency list stores the nodes in such a way that each node is connected to a linked list which contains all the incident edges for each Vertex.

Then we conclude that for retrieving the neighboring nodes for a vertex Dijkstra's algorithm will have to traverse the linked list of the v which has the time complexity of O(deg(V)). We can also consider deg(v) as the edges v is connected to, which means that if we sum up all the deg(v) in our graph we will get the total number of edges (E). In an undirected graph these edges would have access in both directions, so in that case it is 2E. From this we conclude that the time complexity to retrieve all the neighboring nodes for all the vertices will be O(E)

On the other hand, the adjacency matrix is represented as a 2d matrix where both columns and rows correspond to the number of nodes in the graph. And each cell at the intersection of row i and column j indicates the presence and weight of an edge from node i to node j, in this cell we store the value from the wight of the edge. In conclusion, to get the neighboring nodes in an adjacency matrix for each node we need to traverse all the cells from the row of that node. Since each row has the length of V the time complexity for getting its neighbors would be $O(V)$. And considering that the algorithm needs to traverse the whole matrix, then it would have to perform no. of rows * no. of columns steps. In other words, it would have an $O(V^2)$time complexity.

If we compare the efficiency of these data structures when using Dijkstra, we get O(E) versus $O(V^2)$. This way we found out that O(E) is more time efficient than $O(V^2)$.

In our case the London tube graph is a sparse graph because each station is connected only to a few other stations and not all stations are directly connected to each other as in the case of a dense graph. Consequently, the O(deg(v)) for each vertex will be much more efficient because the average deg(v) is much lower than the number of nodes in the graph. Since o deg(v) is much less than the number of nodes in the graph it would be much less than O(V) when using the adjacency matrix.

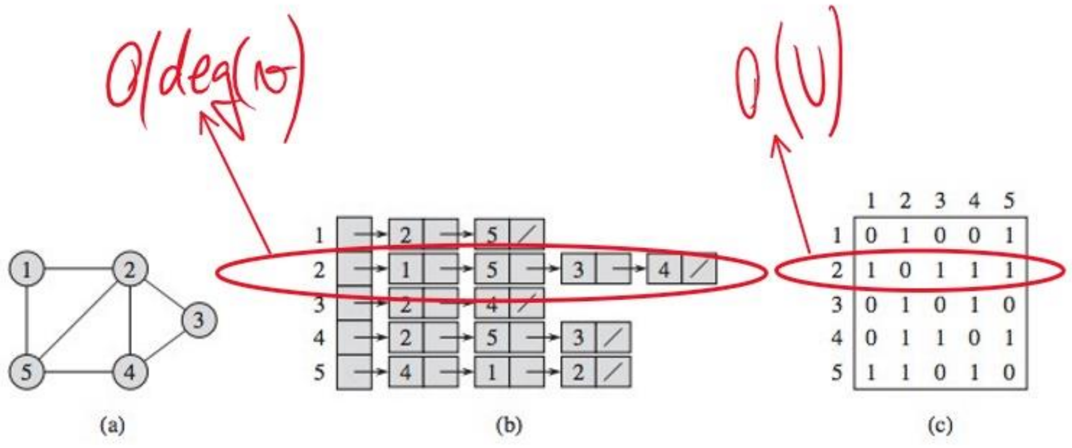


Figure 2

Also, an important difference between the adjacency matrix and the adjacency list is the space complexity because the matrix stores a cell for the connections in $O(V^2)$. The reason for this is because the number of cells will be no. of columns * no. of rows. On the other hand, the space complexity for the adjacency list is O(E+V) because it stores only the necessary connections. Considering we have a sparse graph the space complexity will be much more efficient for our graph if we use an adjacency list. In a dense graph the space and time complexity would be similar in both data structures, but this would not apply in our case.

After choosing the data structure we looked at Dijkstra's implementation and saw that it uses a priority queue implemented trough a Min-Heap to dequeue the minimum distances for a node and to reorder each node's value every time the distance is updated. The operations that are used in the priority queue in Dijkstra are:

- Initialization: set initial distances for each edge to infinity this takes O(V)

- For each of the vertices we dequeue the minimum distance in the queue, dequeue has a time complexity of O(log V) and if we do this for all the vertices it will be O(V log V)
- And the last operations are for each edge V, it potentially performs the decrease Key operation, which is done when a new shorter path is found to the specific node. This decrease key operation will update the nodes key value which is the distance and potentially will move the element to a different position. The time complexity of this operation is (V log V) because moving the element after changing the value is O (log V) and doing it for each vertex it will be O(V log V)

In summary, with the priority queue, Dijkstra's algorithm retrieves and dequeues the node with the smallest estimated distance from the starting point. The dominant time complexity of the priority queue is O(V log V) which is ideal for sparse graphs where E is much less than V^2.

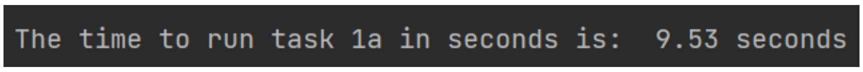
In conclusion for Dijkstra's algorithm, using an adjacency list for a sparse graph like the London tube network is time-efficient due to the O (E) complexity for iterating over all edges or all the connections of the nodes, combined with a priority queue operation dequeue O(V log V) and key update O(V log V) , making the total time complexity O(V log V + E). The reason E is added to the dominant term is that in a dense graph E can be significantly larger than V and that would influence the number of operations significantly. However, E can still be large enough in comparison to V such that E is not negligible, as a result we arrive to the final time complexity of O(V log V + E)

In graph theory, a dense graph is one where the number of edges E is close to the maximum number of edges possible. The maximum number of edges in an undirected graph without self-loops is $\frac{(V^2)}{2}$ and because of that after dropping the constant terms we would have the final time complexity of $O(V\, log\, V + V^2)$

As a conclusion using adjacency list is more efficient than using adjacency matrix, especially considering the efficiency in the time complexity $O(V^2)$ and the space complexity $O(V^2)$ of the Adjacency matrix. Also adding to the benefits is the space complexity for an adjacency list is O (E), which is preferable for sparse graphs.

Empirical analysis:

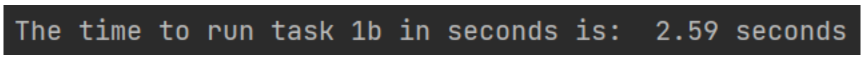
The empirical analysis of Dijkstra's algorithm for task 1a yielded a running time of 9.53 seconds to compute the shortest path between user-specified starting and destination points (see Figure 3).



```
The time to run task 1a in seconds is:  9.53 seconds
```

Figure 3

The empirical analysis using Dijkstra's algorithm for task 1b recorded a running time of 2.59 seconds to construct travel time histograms between station pairs in the London Underground (see Figure 4).



```
The time to run task 1b in seconds is:  2.59 seconds
```

Figure 4

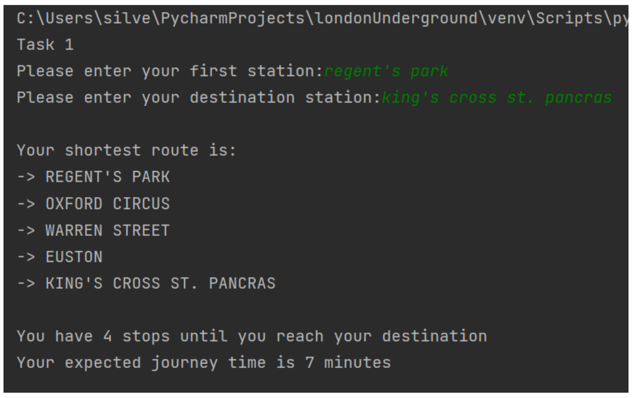
2. *Test Data Selection*

See Testing for correctness and efficiency section.

3. *Result and Functionality Presentation*

The functionality of this task is implemented as follows:

- Taking user input for the starting point and the destination. This input is not case sensitive as the program converts all the input string to upper case. One future improvement could be to apply the longest common substring algorithm in case the user misspells the stations and autocomplete to the correct station name. But for now, the user will need to reenter the destination and starting point stations if they are misspelled.
- Then Dijkstra's algorithm is run, and the shortest path is found for that route.
- The program Displays shortest path with all the so in between and the start and final station is displayed as well.
- As a last step the Total travel time to arrive to the destination is displayed, as well as the total no. of stops.



```
C:\Users\silve\PycharmProjects\londonUnderground\venv\Scripts\py
Task 1
Please enter your first station:regent's park
Please enter your destination station:king's cross st. pancras

Your shortest route is:
-> REGENT'S PARK
-> OXFORD CIRCUS
-> WARREN STREET
-> EUSTON
-> KING'S CROSS ST. PANCRAS

You have 4 stops until you reach your destination
Your expected journey time is 7 minutes
```

Figure 5

4. *Final Remarks and Limitations*

One of the strengths of Dijkstra's algorithm is that it is faster than Bellman-Ford considering the time complexity and the empirical evidence we saw. But it can't be used in every situation because one important limitation that needs to be considered when using Dijkstra is that it will not work with negative weights. And this limitation comes from how the algorithm works on the graph. If we consider a small graph like the one underneath for the sake of demonstrating the limitation. If Dijkstra's algorithm starts from node A then it would consider the two edges it has as connections, which are AB(5) and AC(4). After relaxing these edges it chooses the edge with a lower value which is AC(4). Then it computes the same steps as for node A. Retrieves all the connected edges to node C unless the edge is incident to a node already visited, then that edge will not be considered. In this case node A vas already marked as visited so AC will not be considered only CB(2) will be considered. Then the path to node B is considered trough node C which would be AC(4) +CB(2)= 6 and this value is compared with the value already stored for the distance of B from the first step of AB(5). AB(5) is less than 6 and because of this it will keep AB as a shortest path to B.

After seeing how the algorithm works, considering the second example there is a negative edge. If the algorithm executes on this graph, it will give the wrong shortest result. The problem the algorithm will run into is when starting at node A for example and considering the edges incident to node A. AB(5) and AC(4) it chooses the lowest value of AC. After the only edge from C is CB(-2), then updating the path to B will give AC(4) + CB(-2) = 2 which is lower than AB(5) so the path is updated. The problem arise when searching for the shorter path to C.

Dijkstra has the shorter path saved as AC (4) but the actual shorter path would be AB(5)+BC(-2) = 3 and this value is lower than the saved one AC(4)

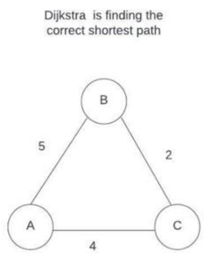


Figure 6

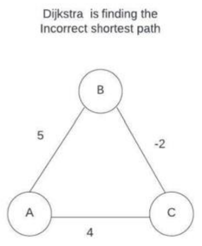


Figure 7

It is worth noting that not all shortest paths will be incorrect in the 2nd situation, but we can't rely on the accuracy of the algorithm anymore in this case and because of this it wouldn't be a good fit to use it in a graph which has negative weights. Luckily in our London Underground graph there are no negative edge weights or travel times and because we only have positive numbers Dijkstra's algorithm is a really good fit for our software.

1b. Histogram of journey times in minutes

How we obtained the Histogram data:

The histogram data obtained is created by two loops this way we are able to compute all the possible combinations of station pairs from our graph. The first loop iterates through all the vertices in the graph and runs Dijkstra's algorithm with each vertex considered as a starting point. The Second loop will start backtracking all the possible paths from that starting point once the first loop is completed. As mentioned before the time complexity for Dijkstra's algorithm is O(V log V + E) and the inner loop will be O(V) because we are considering all the edges at one time. This would result in the overall time complexity of $O(V \cdot (V \log V))$. Also, worth mentioning that we implemented an If statement to check if the path is already computed to further reduce the operations, this is needed because of our graph is undirected and it would compute A to B and B to A which would result in the same travel time or number of stops.

For computing all the station pairs and finding the shortest path one function is reused throughout the whole program when the task is to display all the combinations in a histogram.

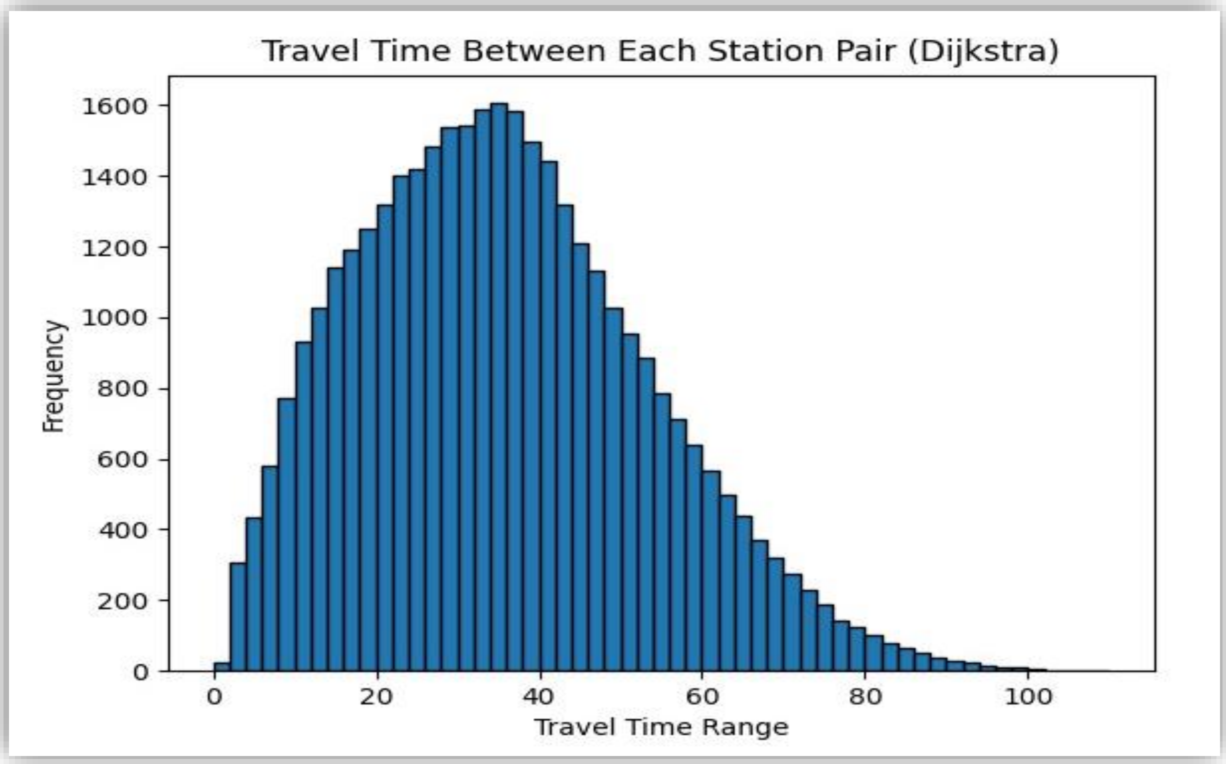


Figure 8 - Travel times between each pair of stations

The histogram displays a bell-shaped distribution with a positive skew of travel times between pairs of stations calculated using the shortest paths identified by Dijkstra's algorithm in the London Underground.

The y-axis frequency indicates how many travels occur into each time interval, while the x-axis pairs of stations are arranged according to increasing travel times. The middle of the x-axis shows the most common travel time, suggesting an optimal duration.

Analyzing the frequency distribution of the histogram, this indicates that the majority journey duration between station pairs is concentrated in a central, with the mode. The most common journey duration occurring in the 20-30minutes range, showing that a significant number of station pairs have routes within this range.

The histogram shows positive skewness, which is evident from its asymmetry, shorter journey times, likely within central London or between closely situated stations, on the left, and a longer tail on the right side. Nevertheless, the skewness is not pronounced, suggesting that the variations in journey durations do not deviate much from the average.

Key observations:

- The first quartile ranged from 0-10 minutes, showing a considerable frequency of journeys. This might reflect the dense network of stations in central London or efficient connections in that area.
- In the right-skewed tail, particularly in the 40–70-minute range, there is a noticeable decrease in frequency beyond the central peak, as the histogram shows. This could indicate longer journeys, potentially from central zones to outer zones of London.
- The uppermost journey time ranges (80-100 minutes), while less frequent, are present and may indicate exceptionally long journeys across the network from one end to the other.

Histogram analysis demonstrates a well-designed London Underground network with optimal journey times for travelers.

Task 2: Station Count Calculation

2a. The count of stations between the starting point and the destination

1. ***Algorithm and Data Structure Selection*:**

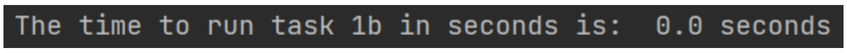
For counting the shortest route Dijkstra's algorithm will not find the correct lowest number of stops if the graph will have the travel time as edge weights. And because of that when loading the data to London Underground is stored in two separate adjacency lists. One graph has the edge weights as the travel time this graph is used when the purpose of the task is to find the shortest travel time. And the other graph has the edge weight of 1 for every edge this way

Dijkstra’s algorithm will find the route that has the least amount of stops or shortest paths by stops. Future improvements can be made by using just one graph and adding the travel time and 1 for each edge as the same wight represented as a linked list. This way the program can be written to use only one graph.

The functionality of backtracking from the destination to the initial station and displaying all the stops on the route is the same as task one. The backtracking is a function used every time we need it for the tasks, this wat reusing the code and making it more readable.

Empirical analysis:

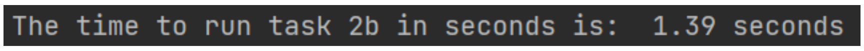
The empirical analysis of Dijkstra's algorithm for task 2a yielded a running time of 0.0 seconds to compute the shortest path between user-specified starting and destination points based on count of stations also seen in Figure 9 below**.**



```
The time to run task 1b in seconds is:  0.0 seconds
```

Figure 9 - Running Time for Task 2a.

The empirical analysis using Dijkstra's algorithm for task 2b recorded a running time of 2.59 seconds to construct travel time histograms between station pairs in the London Underground seen in Figure 10 below.



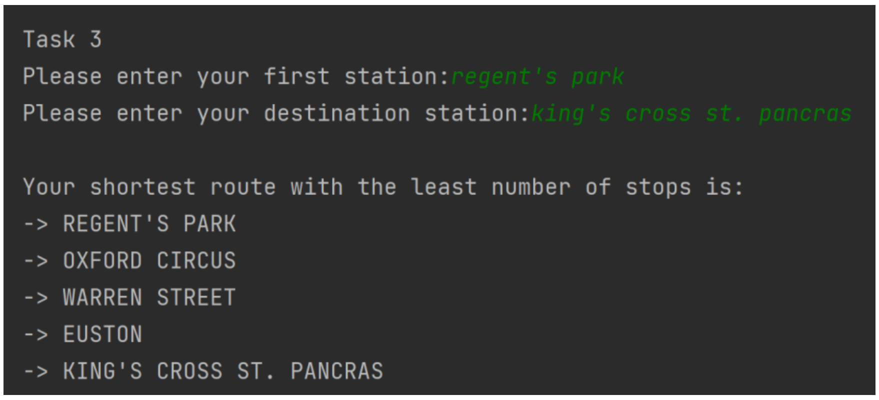
```
The time to run task 2b in seconds is:  1.39 seconds
```

Figure 10 - Running Time for Task 2b.

2. *Test Data Selection*

See Testing below in the testing section for correctness and efficiency.

3. *Result and Functionality Presentation*



```
Task 3
Please enter your first station:regent's park
Please enter your destination station:king's cross st. pancras

Your shortest route with the least number of stops is:
-> REGENT'S PARK
-> OXFORD CIRCUS
-> WARREN STREET
-> EUSTON
-> KING'S CROSS ST. PANCRAS
```

Figure 11 - Output of Task 2a

<u>2b. Histogram of numbers of stops between each station pair.</u>

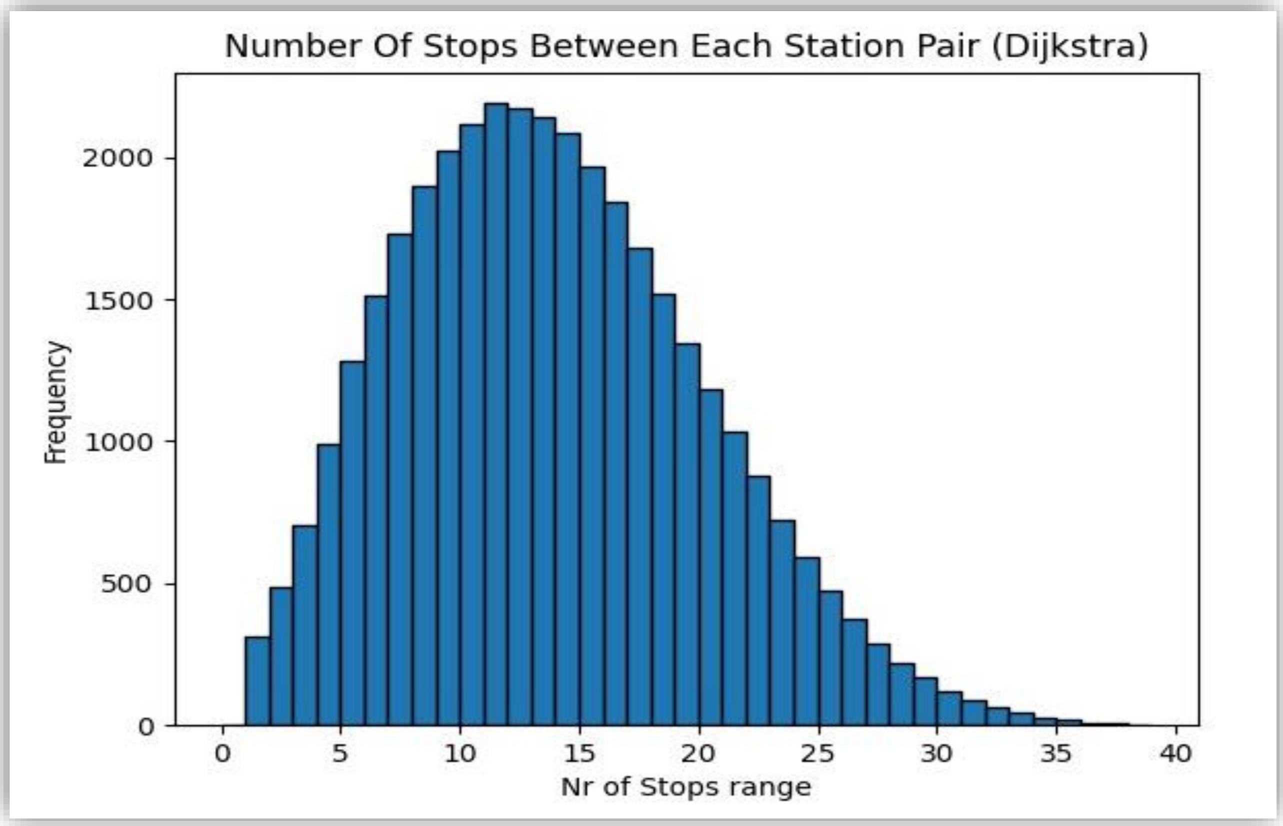


Figure 12 - Histogram based on numbers of stops between each station pair.

Task 2b's histogram shows Dijkstra's algorithm's frequency distribution of numbers of stops between each station pair stops in the London Underground.

Examining the x-axis which indicates the range of the numbers of sops (ranging from 0 to around 35), it is obvious that the distribution mode is centred around 10-20 stops. The histogram peaks here, suggesting that this rage has the most commune journeys length.

The y-axis shows the frequency of the occurrences, of these events. The highest frequency is close to 2000, indicating that a range of 15-20 stops are made between station pairs. As the histogram tails off to the right, fewer journeys have more stops, which could be because the network has fewer long-distance routes or because the Dijkstra it's minimise the stops on longer routes to speed up the travel.

Key observations:

- Range 0-5 stops show a medium frequency which could indicate a moderate number of very short paths within the network.
- The middle range 10- 20 implies that the majority of travellers using the London Underground network are likely to be travelling on medium-length route and have this range of stops.
- The range 20 to around 40 it has a decreased frequency as the numbers of stops increases beyond 20 which can emphasise the efficient design of the system offering quicker routs across the city with fewer numbers of stops. Beyond 25 stops, the histogram drops off but does not drop to zero, indicating that longer journeys are still important to the network. The route planner must appropriately account for these less frequent yet possibly routes with high numbers of stops.

In conclusion, the histogram shows a well-functioning route system with an average amount of stops for most travels, which the software model should account for to provide optimum route and accurate journey duration estimations. The spread and skewness of the histogram provide insights into the network's design and the algorithm's performance, both of which are crucial for developing a user-centred route planning tool.'

Task 3: Alternative Algorithm Comparison

3a. Algorithm Selection

1. Algorithm and Data Structure Selection:

The Bellman-Ford algorithm is commonly used in computer science to calculate the shortest paths from a single source in a graph, even when the edge weights can be negative. However, within the specific framework of our graph, which is undirected and does not have negative weights, the value of the Bellman-Ford algorithm does not lie in its capability to handle negative weights, but rather in its systematic process of relaxing edges to reach the shortest paths.

The graph is represented using an adjacency list ‘AdjacencyListGraph’, which is efficient for sparse graphs like the London Underground network where each station (node) is connected to only a few other stations. This data structure allows for an efficient traversal of connected nodes, which is crucial for the repeated edge relaxations performed by the Bellman-Ford algorithm.

The time complexity of the Bellman-Ford algorithm is O(VE) where V is the number of vertices and E is the number of edges in the graph. When using an adjacency list, the algorithm iterates over all edges for each vertex (V-1 times). Given that each edge is checked once for every vertex, the time complexity of O(VE) is maintained. The adjacency list facilitates the rapid access of all edges originating from a given vertex, which is essential to the algorithm as to the adjacency matrix, especially when the graph is sparse.

The space complexity of an adjacency list is O (V + E). This is due to its capability of storing a list of edges for every vertex. In a graph sparse graph where the number of edges (E) is significantly smaller than the square of the number of vertices (V^2), using an adjacency list is more space-efficient compared to an adjacency matrix. The algorithm itself needs more space to store the distances between each vertex and its predecessor. Since it keeps track of two lists: one for the distance 'd' and another for the predecessor’s 'pi', this is usually O(V).

Performance and Efficiency Analysis

Adjacency list representation of a subset of our graph (London Underground) where each node's list of adjacent nodes is sorted. Each line starts with the node name, followed by their respective adjacent nodes with the weight (duration) of the edge.

Regent's Park (RP):→ Oxford Circus (2)
Oxford Circus (OS):→ Regent's Park (2), Warren Street (2)
Warren Street (WS):→ Oxford Circus (2), Euston (1)
Euston (EU):→ Warren Street (1), King's Cross St. Pancras (2)
King's Cross St. Pancras (KC):→ Euston (2)

Figure 13 - Agency List Representation

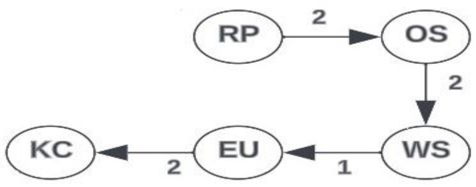


Figure 14 - Graph Representation

This complexity appears from:

- Initialization: bellman-fords starts by setting the initial distance to the source vertex to zero and the other as infinite, this step happens once and takes O(V) time, where V is the number of vertices in the graph.

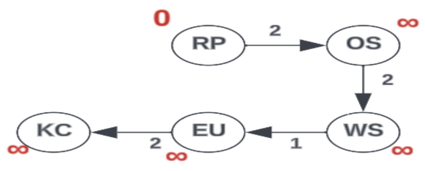


Figure 15 - Initialization

- Edge Relaxation - Main Loop: In the Bellman-Ford algorithm, the main loop iterates V−1 times over all edges in the graph, where V is the number of vertices. In each iteration, every edge is checked and potentially relaxed. Although the relaxation operation takes O (1) time, each iteration takes O(E) time, where E is the number of edges. The relaxation of an edge is an O (1) operation (constant time), but since is done for every edge in each of the V-1 iterations, it contributes to a total time complexity of O(E×(V-1), which simplifies to O(VE).

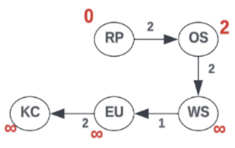


Figure 16 - First Iteration

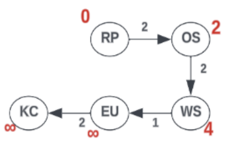


Figure 17 - Second Iteration

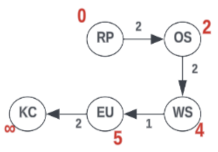


Figure 18 - Third iteration

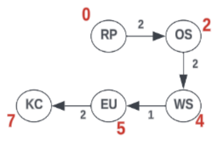


Figure 19 - Fourth iteration

For instance, in our example with 5 vertices and 4 edges, we will have 4 iterations (since V - 1 =5 -1 = 4):

- In the first iteration we start from RP with a distance of 0, we relax the edge RP—2→OC. Oxford Circus distance is updated from ∞ to 2.
- Next, we relax the edge OC—2→WS. Because the OC distance is 2, WS distance is updated from ∞ to 4 (2 from RP to OC plus 2 from OC to WS).
- Then we relax WS—1→EU edge. WS distance is 4, so EU distance is updated from ∞ to 5 (4+1).
- Finally, we relax EU—2→KC, with EU distance of 5, KC distance is updated from ∞ to 7 (5+2).

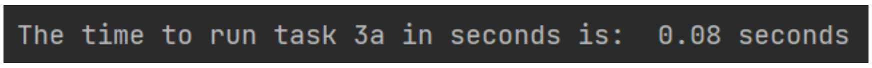
This process of iteration repeats but because we do not have negative weights in this graph, the algorithm may find the shortest path in fewer than V -1 iteration. In our example, the shortest path is likely to be determined within the first iteration itself. The remaining iterations would demonstrate that no shorter paths exist, which is crucial in complex or negative-weighted graphs. This verification phase is essential to the algorithm's design and ensures the shortest path is correct, even if no modifications occur in the remaining iterations.

- Negative-weight cycles: Even though our graph does not have a negative weight, a check for negative-weight cycles in the Bellman-Ford algorithm is necessary to ensure the accuracy of the shortest paths determined. It works as follow: after the main loop has completed, where the algorithm makes one additional pass over all edges to identify any negative-weight cycles, operation with a time complexity of O(E). Although this function will not find negative cycles in our specific graph, it is still executed as a part of the algorithm’s structure and ensure the correctness of the results, but it does not have an impact to the overall complexity due to the absence of a negative cycle.

The overall time complexity of the Bellman-Ford algorithm is O(VE), the reason for this is that the most time-consuming aspect of your algorithm is the nested loop structure, in which you iterate |V|−1 times overall E edges.

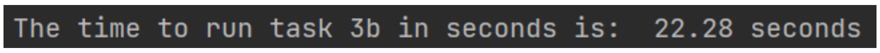
Empirical analysis:

The empirical analysis of Bell-man Ford’s algorithm for task 3a produced a running time of 0.08 seconds seen in Figure 20 and a running time of 22.28 seconds seen in Figure 21, to compute the shortest path between user-specified starting and destination points.



```
The time to run task 3a in seconds is:  0.08 seconds
```

Figure 20 - Running Time Task 3a.



```
The time to run task 3b in seconds is:  22.28 seconds
```

Figure 21- Running Time Task 3b.

2. *Test Data Selection*

See Testing for correctness and efficiency section.

3. *Result and Functionality Presentation*

Code compliance and library used in functions:

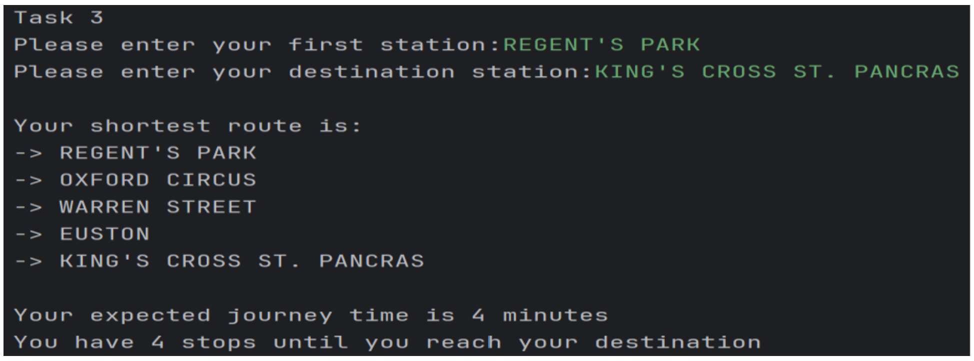
The results of the Bellman-Ford algorithm implementation are as follows:

Total Duration: successfully calculates the total travel time between stations.

Number of stops: it accurately counts the number of stops in the shortest path.

Route: it provides a list of stations that form the shortest path.

Functionality presentation:



```
Task 3
Please enter your first station:REGENT'S PARK
Please enter your destination station:KING'S CROSS ST. PANCRAS

Your shortest route is:
-> REGENT'S PARK
-> OXFORD CIRCUS
-> WARREN STREET
-> EUSTON
-> KING'S CROSS ST. PANCRAS

Your expected journey time is 4 minutes
You have 4 stops until you reach your destination
```

Figure 22 - Output of the Task 3a

4. *Final Remarks and Limitations*

The algorithm effectively calculates the shortest paths in the London Underground network. The use of adjacency list and the careful selections of test data underscore the algorithm reliability and efficiency.

Although Bellman-Ford's $O(V \cdot E)$ time complexity makes it less efficient when negative weights are absent (as compared to alternatives like Dijkstra's), it still guarantees achieve to a solution after V-1 iterations and guarantees it as being accurate despite appearing less efficient than alternatives because of the sparsity of the London Underground network were the number of edges E is much lower to V^2, the square of the number of vertices. Our graph being undirected with no negative weights may render its efficiency suboptimal when compared to alternatives; nevertheless, meets the requirements of achieving the shortest paths.

3b. Histogram of numbers of stops between each station pair.

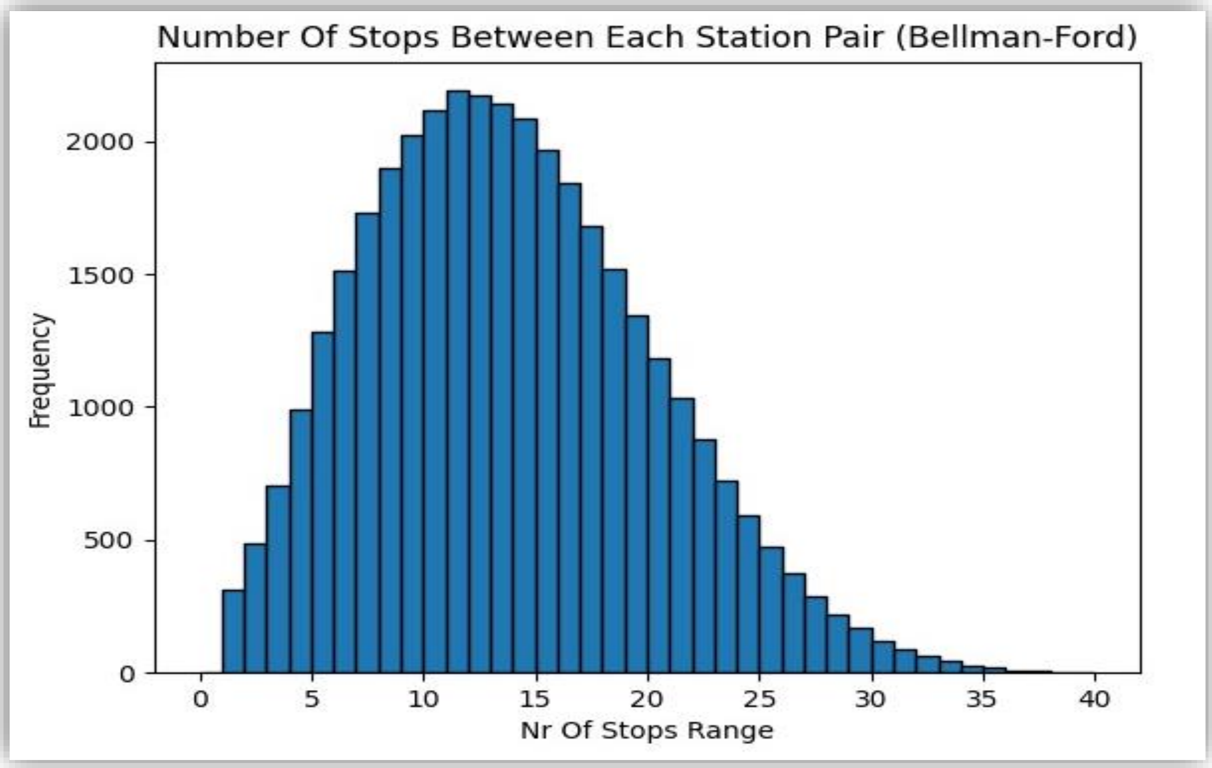


Figure 23 - Histogram displaying numbers of stops between each station pair (Bellman-Ford)

Based on the algorithm in Task 3a, the histogram displays the London Underground numbers of stops between each station pair. The data is generated from a different algorithm than the one used in Task 1a, focusing on the count of stations/stops instead of journey duration in minutes.

The y-axis represents the frequency of each stop count, while the x-axis represents station pairs arranged according to the number of stops, with the highest bar representing the mode of the data. This suggests that most routes within the Underground network have a constant and steady number of stops between each pair station.

The data indicates positive skewness, as seen by the presence of a right-side tail. This implies that there are station pairs with a number of stops exceeding the mode, although this is less common specially form 35 to the end where 2 bins appear.

Key observations:

- Initial increase (0-10 stops) where the network's frequency rises gradually from 5 to 10 stops on the x-axis, suggesting a moderate number of stops. This might reflect well-connected central zones with close stations.
- Range 10-15 is the most frequent range of stops having the highest frequency which is above 2000 showing that the majority of journeys involve 10-15 stops. This indicates a notable consistency in the durations of journeys throughout the network and that the network's design is optimised for efficiency, balancing the number of stops with the travel time.
- From the 15-20 stops range we can notice the decline in frequency which can suggest routes between fare zone in Underground system or the presence of alternatives routes which requires less time and stations for such distance.
- The above 30 range where were the decline in frequency continues suggests the existence of less frequent, but possible, longer routes that travellers going from one end of the network to the other may use, or that may require multiple changes.

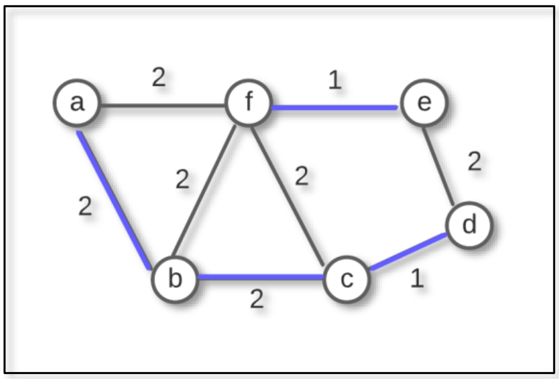
This histogram is essential for evaluating the Bellman-Ford algorithm's routing detail capture and ensuring that the software model gives correct and valuable information to end-users.

Task 4: Tube Line Closure Analysis

4a. Algorithm Selection

1. Algorithm and Data Structures selection:

To determine which tube lines between neighbouring stations could be closed, our graph $G = (V, E)$, an undirected and connected graph, with V the set of stations and E the set of possible connections between station pairs, where for each edge $(u, v) \in E$, the weight $w(u, v)$ represents the cost (the time in minutes) between the interconnected u and v. First, the team needed to determine a subdivision tree T without any repeated cycles which connects all the stations retrieving the least amount of time in minutes. T must form a tree because it is non-cyclic and unites all vertices. Therefore, the name "spanning tree" and because it "spans" the graph, hence the ***"minimum-spanning tree" algorithm*** helps find the tree T . In this report, we will be utilising ***Kruskal's algorithm*** to find the minimum-spanning tree of the entire graph.



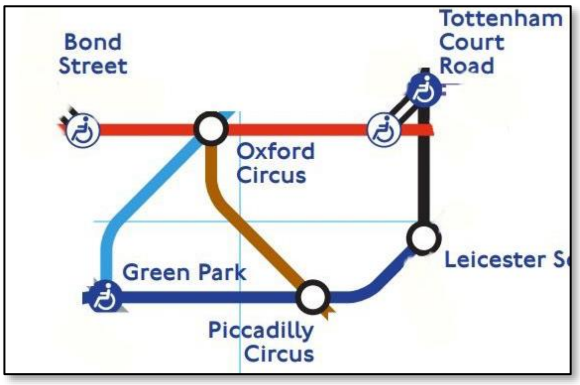


Figure 24 - An undirected graph with 6 vertices and 8 edges and its minimum spanning tree

From Figure 24 it can be seen that a minimum spanning tree of a segment from the entire London Underground Map with six vertices (stations) from "a" to "f". Each edge consists of a weight shown in the figure, and the minimum spanning tree is represented with the blue edges. The represented tree has a total weight of 6. However, by removing the edge (a, b) and replacing it with (a, f), a new spanning tree forms with the same total weight of 6, meaning that the initial tree is non-distinctive.

Both Kruskal and Prim's algorithms use a certain "greedy" approach to contract a minimum-weight spanning tree. The two algorithms are different in terms of how they employ the purpose to find the minimum spanning tree for G. Kruskal's algorithm, unlike Prim's, it doesn't construct the minimum-spanning tree by developing a single tree to ultimately span the graph G. The algorithm repetitively manages a set of groups ordered as a ***forest*** and endlessly joining pairs of groups to finally extent the concluding graph with only one collection, and from Figure 1 it can also be statistically verified that it indicates to the best result. The algorithm choses edge by edge in a nondecreasing order. Then if two nodes, such as node ***a*** and node ***b***, are connected, they form an edge ***e,*** and the algorithm ***merge-sort*** helps ordering the edges by weight in a nondecreasing order. As seen in Figure 24 above, our minimum spanning tree should reject any connections which could form a cycle, since the final tree must be noncyclic. It can also be seen that the segment from Figure 24 is a disjoint segmentation of the vertices V

due to the fact that the algorithm adds and edge *(u, v)* to the minimum-spanning tree *T* forming a division of the set of vertices *V* with V_1 the group of vertices *v* and V_2 the remnants of vertices in *V.* This paper gives an overview of Kruskal's algorithm applied to the graph in contrast with ***Dijkstra's algorithm,*** as explained above in task 1a, and compares and contrasts the results of finding the shortest path between any two given stations.

As discussed above, the implementation of Kruskal's algorithm within the London Underground Map Graph, is utilising again the Adjacency List Graph for representing the graph and Dijkstra for finding the shortest route. It also utilises the minimum-spanning tree consisting initially of a ***forest*** of clusters. The algorithm also applies two more data structures***, disjoint partitions,*** and ***union-find.***

- Initialization:

Since our graph is an undirected, connected, and weighted graph, Kruskal's algorithm takes the space and time complexity of $O(E \log E)$ time complexity for sorting the edges and $O(E + log\, V)\ or\ Big - O(279 + log 280)$ time complexity to operate the total amount of union-find processes and preventing arising any cycles. The ***first step*** to initialise the algorithm we will be utilising a forest array which will take $O(V)$ time complexity. The ***Second step*** is creating an empty array to store the weighted edges and will take the time complexity of $O(E)$. Sorting the edges represents the ***third step*** to initialise the algorithm also using the merge sort algorithm and with a time complexity of $O(E \log E)$. The algorithm's primary loop will implement methods such as finding unions and sets by iterating over every single edge in the ordered edges array. With path density and union by sorted weights, in the worst case, each find, and union process will take approximately $O(1)$ time complexity. Thus, the loop will perform a time complexity of $O(E * \hat{\alpha}(V))$ and in this case, according to Cormen et al. (2022), $\hat{\alpha}(V)$ represents the inverse Ackermann function which exhibits a very slow-growing rate.

- Time and space complexity:

Merge-sort is typically the most unproductive when it comes to sort the station pairs in terms of time complexity. When Kruskal's algorithm runs, the disjoint-set processes take $O(V)\ or\ Big - O(280)$ space (where V represents the total number of stations within our graph). In the perspective of the tree-based operation of a separate data structure shown in Figure 24 above, the search for a particular element e involves navigating beginning with the position of e to the root of the tree. As per the worst-case scenario, this procedure takes $O(n)$

time. Moreover, the union operation for elements e and f can be initiated by integrating one of the trees as a subtree of the other. This method implies finding the roots of the two trees and then, with a supplementary $O(1)$ time, one root will point to the other root by updating the top-most point of the tree.

Merge Sort	$O(E\ log\ E)$ time
Union-Find	$O(E + log\ V)$ time
Edge Iteration	$O(E \log V)$ time
Disjoint-Set Data Structure	$O(log\ V)$ time
Edges	$O(E)$ space
Disjoint-set as forest	$O(V)$ space
Total Space Complexity	$O(V + E)$ space
Total Time Complexity	$O(E\ log\ E)$ time

Table 1 – Time and Space Complexity

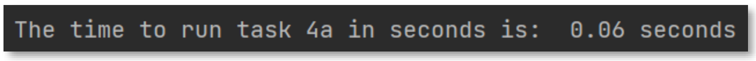
To conclude, the most dominant operation of Kruskal's algorithm, as seen from the table above and also from the given algorithm, is the process of sorting the edges in a nondecreasing order with a total time complexity of $O(E\ log\ E)$ or in our case of the London Underground Network, $O(322\ log\ 322)\ or\ Big - O(807.530)$. This means that the algorithm takes at least a total amount of approximately 807.530 operations to sort the edges. Additionally, due to the fact that our graph is a sparse graph, and the number of edges is higher than the number of vertices, the process of sorting those edges is in fact the topmost predominant process. Thus, the total time complexity will also be $O(322\ log\ 322)\ or\ Big - O(807.530)$.

- Kruskal's Algorithm with Adjacency List Graph versus Adjacency Matrix comparison:

Applying the adjacency matrix to the London Underground Tube Map, would be more disadvantageous against the adjacency list graph if our graph would have been a dense graph for several reasons which are explained in our graph creation above. Therefore, Kruskal's algorithm would have taken extra processing complexity with a worst space complexity of $O(V^2)$ and an overall time complexity of $O(V^2 \log V^2)$.

Empirical analysis:

The empirical analysis Kruskal's algorithm for task 4a produced a running time of 0.06 seconds to compute the shortest path between user-specified starting and destination points.



```
The time to run task 4a in seconds is:  0.06 seconds
```

Figure 25 – Running time of Kruskal's algorithm.

2. Test data selection:

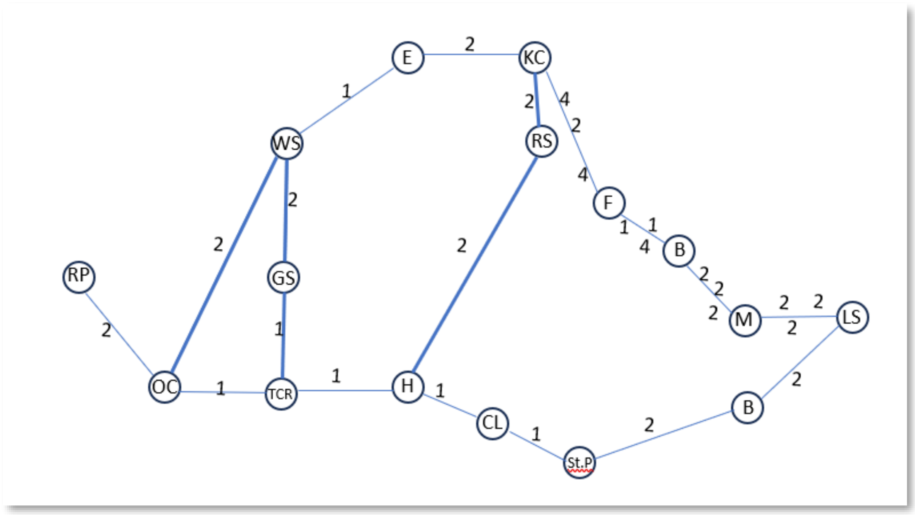


Figure 26 Shortest Path Between Regent's Park and King's Cross St. Pancras before and after closure

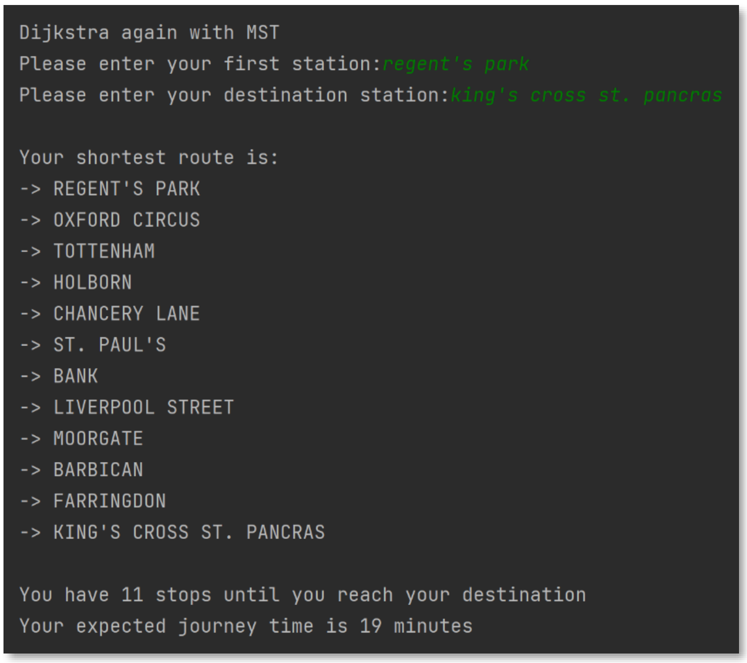
Figure 26 above gives an overview of how Kruskal's algorithm functions between two given stations. For instance, the shortest route between Regent's Park (RP) and King's Cross St. Pancras, firstly by using Dijkstra's algorithm as explained in task 1, it first iterates the edges as follow: RP OC (2 minutes), OC → WS (2 minutes), WS → E (1 minute), E → KC (2 minutes), with a total amount of time in minutes (weight) of 7 minutes. Therefore, the result from running the software is accurate. However, after the following edges: OC → WS (2 minutes), TCR → GS (1 minute), GS → WS (2 minutes), H → RS (2 minutes), RS → KC (2 minutes), are deleted by utilising Kruskal, clarified above in task 4a, a new shortest route arises between the two given stations with a total time of 19 minutes and 11 stops.

3. *Result and Functionality Presentation:*

Code compliance and library used in functions:

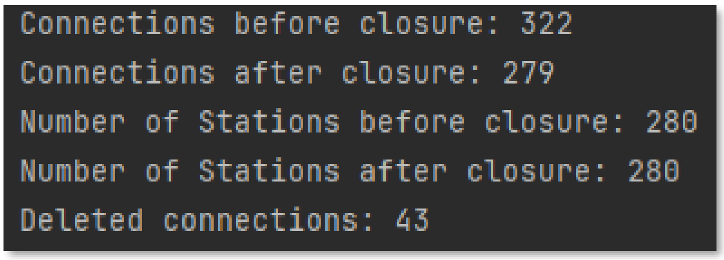
Kruskal's algorithm for the minimum spanning tree executed again with Dijkstra's algorithm results are as follows:

It can be seen from Figure 27 below that the number of stops between Regent's Park and King's Cross St. Pancras is 11 and the time to reach the destination is 19 minutes. The software is running again with the same input stations as mentioned above in task's one results, however, this time having a minimum spanning tree implemented by applying Kruskal's algorithm.



```
Dijkstra again with MST
Please enter your first station:regent's park
Please enter your destination station:king's cross st. pancras

Your shortest route is:
-> REGENT'S PARK
-> OXFORD CIRCUS
-> TOTTENHAM
-> HOLBORN
-> CHANCERY LANE
-> ST. PAUL'S
-> BANK
-> LIVERPOOL STREET
-> MOORGATE
-> BARBICAN
-> FARRINGDON
-> KING'S CROSS ST. PANCRAS

You have 11 stops until you reach your destination
Your expected journey time is 19 minutes
```

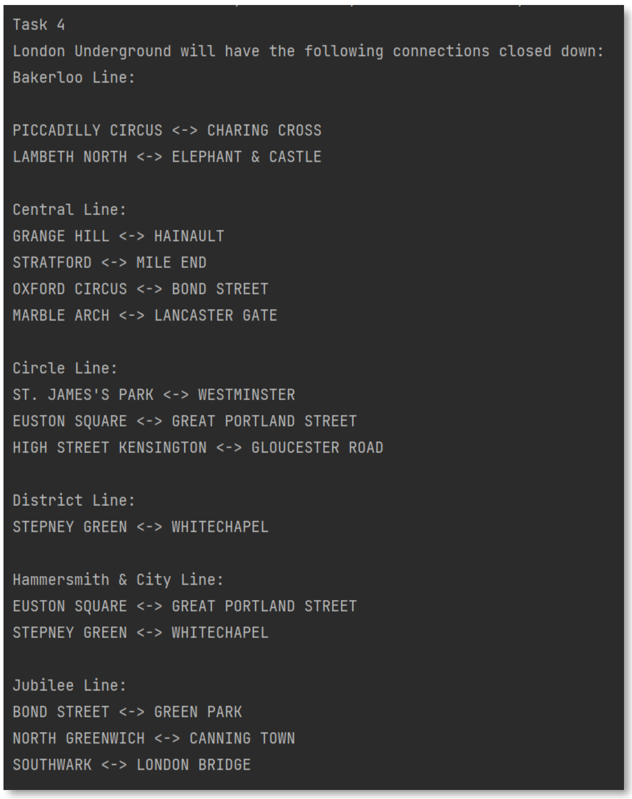
Figure 27 Running Dijkstra with Kruskal's for the minimum spanning tree between Regent's Park and King's Cross St. Pancras for the shortest route.



```
Connections before closure: 322
Connections after closure: 279
Number of Stations before closure: 280
Number of Stations after closure: 280
Deleted connections: 43
```

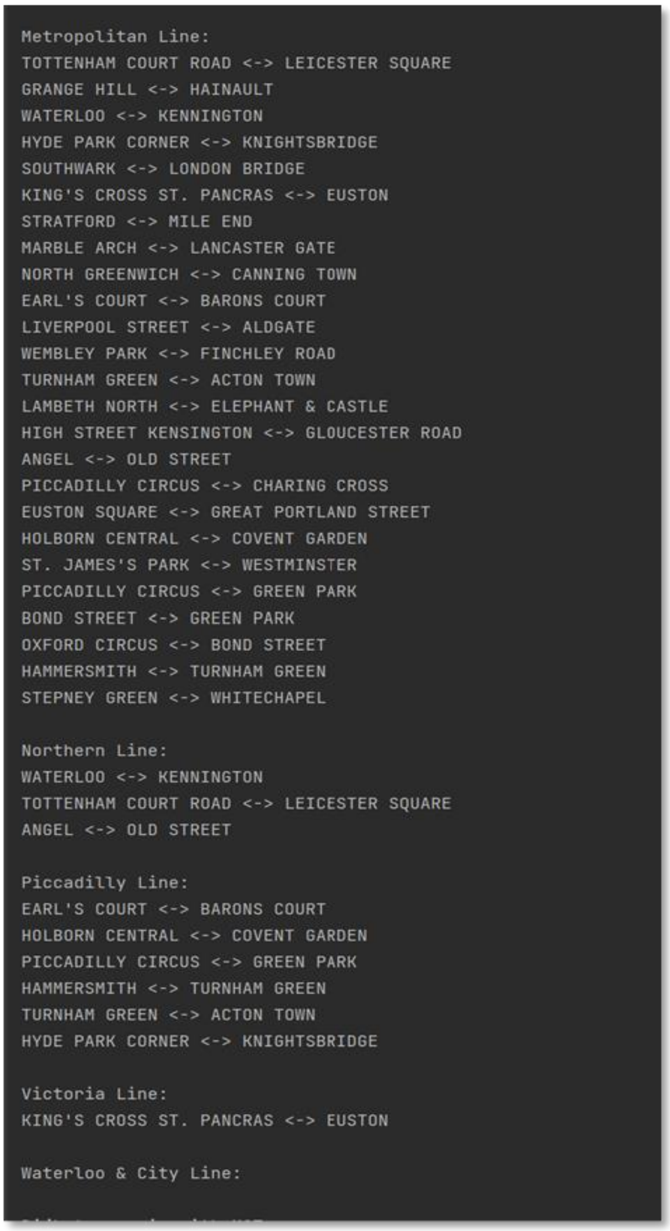
Figure 28 Results from applying Kruskal for MST before and after closure.

As seen from Figure 28 above, our graph before representing the graph applying Kruskal's algorithm to find the minimum spanning tree, the number of connections between each station pairs were a total of 322. However, after finding the minimum spanning tree, the total number of deleted edges was 43 with the remaining number of 279 connections. It can also be seen that the number of stations (vertices V) remained the same. As mentioned above, the algorithm could not close more than 43 edges because of a constraint in the existing edges, meaning that it could form a cycle. In conclusion, the main reason why the algorithm cannot delete more than that number of edges is due to the fact that our graph is a connected graph.



```
Task 4
London Underground will have the following connections closed down:
Bakerloo Line:

PICCADILLY CIRCUS <-> CHARING CROSS
LAMBETH NORTH <-> ELEPHANT & CASTLE

Central Line:
GRANGE HILL <-> HAINAULT
STRATFORD <-> MILE END
OXFORD CIRCUS <-> BOND STREET
MARBLE ARCH <-> LANCASTER GATE

Circle Line:
ST. JAMES'S PARK <-> WESTMINSTER
EUSTON SQUARE <-> GREAT PORTLAND STREET
HIGH STREET KENSINGTON <-> GLOUCESTER ROAD

District Line:
STEPNEY GREEN <-> WHITECHAPEL

Hammersmith & City Line:
EUSTON SQUARE <-> GREAT PORTLAND STREET
STEPNEY GREEN <-> WHITECHAPEL

Jubilee Line:
BOND STREET <-> GREEN PARK
NORTH GREENWICH <-> CANNING TOWN
SOUTHWARK <-> LONDON BRIDGE
```

Figure 29 Adjacent Station Closed List No. 1



```
Metropolitan Line:
TOTTENHAM COURT ROAD <-> LEICESTER SQUARE
GRANGE HILL <-> HAINAULT
WATERLOO <-> KENNINGTON
HYDE PARK CORNER <-> KNIGHTSBRIDGE
SOUTHWARK <-> LONDON BRIDGE
KING'S CROSS ST. PANCRAS <-> EUSTON
STRATFORD <-> MILE END
MARBLE ARCH <-> LANCASTER GATE
NORTH GREENWICH <-> CANNING TOWN
EARL'S COURT <-> BARONS COURT
LIVERPOOL STREET <-> ALDGATE
WEMBLEY PARK <-> FINCHLEY ROAD
TURNHAM GREEN <-> ACTON TOWN
LAMBETH NORTH <-> ELEPHANT & CASTLE
HIGH STREET KENSINGTON <-> GLOUCESTER ROAD
ANGEL <-> OLD STREET
PICCADILLY CIRCUS <-> CHARING CROSS
EUSTON SQUARE <-> GREAT PORTLAND STREET
HOLBORN CENTRAL <-> COVENT GARDEN
ST. JAMES'S PARK <-> WESTMINSTER
PICCADILLY CIRCUS <-> GREEN PARK
BOND STREET <-> GREEN PARK
OXFORD CIRCUS <-> BOND STREET
HAMMERSMITH <-> TURNHAM GREEN
STEPNEY GREEN <-> WHITECHAPEL

Northern Line:
WATERLOO <-> KENNINGTON
TOTTENHAM COURT ROAD <-> LEICESTER SQUARE
ANGEL <-> OLD STREET

Piccadilly Line:
EARL'S COURT <-> BARONS COURT
HOLBORN CENTRAL <-> COVENT GARDEN
PICCADILLY CIRCUS <-> GREEN PARK
HAMMERSMITH <-> TURNHAM GREEN
TURNHAM GREEN <-> ACTON TOWN
HYDE PARK CORNER <-> KNIGHTSBRIDGE

Victoria Line:
KING'S CROSS ST. PANCRAS <-> EUSTON

Waterloo & City Line:
```

Figure 30 Adjacent Station Closed - List No. 2

Figures 29 and 30 above illustrate the entire list of all the stations closed after running the program with Kruskal's algorithm implemented.

4. Final Remarks and Limitations

To find the minimum spanning tree of our connected and undirected graph, Kruskal's algorithm was one of the options due to its effectiveness and because it runs based on sorting the edges by their weights, making it suitable for this specific graph for the reason that it is a connected graph. Although the option was to apply Kruskal's algorithm for finding the minimum spanning tree, the limitations are mostly in terms of its time complexity when compared to Prim's algorithm which is lower than Kruskal's.

4b.

Impacts of the closure of adjacent stations:

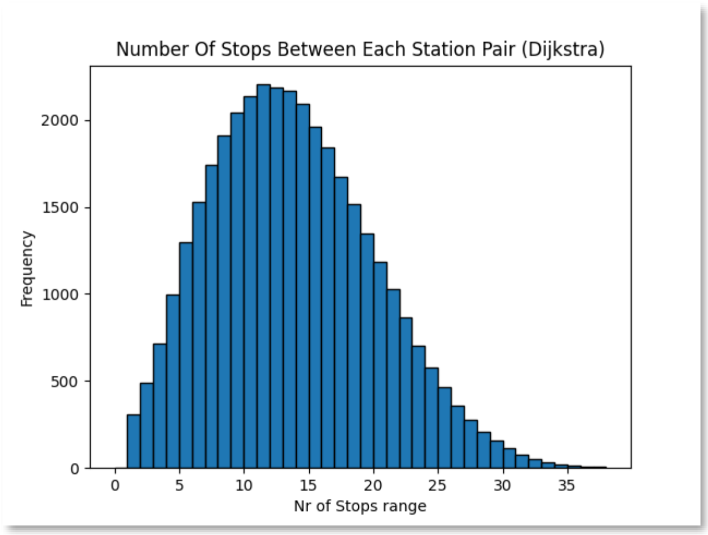


Figure 31 Number of stops before closure.

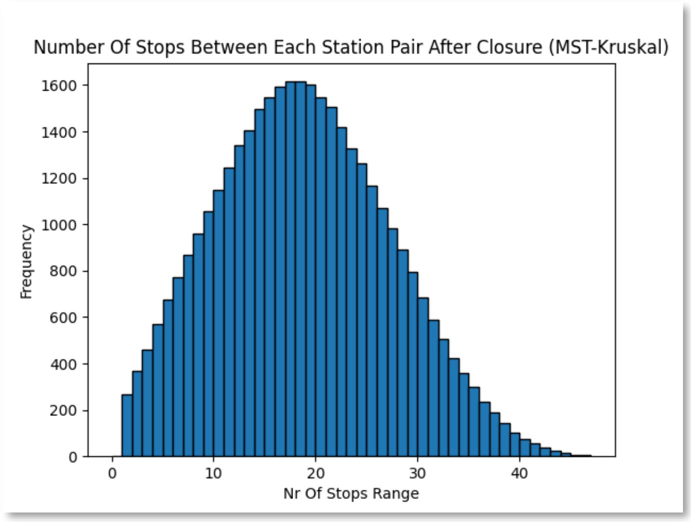
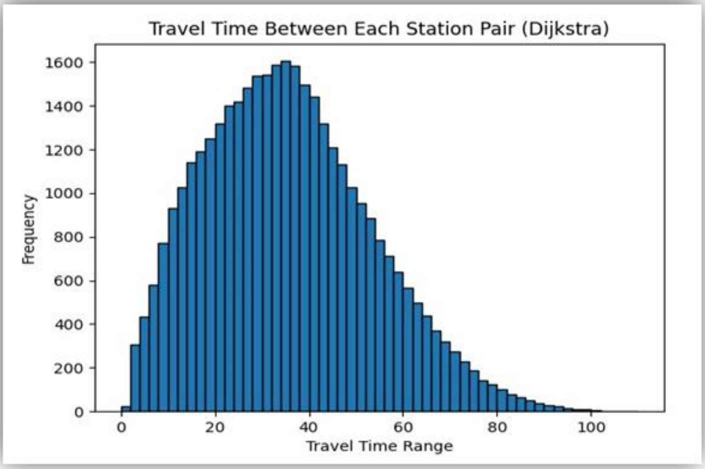


Figure 32 Number of stops after closure.

From Figures 31 and 32 above we can conclude that when we run the software using Dijkstra's algorithm to find the number of stops between each station pair and Kruskal's algorithm to create a minimum spanning tree, we can find the different frequency distribution. The two examples above illustrate the range of station pairs between 0 and 40. The first figure shows that when we execute the program using only Dijkstra's algorithm, the frequency of the number of stops was centred more between station pairs 5 and 20. In the second figure, we can determine that, after the execution of Kruskal's algorithm the frequency of the number of stops was more centred between stations pairs 3 and 30. It can also be seen that the frequency of the number of stops decreases after the closure of station pairs. Therefore, we can determine the impact of Kruskal's algorithm on the connection between stations of any two given stations. This also demonstrates that after we apply Kruskal's algorithm, the quantity of stops has substantially expanded. Lastly, the chart illustration, emphasizes the algorithm effect on the connectivity model for the total number of stops between each station pair from the London underground network graph.



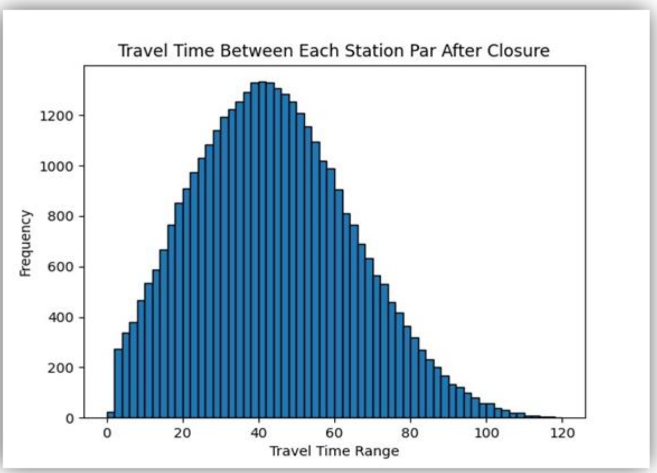


Figure 33 Travel time Histogram

Figure 34 Travel time after closure

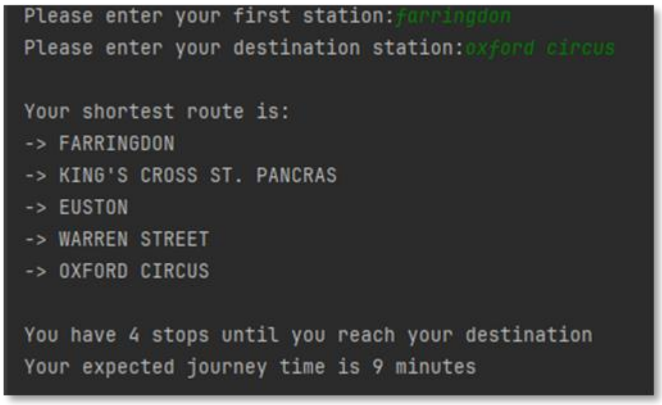
Figures 33 and 34 above show an overview of the effect of shutting the adjacent stations before and after closure and it can clearly be seen that there is a higher frequency before the closure. It can also be seen that travel time ranges differently with a slight change after closure with the bins ranging from 0 to 100 before closure and from 0 to 120 after closure indicating that the time also increases when all the adjacent station pairs are closed.

Testing for correctness and efficiency

Case	Test case	Inputs	Expected output	Actual Output	Pass/ Fail	Corrective Action
1	Retrieving the shortest path between two given stations before and after closure with MST confirmed the precision of the output from task 1a and task 4a	**Starting: Regent's** Park **Destination**: King's Cross St. Pancras	Your shortest route is: -> REGENT'S PARK -> OXFORD CIRCUS -> WARREN STREET -> EUSTON -> KING'S CROSS ST. PANCRAS Your shortest route is: -> REGENT'S PARK -> OXFORD CIRCUS -> TOTTENHAM -> HOLBORN -> CHANCERY LANE -> ST. PAUL'S -> BANK -> LIVERPOOL STREET -> MOORGATE -> BARBICAN -> FARRINGDON -> KING'S CROSS ST. PANCRAS	Your shortest route is: -> REGENT'S PARK -> OXFORD CIRCUS -> WARREN STREET -> EUSTON -> KING'S CROSS ST. PANCRAS Your shortest route is: -> REGENT'S PARK -> OXFORD CIRCUS -> TOTTENHAM -> HOLBORN -> CHANCERY LANE -> ST. PAUL'S -> BANK -> LIVERPOOL STREET -> MOORGATE -> BARBICAN -> FARRINGDON -> KING'S CROSS ST. PANCRAS	PASS	
2	Verified the accuracy of the output for task 1 and 2 where we have to get the shortest path between a specified starting and destination station	**Starting:** Warren Street **Destination**: Charing Cross	Your shortest route is: -> WARREN STREET -> OXFORD CIRCUS -> PICCADILLY CIRCUS -> CHARING CROSS	Your shortest route is: -> WARREN STREET -> GOODGE STREET -> TOTTENHAM COURT ROAD -> LEICESTER SQUARE -> CHARING CROSS	FAILED at first than we fixed the bug and PASS the test	As a fix, the dataset was filtered to only include rows with non-missing values in the "From Station" and "Connection" columns. These columns were then standardised by removing any extra spaces and changing them to uppercase, which made sure that the data was correct and consistent.

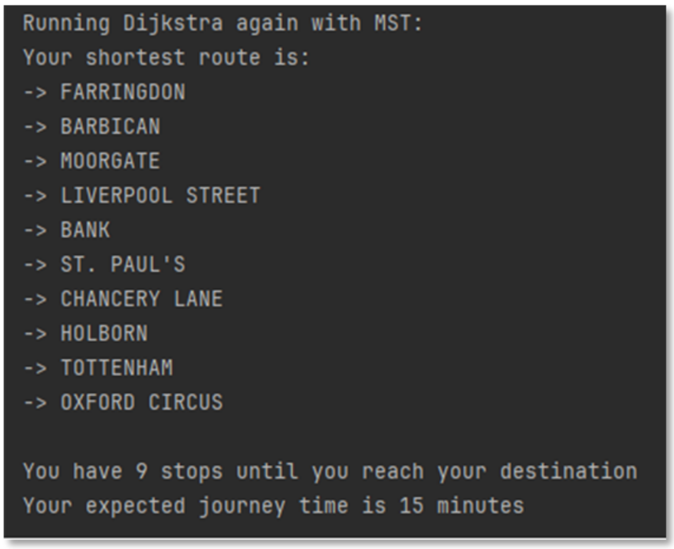
Test cases:

Test 1: Journey Itinerary and Duration



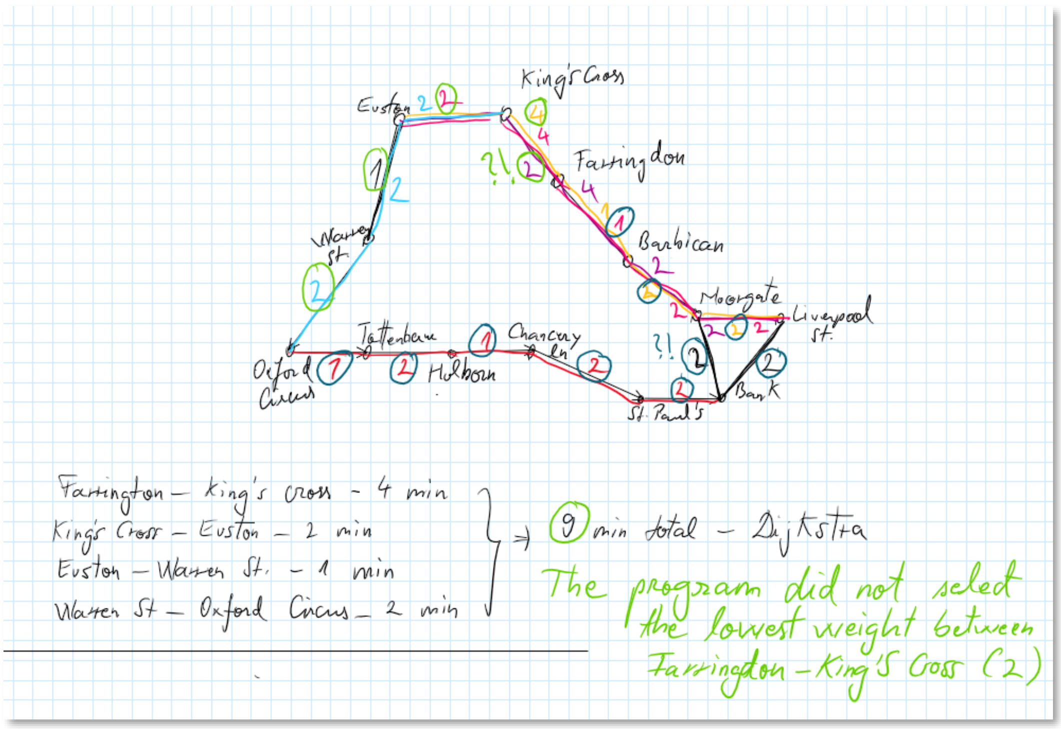
```
Please enter your first station:farringdon
Please enter your destination station:oxford circus

Your shortest route is:
-> FARRINGDON
-> KING'S CROSS ST. PANCRAS
-> EUSTON
-> WARREN STREET
-> OXFORD CIRCUS

You have 4 stops until you reach your destination
Your expected journey time is 9 minutes
```

Figure 35 - Shortest Route from Farringdon to Oxford Circus using Dijkstra.



```
Running Dijkstra again with MST:
Your shortest route is:
-> FARRINGDON
-> BARBICAN
-> MOORGATE
-> LIVERPOOL STREET
-> BANK
-> ST. PAUL'S
-> CHANCERY LANE
-> HOLBORN
-> TOTTENHAM
-> OXFORD CIRCUS

You have 9 stops until you reach your destination
Your expected journey time is 15 minutes
```

Figure 36 - Shortest Route from Farringdon to Oxford Circus using Dijkstra after applying Kruskal's algorithm.

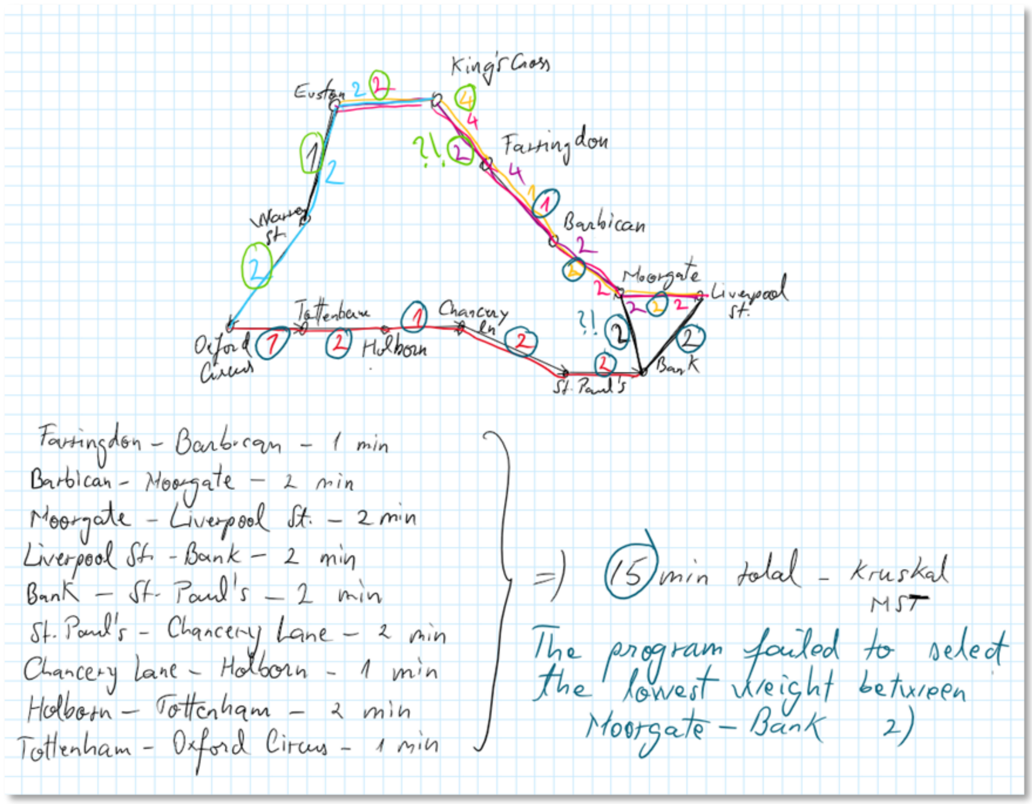


Farrington – King's Cross – 4 min
King's Cross – Euston – 2 min
Euston – Warren St. – 1 min
Warren St – Oxford Circus – 2 min

⇒ 9 min total – Dijkstra

The program did not select the lowest weight between Farringdon – King's Cross (2)

Figure 37 - Test Case against the resulting output of the program for the shortest route between Farringdon and Oxford Circus using Dijkstra.



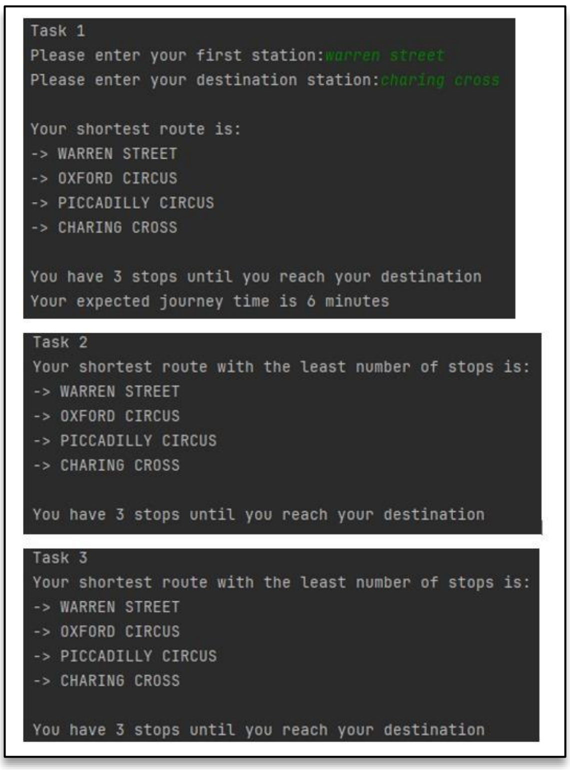
Farringdon – Barbican – 1 min
Barbican – Moorgate – 2 min
Moorgate – Liverpool St. – 2 min
Liverpool St. – Bank – 2 min
Bank – St. Paul's – 2 min
St. Paul's – Chancery Lane – 2 min
Chancery Lane – Holborn – 1 min
Holborn – Tottenham – 2 min
Tottenham – Oxford Circus – 1 min

⇒ 15 min total – kruskal MST

The program failed to select the lowest weight between Moorgate – Bank 2)

Figure 38 - Test Case against the resulting output of the program for the shortest route between Farringdon and Oxford Circus running Dijkstra with MST

Test 2: Accuracy of task 1 and 2/3

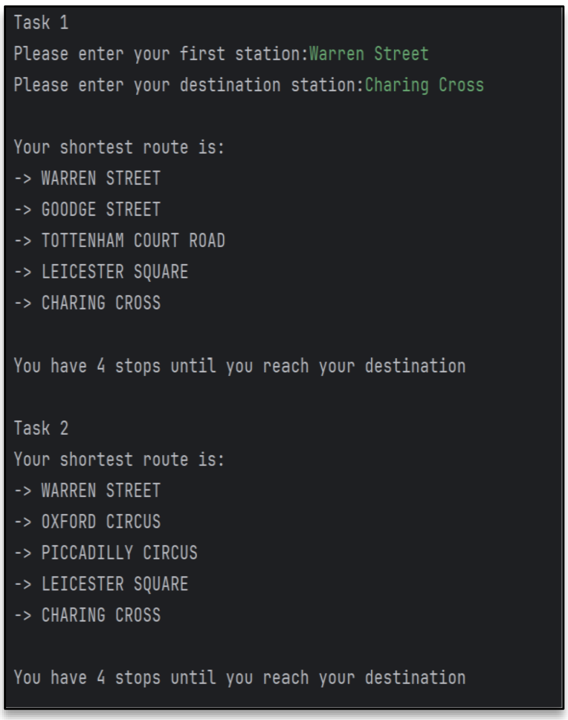


```
Task 1
Please enter your first station:warren street
Please enter your destination station:charing cross

Your shortest route is:
-> WARREN STREET
-> OXFORD CIRCUS
-> PICCADILLY CIRCUS
-> CHARING CROSS

You have 3 stops until you reach your destination
Your expected journey time is 6 minutes
```

```
Task 2
Your shortest route with the least number of stops is:
-> WARREN STREET
-> OXFORD CIRCUS
-> PICCADILLY CIRCUS
-> CHARING CROSS

You have 3 stops until you reach your destination
```

```
Task 3
Your shortest route with the least number of stops is:
-> WARREN STREET
-> OXFORD CIRCUS
-> PICCADILLY CIRCUS
-> CHARING CROSS

You have 3 stops until you reach your destination
```

Figure 39 - Output of the code with the bug



```
Task 1
Please enter your first station:Warren Street
Please enter your destination station:Charing Cross

Your shortest route is:
-> WARREN STREET
-> GOODGE STREET
-> TOTTENHAM COURT ROAD
-> LEICESTER SQUARE
-> CHARING CROSS

You have 4 stops until you reach your destination

Task 2
Your shortest route is:
-> WARREN STREET
-> OXFORD CIRCUS
-> PICCADILLY CIRCUS
-> LEICESTER SQUARE
-> CHARING CROSS

You have 4 stops until you reach your destination
```

Figure 40 - Output of the testing of the code after sorting out the bug

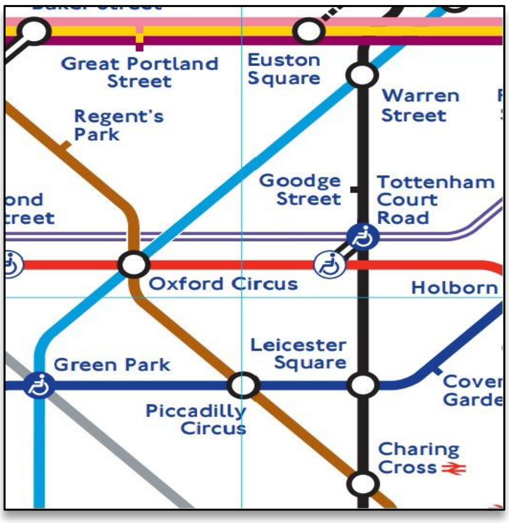


Figure 41 - Section of the Map

During the evaluation phase of the software, it has come to our attention that the Dijkstra algorithm failed to produce accurate path results when determining the route from the originating station to the destination. The discrepancy results from the algorithm's incapacity to produce the right answer as reported by the dataset, particularly with regard to the number of stations passed by, as the sample map makes clear.

Looking more closely at the sample of the map, we can see that Piccadilly Circus and Charing Cross stations are directly adjacent. Thus, choosing the edge that goes from Leicester Square to Charing Cross and then the edge that goes from Leicester Square to Charing Cross turns out to be an unnecessary path. This departure from the anticipated result emphasises how the algorithm fails to correctly calculate the shortest path based on the number of stations, which is an essential component of the current assignment.

As a fix, the dataset was filtered to only include rows with non-missing values in the "From Station" and "Connection" columns. These columns were then standardised by removing any extra spaces and changing them to uppercase, which made sure that the data was correct and consistent. As result we get the right path and accurate path.

Conclusion

In conclusion, this comprehensive report successfully achieved its objective of creating an efficient software model for the London Underground's route planner. By employing advanced algorithms and data structures, the team efficiently handled various tasks, such as journey planning, station count evaluation, algorithm comparison, and tube line closure analysis. The use of the Adjacency List proved effective for representing the network's sparse graph structure, and Dijkstra's algorithm was pivotal for journey itinerary and duration analysis. The report demonstrates the effectiveness of the chosen methods through empirical analysis, reflecting a commitment to robust implementation and rigorous testing. The team's collaborative approach and problem-solving skills were crucial in overcoming challenges and delivering a software model that enhances travel planning efficiency for the public.

Teamwork and Communication

Participants

Student	*ID*	*contribution*
Daniela E Vilcea		20
Silvester Daniel Dan		20

Eusebiu Tamas	20
Yordan Genchov	20
Mohamed T Zedan	20
	100

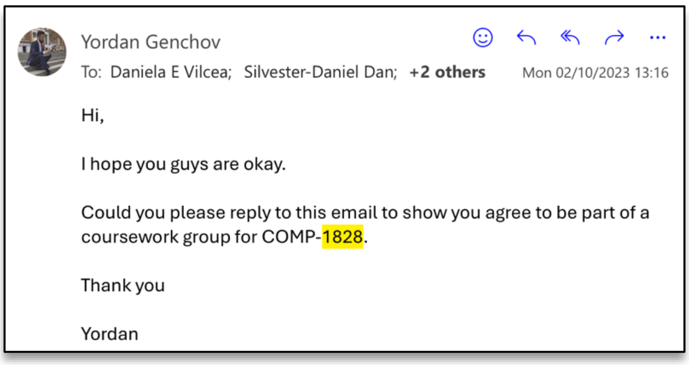
Meeting Number	Date	Milestones, tasks
1.	02/10/2023	• **Group creation** - Forming our team, setting up roles, and organizing how working together will be, considering team members schedules. • **Discussion of the requirements** – Coursework specifications in details, getting familiar with all guidelines, making sure we're clear on what's needed. • **Team introductions** - Getting to know each other, sharing skills, and understanding how we can collaborate effectively.
2.	16/10/2023	• **Role assignment** - Assign roles based on strengths and interests within the team. • **Task Breakdown** - Divide the coursework into manageable parts for each team member. • **Brainstorming Approaches and Strategies** - Explore different ways to tackle the project, sharing ideas, and planning our data structure approach.
3.	24/10/2023	• **Software implementation** on Task 1a. Graph creation - Work on building the graph. Discussion about and implementation. • **Dijkstra's algorithm** – Focus on implementing and understanding Dijkstra's algorithm and how it will work with our graph. • **Task 3a Discussion and planning**
4.	31/10/2023	• **Progress Discussion** - Reviewing progress and debugging code, specifically modifying histograms for Task 1 and 2. • **Task 3a main discussion** - understanding Bellman-Ford algorithm and how it can help us solve the task. • **Task 4** – General planning and reviewing Task 4 specifications with solution proposal
5.	06/11/2023 Reading week meeting	• **Task 4** - Discussing implementation of minimum spanning tree algorithm and reviewing the code checking for potential issues and bugs. • **Discussion on histograms** - Comprehensive discussion about histograms for all completed tasks so far and analysing their implementation, comparing results, and ensuring consistency.
6.	13/11/2023	• **Progress so far** - Review progress to date and edit report layout for better clarity and coherence by restructuring sections, reordering content, and ensuring a more seamless flow. • **Discussion on necessary adjustments and additions** - Identifying areas needing refinement in the report and discussing additional information or sections that could enhance its quality. • **Adding additional information and more comparison on histograms** – edit the report by including extra details and further comparative analysis of histograms. This includes expanding on existing information, providing deeper insights, and comparing different aspects of histograms. • **Tutor queries about test cases** - clarifying our testing methodology, presenting results from carried out tests, and ensuring our test cases align with the project requirements.
7.	20/11/2023	• **Progress discussion** - Progress made since our last meeting and addressing any issues or bugs encountered during the project. This involves identifying challenges faced, solutions implemented, and any ongoing issues that need attention. • **Kruskal's Algorithm implementation** - Comparing Kruskal's Algorithm implementation using two different data structures: Adjacency List Graph and Adjacency Matrix by analysing their efficiency, advantages, and drawbacks when implemented in our project code. • **Time and space complexity** - Examining the time and space complexity of implementation of Kruskal's Algorithm. This analysis involves evaluating the algorithm's efficiency in terms of computational time required and memory usage for each data structure used (Adjacency List Graph and Adjacency Matrix).
	27/11/2023 Submission	• **Final remarks** - Summarize the project with final remarks, highlighting the strengths and acknowledging limitations or areas where improvements could be made.

8.	Day	• **Paper and pen test cases** - manual test cases using paper and pen to validate the functionality and logic of our code simulating inputs, running through algorithms manually, and verifying expected outputs against our code's results. • **Final check of references**

Weekly communication documentation

Further insights into our team's communication throughout the project can be found in the detailed screenshots provided in the appendix.

Our team primarily utilized email for organizing and scheduling regular meetings, ensuring everyone was on the same page regarding their availability and agenda topics. Additionally, we employed a WhatsApp group chat to maintain constant communication throughout the week, enabling team members to update their progress on assigned tasks promptly.



Yordan Genchov
To: Daniela E Vilcea; Silvester-Daniel Dan; +2 others
Mon 02/10/2023 13:16

Hi,

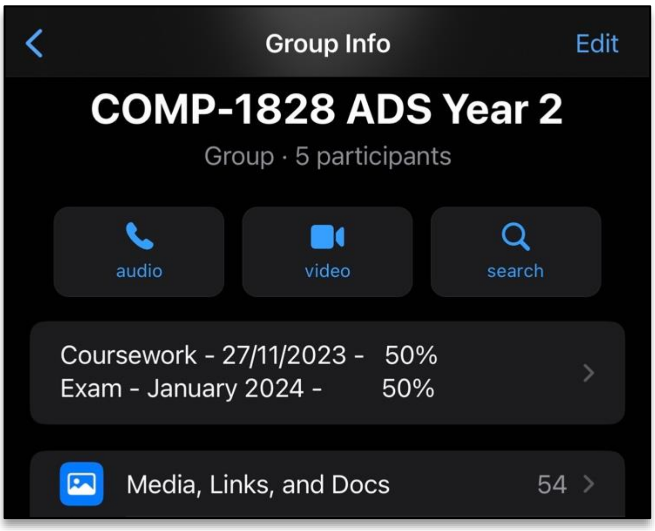
I hope you guys are okay.

Could you please reply to this email to show you agree to be part of a coursework group for COMP-1828.

Thank you

Yordan

Figure 42 Sample of email communication.



Group Info

COMP-1828 ADS Year 2

Group · 5 participants

Coursework - 27/11/2023 - 50%
Exam - January 2024 - 50%

Figure 43 WhatsApp group formation

Reflection and critical discussion

The implementation of Dijkstra's algorithm for pathfinding within the London Underground network exhibited remarkable efficiency. Its tailored approach to finding the shortest paths between stations, coupled with the suitability of the AdjacencyListGraph representation, underscored its effectiveness in handling sparse graph structures.

Conversely, Kruskal’s algorithm, while demonstrating a higher time complexity, proved adept at constructing minimum-spanning trees within the connected graph framework. The use of Kruskal's algorithm showcased its ability to efficiently navigate graph connections, contributing to the development of a comprehensive minimum-spanning tree for the London Underground network.

These algorithmic approaches—Dijkstra's for pathfinding and Kruskal’s for spanning tree construction—exemplify the significance of algorithm selection tailored to specific graph characteristics. Despite the higher time complexity of Kruskal’s, its utility within a connected graph context remained evident, emphasizing the importance of aligning algorithmic choices with the unique attributes of the graph in question for optimal solutions.

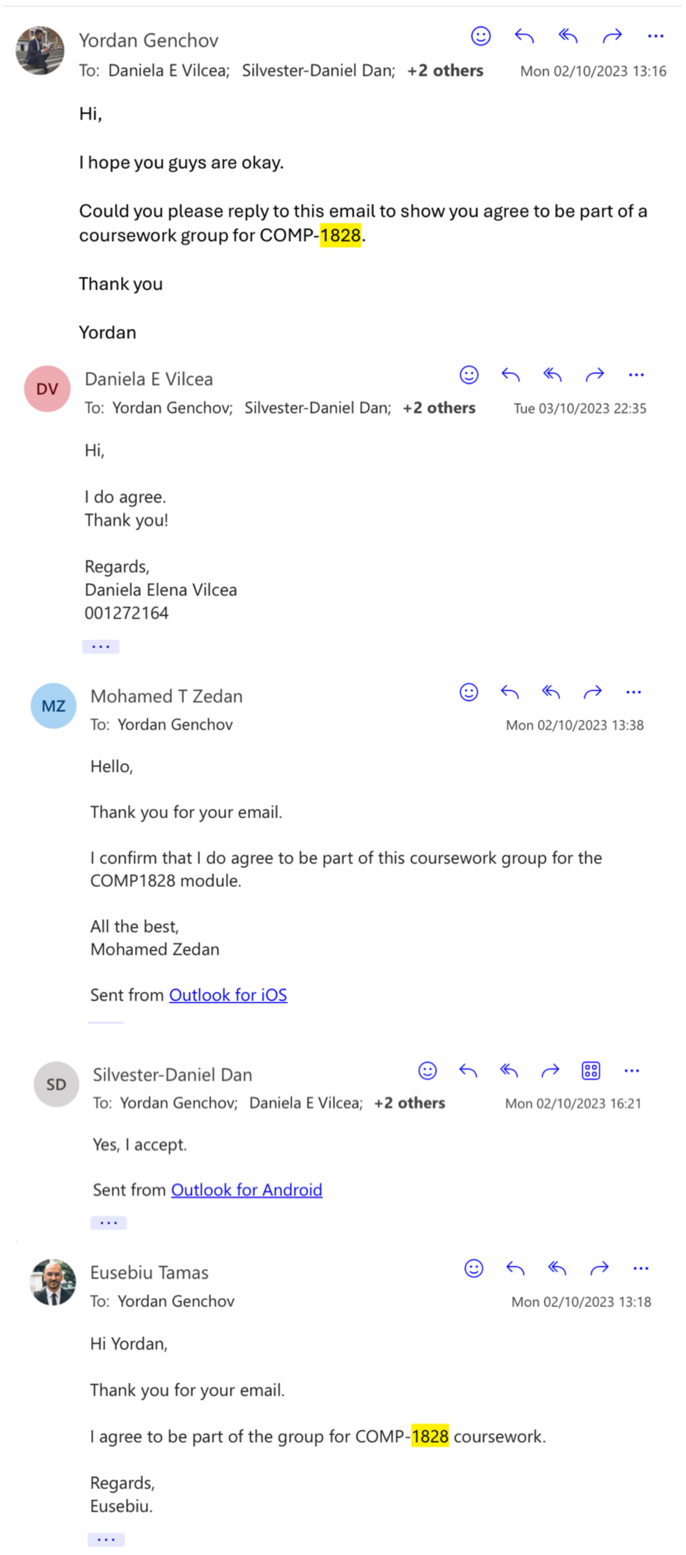
References

Cormen, T.H. *et al*. (2022) ‘19 Data Structures for Disjoint Sets’, in *Introduction to algorithms*. Cambridge, MA: The MIT Press.

Appendices

Additional Documentation

Weekly email communication 27/11/2023



Yordan Genchov
To: Daniela E Vilcea; Silvester-Daniel Dan; **+2 others** Mon 02/10/2023 13:16

Hi,

I hope you guys are okay.

Could you please reply to this email to show you agree to be part of a coursework group for COMP-1828.

Thank you

Yordan

Daniela E Vilcea
To: Yordan Genchov; Silvester-Daniel Dan; **+2 others** Tue 03/10/2023 22:35

Hi,

I do agree.
Thank you!

Regards,
Daniela Elena Vilcea
001272164

Mohamed T Zedan
To: Yordan Genchov Mon 02/10/2023 13:38

Hello,

Thank you for your email.

I confirm that I do agree to be part of this coursework group for the COMP1828 module.

All the best,
Mohamed Zedan

Sent from Outlook for iOS

Silvester-Daniel Dan
To: Yordan Genchov; Daniela E Vilcea; **+2 others** Mon 02/10/2023 16:21

Yes, I accept.

Sent from Outlook for Android

Eusebiu Tamas
To: Yordan Genchov Mon 02/10/2023 13:18

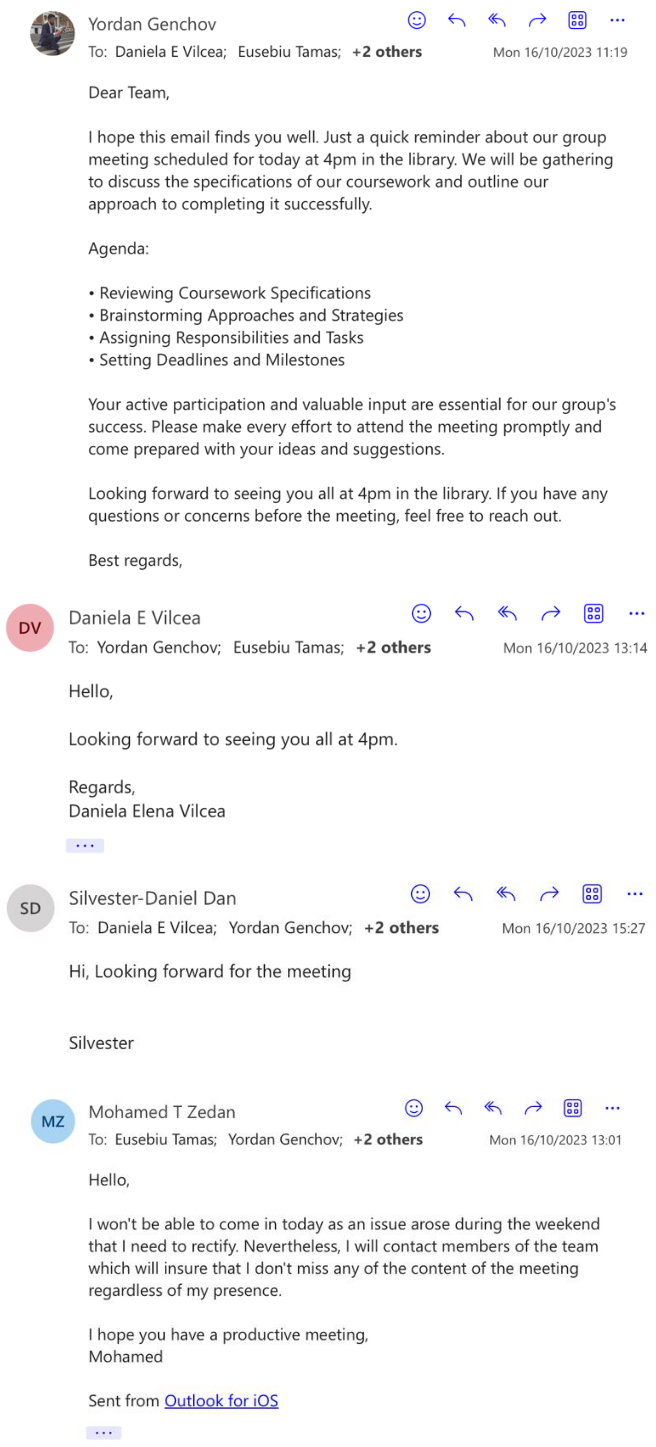
Hi Yordan,

Thank you for your email.

I agree to be part of the group for COMP-1828 coursework.

Regards,
Eusebiu.

Weekly email communication 16/10/2023



Yordan Genchov
To: Daniela E Vilcea; Eusebiu Tamas; **+2 others**
Mon 16/10/2023 11:19

Dear Team,

I hope this email finds you well. Just a quick reminder about our group meeting scheduled for today at 4pm in the library. We will be gathering to discuss the specifications of our coursework and outline our approach to completing it successfully.

Agenda:

• Reviewing Coursework Specifications
• Brainstorming Approaches and Strategies
• Assigning Responsibilities and Tasks
• Setting Deadlines and Milestones

Your active participation and valuable input are essential for our group's success. Please make every effort to attend the meeting promptly and come prepared with your ideas and suggestions.

Looking forward to seeing you all at 4pm in the library. If you have any questions or concerns before the meeting, feel free to reach out.

Best regards,

DV Daniela E Vilcea
To: Yordan Genchov; Eusebiu Tamas; **+2 others**
Mon 16/10/2023 13:14

Hello,

Looking forward to seeing you all at 4pm.

Regards,
Daniela Elena Vilcea

SD Silvester-Daniel Dan
To: Daniela E Vilcea; Yordan Genchov; **+2 others**
Mon 16/10/2023 15:27

Hi, Looking forward for the meeting

Silvester

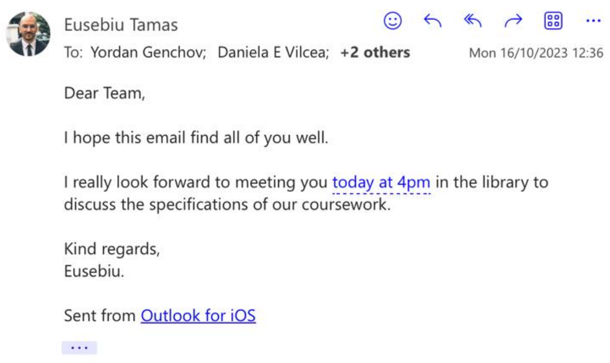
MZ Mohamed T Zedan
To: Eusebiu Tamas; Yordan Genchov; **+2 others**
Mon 16/10/2023 13:01

Hello,

I won't be able to come in today as an issue arose during the weekend that I need to rectify. Nevertheless, I will contact members of the team which will insure that I don't miss any of the content of the meeting regardless of my presence.

I hope you have a productive meeting,
Mohamed

Sent from Outlook for iOS



Eusebiu Tamas
To: Yordan Genchov; Daniela E Vilcea; **+2 others** Mon 16/10/2023 12:36

Dear Team,

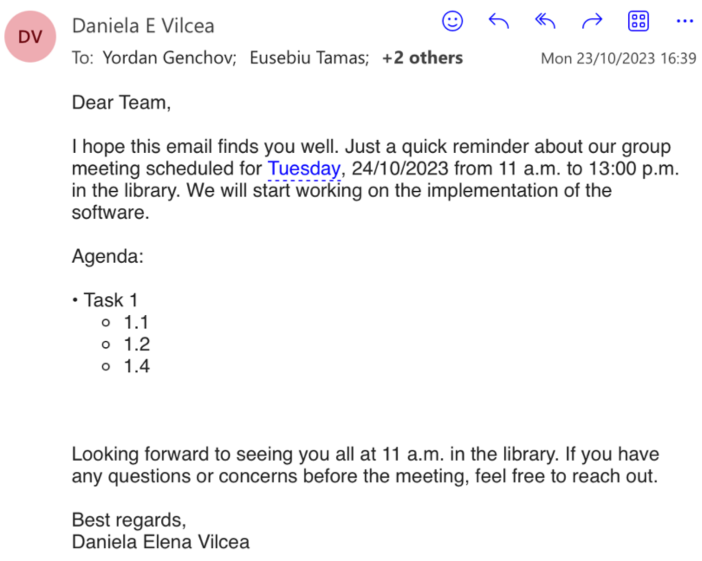
I hope this email find all of you well.

I really look forward to meeting you today at 4pm in the library to discuss the specifications of our coursework.

Kind regards,
Eusebiu.

Sent from Outlook for iOS

Weekly email communication 21/10/2023



Daniela E Vilcea
To: Yordan Genchov; Eusebiu Tamas; **+2 others** Mon 23/10/2023 16:39

Dear Team,

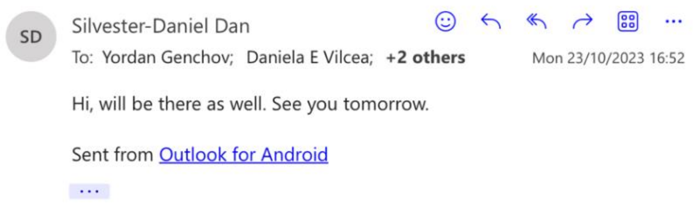
I hope this email finds you well. Just a quick reminder about our group meeting scheduled for Tuesday, 24/10/2023 from 11 a.m. to 13:00 p.m. in the library. We will start working on the implementation of the software.

Agenda:

- Task 1
  - 1.1
  - 1.2
  - 1.4

Looking forward to seeing you all at 11 a.m. in the library. If you have any questions or concerns before the meeting, feel free to reach out.

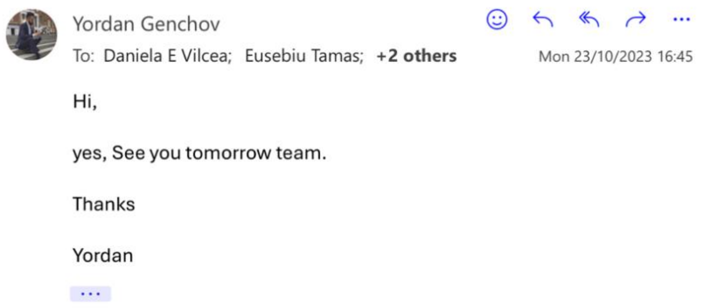
Best regards,
Daniela Elena Vilcea



Silvester-Daniel Dan
To: Yordan Genchov; Daniela E Vilcea; **+2 others** Mon 23/10/2023 16:52

Hi, will be there as well. See you tomorrow.

Sent from Outlook for Android



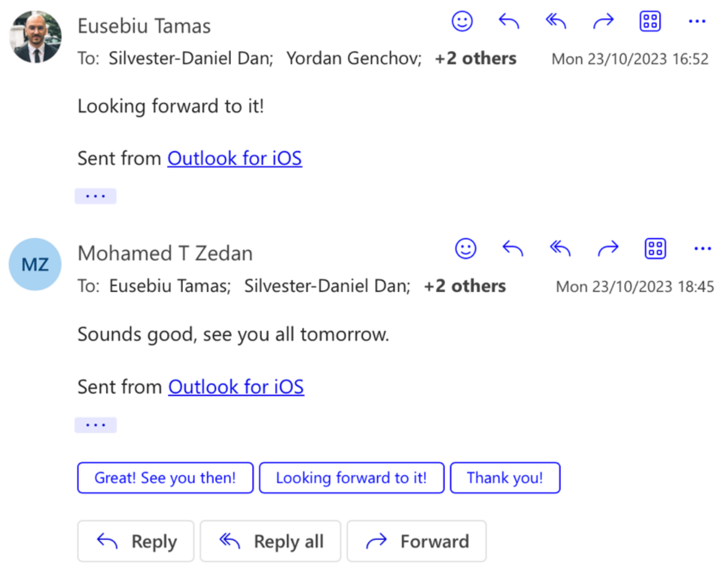
Yordan Genchov
To: Daniela E Vilcea; Eusebiu Tamas; **+2 others** Mon 23/10/2023 16:45

Hi,

yes, See you tomorrow team.

Thanks

Yordan


Eusebiu Tamas
To: Silvester-Daniel Dan; Yordan Genchov; **+2 others** Mon 23/10/2023 16:52

Looking forward to it!

Sent from Outlook for iOS

...

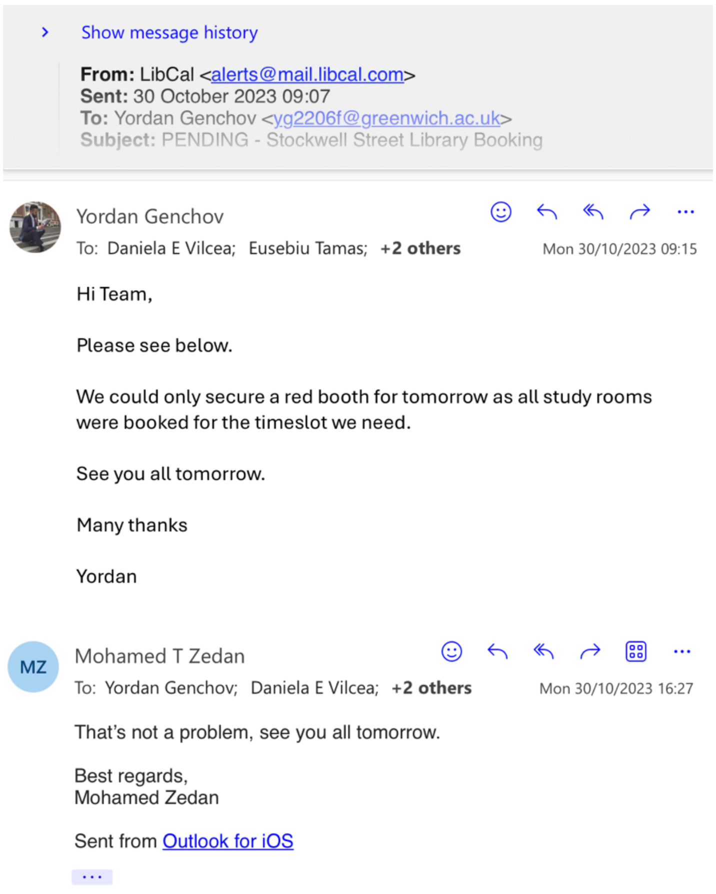
Mohamed T Zedan
To: Eusebiu Tamas; Silvester-Daniel Dan; **+2 others** Mon 23/10/2023 18:45

Sounds good, see you all tomorrow.

Sent from Outlook for iOS

...

Weekly email communication 31/10/2023



Show message history

From: LibCal <alerts@mail.libcal.com>
Sent: 30 October 2023 09:07
To: Yordan Genchov <yg2206f@greenwich.ac.uk>
Subject: PENDING - Stockwell Street Library Booking

Yordan Genchov
To: Daniela E Vilcea; Eusebiu Tamas; **+2 others** Mon 30/10/2023 09:15

Hi Team,

Please see below.

We could only secure a red booth for tomorrow as all study rooms were booked for the timeslot we need.

See you all tomorrow.

Many thanks

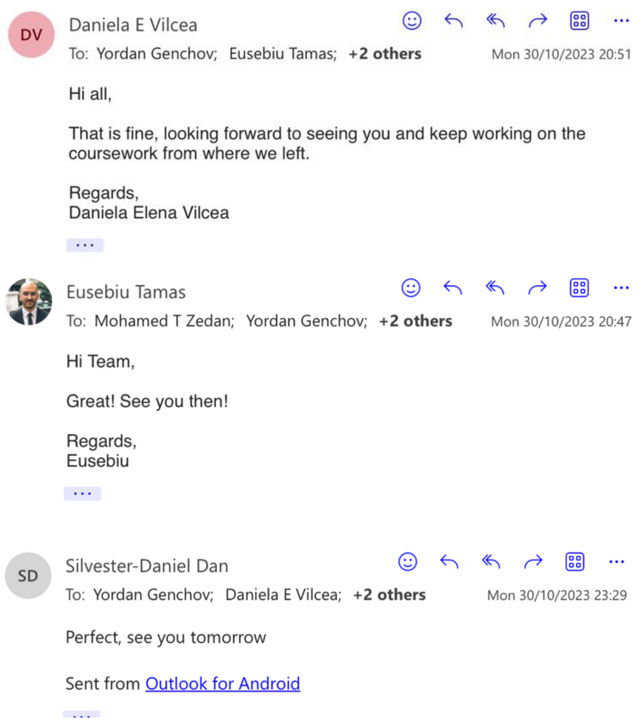
Yordan

Mohamed T Zedan
To: Yordan Genchov; Daniela E Vilcea; **+2 others** Mon 30/10/2023 16:27

That's not a problem, see you all tomorrow.

Best regards,
Mohamed Zedan

Sent from Outlook for iOS



Daniela E Vilcea
To: Yordan Genchov; Eusebiu Tamas; **+2 others** Mon 30/10/2023 20:51

Hi all,

That is fine, looking forward to seeing you and keep working on the coursework from where we left.

Regards,
Daniela Elena Vilcea

Eusebiu Tamas
To: Mohamed T Zedan; Yordan Genchov; **+2 others** Mon 30/10/2023 20:47

Hi Team,

Great! See you then!

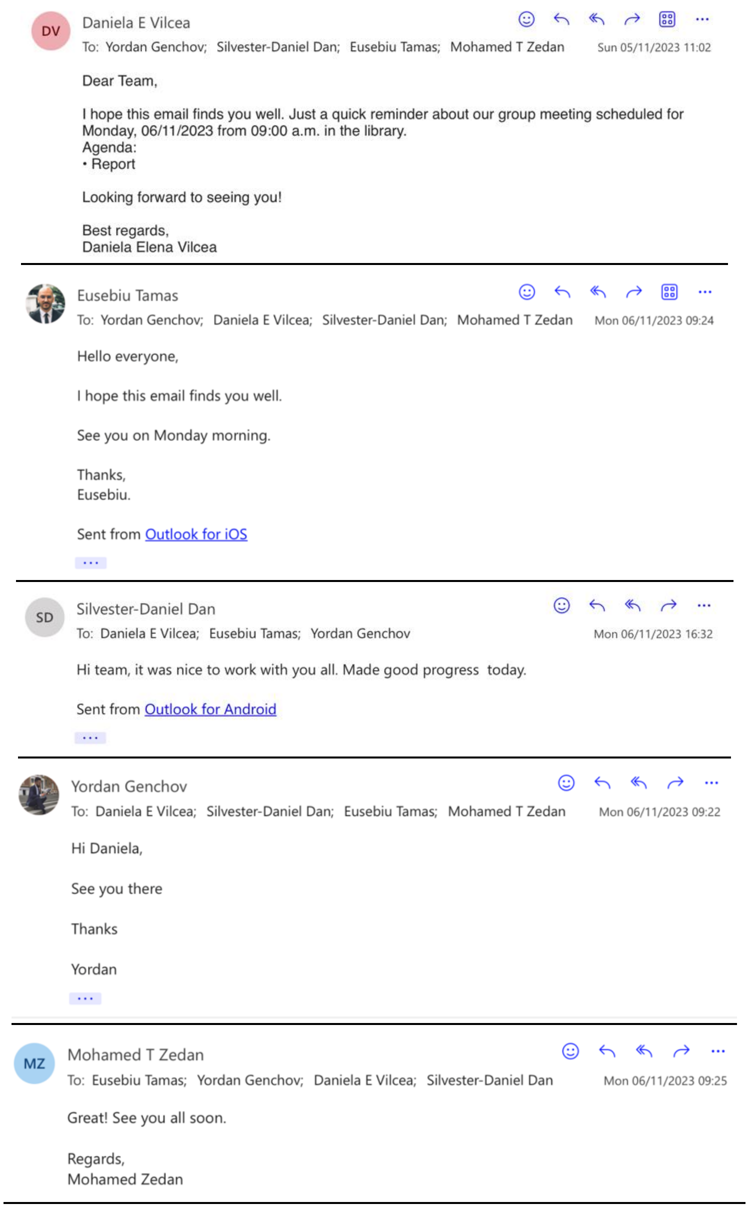
Regards,
Eusebiu

Silvester-Daniel Dan
To: Yordan Genchov; Daniela E Vilcea; **+2 others** Mon 30/10/2023 23:29

Perfect, see you tomorrow

Sent from Outlook for Android

Weekly email communication 06/11/2023



Daniela E Vilcea
To: Yordan Genchov; Silvester-Daniel Dan; Eusebiu Tamas; Mohamed T Zedan   Sun 05/11/2023 11:02

Dear Team,

I hope this email finds you well. Just a quick reminder about our group meeting scheduled for Monday, 06/11/2023 from 09:00 a.m. in the library.
Agenda:
• Report

Looking forward to seeing you!

Best regards,
Daniela Elena Vilcea

Eusebiu Tamas
To: Yordan Genchov; Daniela E Vilcea; Silvester-Daniel Dan; Mohamed T Zedan   Mon 06/11/2023 09:24

Hello everyone,

I hope this email finds you well.

See you on Monday morning.

Thanks,
Eusebiu.

Sent from Outlook for iOS

Silvester-Daniel Dan
To: Daniela E Vilcea; Eusebiu Tamas; Yordan Genchov   Mon 06/11/2023 16:32

Hi team, it was nice to work with you all. Made good progress today.

Sent from Outlook for Android

Yordan Genchov
To: Daniela E Vilcea; Silvester-Daniel Dan; Eusebiu Tamas; Mohamed T Zedan   Mon 06/11/2023 09:22

Hi Daniela,

See you there

Thanks

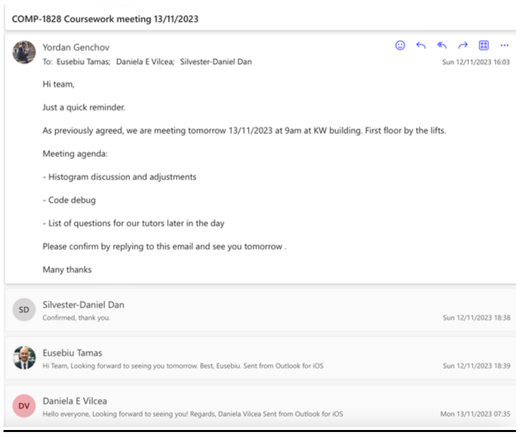
Yordan

Mohamed T Zedan
To: Eusebiu Tamas; Yordan Genchov; Daniela E Vilcea; Silvester-Daniel Dan   Mon 06/11/2023 09:25

Great! See you all soon.

Regards,
Mohamed Zedan

Weekly email communication 13/11/2023



COMP-1828 Coursework meeting 13/11/2023

Yordan Genchov
To: Eusebiu Tamas; Daniela E Vilcea; Silvester-Daniel Dan
Sun 12/11/2023 16:03

Hi team,

Just a quick reminder.

As previously agreed, we are meeting tomorrow 13/11/2023 at 9am at KW building. First floor by the lifts.

Meeting agenda:

- Histogram discussion and adjustments

- Code debug

- List of questions for our tutors later in the day

Please confirm by replying to this email and see you tomorrow .

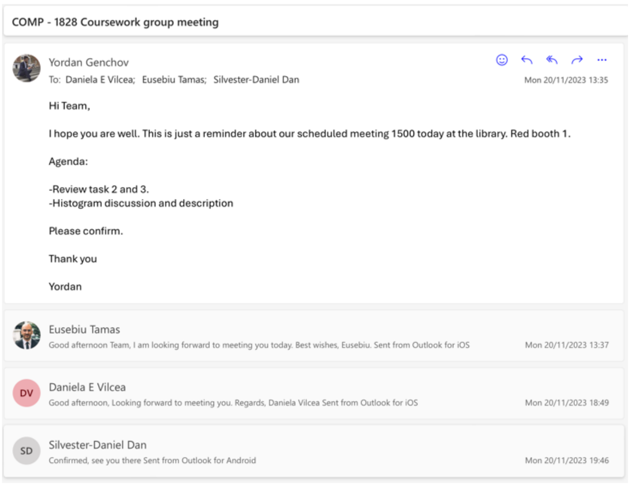
Many thanks

Silvester-Daniel Dan
Confirmed, thank you.
Sun 12/11/2023 18:38

Eusebiu Tamas
Hi Team, Looking forward to seeing you tomorrow. Best, Eusebiu. Sent from Outlook for iOS
Sun 12/11/2023 18:39

Daniela E Vilcea
Hello everyone, Looking forward to seeing you! Regards, Daniela Vilcea Sent from Outlook for iOS
Mon 13/11/2023 07:35

Weekly email communication 20/11/2023



COMP - 1828 Coursework group meeting

Yordan Genchov
To: Daniela E Vilcea; Eusebiu Tamas; Silvester-Daniel Dan
Mon 20/11/2023 13:35

Hi Team,

I hope you are well. This is just a reminder about our scheduled meeting 1500 today at the library. Red booth 1.

Agenda:

-Review task 2 and 3.
-Histogram discussion and description

Please confirm.

Thank you

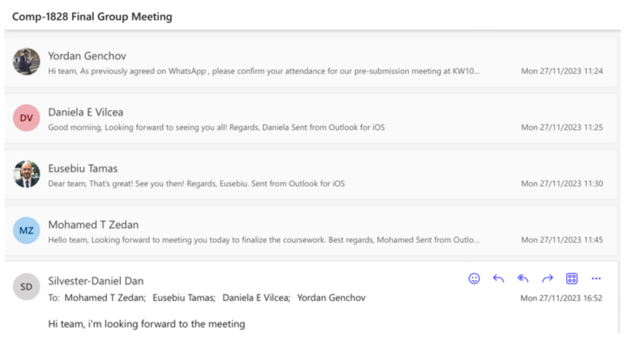
Yordan

Eusebiu Tamas
Good afternoon Team, I am looking forward to meeting you today. Best wishes, Eusebiu. Sent from Outlook for iOS
Mon 20/11/2023 13:37

Daniela E Vilcea
Good afternoon, Looking forward to meeting you. Regards, Daniela Vilcea Sent from Outlook for iOS
Mon 20/11/2023 18:49

Silvester-Daniel Dan
Confirmed, see you there Sent from Outlook for Android
Mon 20/11/2023 19:46

Weekly email communication 27/11/2023



Comp-1828 Final Group Meeting

Yordan Genchov
Hi team, As previously agreed on WhatsApp , please confirm your attendance for our pre-submission meeting at KW10...
Mon 27/11/2023 11:24

Daniela E Vilcea
Good morning, Looking forward to seeing you all! Regards, Daniela Sent from Outlook for iOS
Mon 27/11/2023 11:25

Eusebiu Tamas
Dear team, That's great! See you then! Regards, Eusebiu. Sent from Outlook for iOS
Mon 27/11/2023 11:30

Mohamed T Zedan
Hello team, Looking forward to meeting you today to finalize the coursework. Best regards, Mohamed Sent from Outlo...
Mon 27/11/2023 11:45

Silvester-Daniel Dan
To: Mohamed T Zedan; Eusebiu Tamas; Daniela E Vilcea; Yordan Genchov
Mon 27/11/2023 16:52

Hi team, i'm looking forward to the meeting