COMP 1828
DESIGNING, DEVELOPING AND TESTING SOLUTIONS FOR LONDON UNDERGROUND SYSTEM

GROUP 2

CHISEL CHAVUSH	:	(001131628)
BARIS CELIKTUTAN	:	(001120868)
ARDA SAHAN	:	(001150659)
OZAN GOKBERK	:	(001151346)

1) ***JUSTIFICATION OF THE CHOICE OF THE DATA STRUCTURES AND ALGORITHMS:***

Dijkstra's Algorithm:

For the first task the algorithm used is Dijkstra's algorithm. This algorithm also known as the single-source shortest path algorithm. The algorithm works by finding the shortest path between two nodes in a graph. To use this method, the graph needs to be weighted. The minimum spanning tree is the result of this algorithm, it is a tree like structure that connects from the source node to all the other nodes in the graph whilst following the shortest path. It does this by using the weights of the edges to find the shortest path to the destination node. In this scenario the London underground stations represent the nodes and the time taken between each station is the weight. Therefore, this algorithm is a perfect choice for the given scenario as our objective is to find the shortest path between two London underground stations provided by the user. In this task, to implement Dijkstra's algorithm the existing library "networkx" was used.

Kruskal Algorithm:

We were asked to create a new software to meet the requirements of the government. In order to do that, we had to search new algorithm as we cannot process the same efficiency with Dijkstra's Algorithm. In this task, the system had to find another path while there are as many closures as possible. To complete this task, we start to search for an alternative algorithm which is Kruskal's Algorithm. This algorithm works by adding all the edges in increasing order of their weight and then adds the nodes to the tree only if it does not form any cycle in the tree. While the Kruskal algorithm does not form any cycle on the tree, that represent our closed immediate neighbour stations. Therefore, this algorithm was suitable for the needs of this task since a new graph was required to be created which contained these closures.

Linear Search Algorithm:

In this final section, we had to come up with a new task which also required a new algorithm to solve it. The final task that was decided by the group was to output all the lines that a stations in path was linked to, and to complete this, the linear search algorithm was used. Linear search algorithm works by comparing each item in a list to the target item until it finds a match, it starts from the beginning until the end of the list. For that reason, this algorithm was suitable for the stated task because we have multiple unordered lists where we are trying to search all the elements in the list to find all the lines connected to each station. Once the algorithm has found the train lines of each station, it can then output it to the user once they have inputted the two stations that they want to destinate and to arrive. In this task algorithm can display the path that user taking on task 1. Also providing when to change station to which line.

2) ***CRITICAL EVALUATION OF THE PERFORMANCE OF THE DATA STRUCTURES AND ALGORITHMS USED:***

To measure the performance, we decided to use the very common approach of Big O and using time and space complexity. Time complexity is determined by the computer measuring how long it takes to run the algorithm or multiple algorithms. The space complexity of an algorithm is determined by calculating how much memory space is required to solve an instance of the computational problem as the function of the characteristics of the input whether that be an array or user inputs. Simply it is just how much computer memory is required to run an algorithm till it executes completely.

Task 1 Dijkstra’s Algorithm Evaluation:

As stated previously for task 1 Dijkstra’s algorithm was used to find the shortest possible path. The Big O complexity of Dijkstra's Algorithm is **O(V^2)**. The space complexity of Dijkstra's Algorithm is **O(V)**. In both the time complexity and space complexity the V denotes the number of vertices or nodes in the graph that implements Dijkstra's Algorithm. **O(V^2)** is known as quadratic time. Quadratic time is a function that has a growth rate of **n^2**. Therefore, if the input size of the algorithm is 2, it will execute four operations. **O(n)** is space complexity is known as linear complexity. This means that the space that is taken up is directly proportionate to the input size of the algorithm.

The time taken for the system for Task 1a to display the information to the user was 0.0009973 seconds which shows how fast Dijkstra’s algorithm works to find the shortest path and weight of it, out of all the possible routes. It was important to also measure the time taken to generate the histogram in the first task. This is because, it shows us all the possible journey times and how long it took to generate the histogram, the data found is 67077 journeys and approximately 171.41 seconds respectively.

Task 2 Kruskal’s Algorithm Evaluation:

The time complexity for Kruskal’s algorithm is **O(E log V)**, this means that as the input size grows, the number of operations grows very slowly. The space complexity is **O(log(E))** takes space proportional to the log of the input size. Once again, the **“V”** represents the number of vertices in the graph also known as nodes, **“E“** represents the edges in the graph.

The time taken for the system to create the graph using Kruskal’s algorithm in our software was 0.15758 seconds. A histogram was also created for this task with the closures, the time taken to create the histogram was 81.39 seconds which was much quicker than the histogram from the previous task. This is due to the closures of stations as the possible journeys decreased to 60750.

Task 3 Linear Search Algorithm Evaluation:

The time complexity of linear search is **O(n)**, this is because the items in the list are only compared once. This means that linear search algorithm will take proportionally longer to complete as the input size grows. The space complexity is **O(1)** this means that the memory space required by the algorithm to process data is constant, it does not grow with the size of the data.

The total time taken for this algorithm to complete this task was 0.000997 seconds this is very fast as the size of the list is very small therefore, it does not need much time to search through and find the target.

Table 1: Comparing the Performance of all the Tasks

	TIME COMPLEXITY	SPACE COMPLEXITY	GROWTH RATE	PROCESSING TIME (TASK1A / TASK2A / TASK3)	PROCESSING TIME (HISTOGRAM) (TASK1B / TASK2B)
TASK 1	**O(V^2)**	**O(V)**	**n^2**	**0.0009973 seconds**	**171.41 seconds**
TASK 2	**O(E log V)**	**O(log(E))**	**C log(n)**	**0.15758 seconds**	**81.39 seconds**
TASK 3	**O(n)**	**O(1)**	**n**	**0.000997 seconds**	**-**

3) DISCUSSION FOR THE CHOICE OF TEST DATA YOU PROVIDE AND A TABLE DETAILING THE TESTS PERFORMED:

To understand if algorithms provide the required outputs, we have been testing all the algorithms that have been written. While we were checking the process and functions, we used Black-box and White-Box testing.

Black-box tests are checking if software does what it says on the tin. The tests are devised to explore various inputs and corresponding outputs, without any knowledge of the working of the program itself. This also known as “Functional Testing”.

Therefore, while we were writing the algorithms for all the tasks, we were doing White-box testing, also known as System or Structural Testing. That allows us to test the various path that the program execution can take. We were checking that each path through our code works as expecting. Every so often, we would debug the program to understand the steps that the algorithm was going through. Generally, the algorithms worked as expected, while we were working on the algorithms some of the functions failed the test, then we did the debugging and tested with the smaller data sets, to get the correct output.

Table 2: Task 1 Black-Box Testing

CASE	TEST CASE (WHAT IS BEING TESTED)	ACTIONS	MANUEL INPUTS	EXPECTED OUTPUTS	ACTUAL OUTPUT	PASS - FAIL	CORRECTIVE ACTION
1	Can the program find the shortest path ?	Shortest path algorithm has been implemented. Destinate from: Canary Wharf Arrive to: Bermondsey	In order to use Dijkstra's Algorithm, "Netwrokx" library used.	1st:Canary Wharf 2nd:Canada Water 3rd:Bermondsey	List of stations the customer will travel: ['Canary Wharf', 'Canada Water', 'Bermondsey']	Pass	-
2	Does program calculate the correct taken time between stations ?	The weights are added to each other by "networkx".	In order to find taken time on path "Netwrokx" library used.	10 minutes according to provided data sheet.	Total taken time: 10 minutes.	Pass	-

Table 3: Task 2 Black-Box Testing

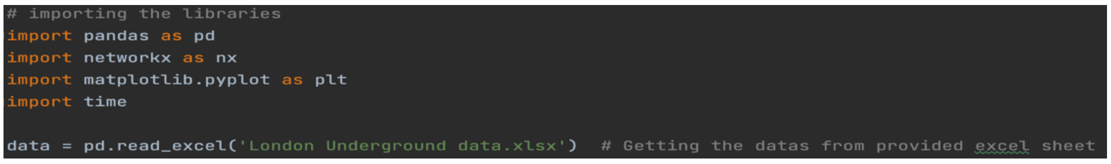
CASE	TEST CASE (WHAT IS BEING TESTED)	ACTIONS	MANUEL INPUTS	EXPECTED OUTPUTS	ACTUAL OUTPUT	PASS - FAIL	CORRECTIVE ACTION
1	Can algorithm provide closure between immediate neighbour stations ?	Kruskal's Algorithm has been implemented with class.	Created a new graph which does not form any cycle on the tree	List of closed immediate neighbour stations for each line.	Following immediate neighbouring stations could be closed from Victoria line: Euston - King's Cross St. Pancras...	Pass	-
2	Does program measure how many immediate neighbours station can be closed?	Taking the length of the [closed] list.	The immediate neighbour stations skipped while creating the new graph	109 stations can be closed according to data sheet.	Maximum 109 immediate neighbouring stations can be closed.	Pass	-

Table 4: Task 3 Black-Box Testing

CASE	TEST CASE (WHAT IS BEING TESTED)	ACTIONS	MANUEL INPUTS	EXPECTED OUTPUTS	ACTUAL OUTPUT	PASS - FAIL	CORRECTIVE ACTION
1	Can algorithm provide the which line user needs to get in current station	Lines has been called through the statement.	If statement in order to create condition to call the lines.	Providing lines with stations	Start from "Green Park" in "Jubilee" line, when you reach "Baker Street", change line to "Circle".	Pass	-

4) ***SCREEN-CAPTURED DEMONSTRATION OF THE WORKING SOURCE CODE:***

Task 1 provides the shortest path between given stations by the user by using the “networkx” library which already exist in python packages that uses Dijkstra’s algorithm (Figure 1).
First, we create a graph including all the immediate neighbour stations(edges) where we receive the data from the provided excel sheet (Figure 1).

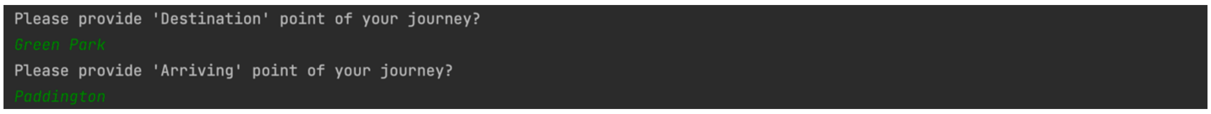


```
# importing the libraries
import pandas as pd
import networkx as nx
import matplotlib.pyplot as plt
import time

data = pd.read_excel('London Underground data.xlsx')  # Getting the datas from provided excel sheet
```

Figure 1: Importing libraries and London Underground data sheet.

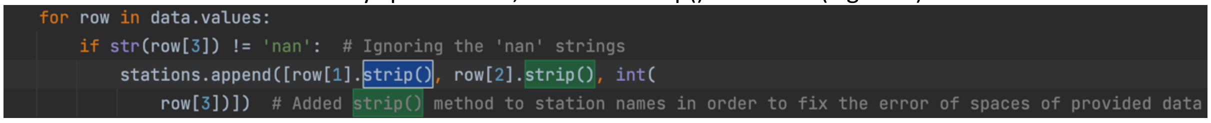
To determine where the user wants to go, we asked the user for input, that provides us with the stations they wanted to travel to (Figure 2). When the program receives the input, it checks if the user input matches with the data set.



```
Please provide 'Destination' point of your journey?
Green Park
Please provide 'Arriving' point of your journey?
Paddington
```

Figure 2:Required input from the user in the terminal

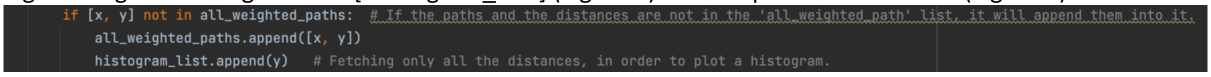
In order to avoid the unnecessary space errors, we used “.strip()” method. (Figure 3)



```
    for row in data.values:
        if str(row[3]) != 'nan':  # Ignoring the 'nan' strings
            stations.append([row[1].strip(), row[2].strip(), int(
                row[3])])  # Added strip() method to station names in order to fix the error of spaces of provided data
```

Figure 3: ".strip()" method.

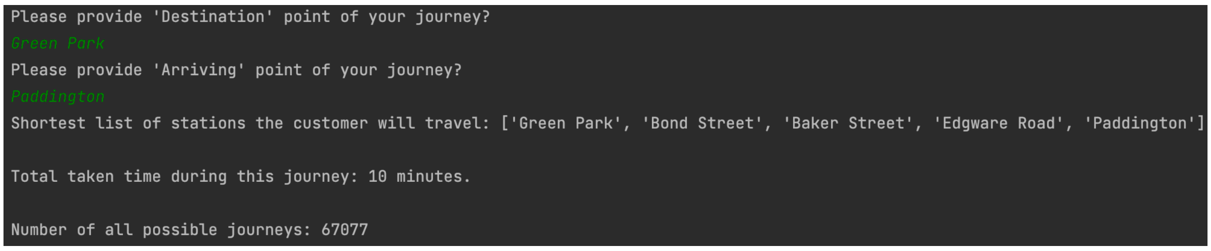
The algorithm can measure the total number of the all the possible journeys. To be able to calculate that, the algorithm gets the length of the [“histogram_list”] (Figure 4). The output can be seen on (Figure 5).



```
        if [x, y] not in all_weighted_paths:  # If the paths and the distances are not in the 'all_weighted_path' list, it will append them into it.
            all_weighted_paths.append([x, y])
            histogram_list.append(y)   # Fetching only all the distances, in order to plot a histogram.
```

Figure 4: Calculating the number of the possible journeys

Once, the program receives all the required inputs, it starts to run the algorithm to find the shortest path by using Dijkstra’s Algorithm. In this report as an example, destination point is Green Park and arriving point is Paddington. The software does that by checking the given functions and conditions. If the statements become true, output will be printed on the terminal (Figure 5). Also, as requested on the specifications, the “networkx” library calculates the total taken time during the journey by adding the weights(edges) to each other. That will be printed on outputs as well (Figure 5).



```
Please provide 'Destination' point of your journey?
Green Park
Please provide 'Arriving' point of your journey?
Paddington
Shortest list of stations the customer will travel: ['Green Park', 'Bond Street', 'Baker Street', 'Edgware Road', 'Paddington']

Total taken time during this journey: 10 minutes.

Number of all possible journeys: 67077
```

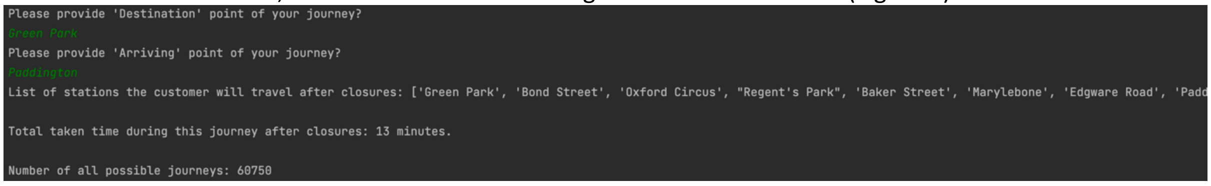
Figure 5: Output of the Task 1a

Task 1 is also providing the histogram that shows the frequency of the Journeys according to times. Histogram displays the quickest journeys (Figure 14).

**Note: While testing the source code in order to get the quickest result from the Task 2, you might consider taking mentioned lines into comment in (Task1a_b.py)=(Line 56/57/58 and for histogram line 9292)

Task 2 is providing a software to assist the government with the decision of closures. The algorithm is designed to close all possible immediate neighbouring stations. The journey should still be possible between any pair of stations in a different route. In order to do that the program uses “Kruskal’s Algorithm” (2020). This algorithm does not form any cycles on the tree as that represents our closures.

As an output, Task 2 provides the alternative path after the closures by calling Task 1 functions again with using the graph that Kruskal’s Algorithm created (Figure 6). The path that is found on Task 2 is longer than Task 1 because of the closures, due to that taken time is higher than Task 1 as well (Figure 6).



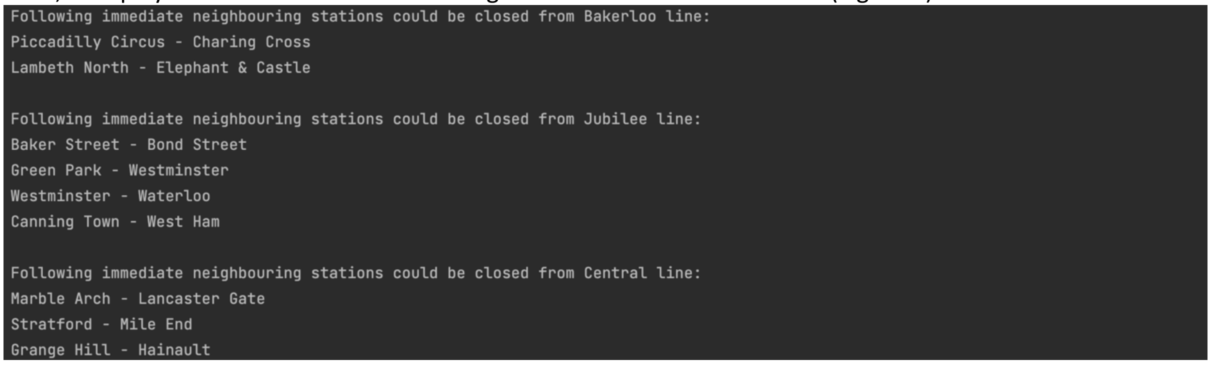
```
Please provide 'Destination' point of your journey?
Green Park
Please provide 'Arriving' point of your journey?
Paddington
List of stations the customer will travel after closures: ['Green Park', 'Bond Street', 'Oxford Circus', "Regent's Park", 'Baker Street', 'Marylebone', 'Edgware Road', 'Padd

Total taken time during this journey after closures: 13 minutes.

Number of all possible journeys: 60750
```

Figure 6:Output of Task 2a (Kruskal's Algorithm)

Also, it displays all the closed immediate neighbour stations with their lines (Figure 7).



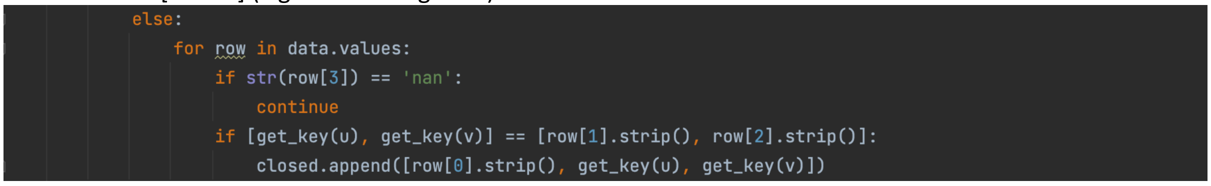
```
Following immediate neighbouring stations could be closed from Bakerloo line:
Piccadilly Circus - Charing Cross
Lambeth North - Elephant & Castle

Following immediate neighbouring stations could be closed from Jubilee line:
Baker Street - Bond Street
Green Park - Westminster
Westminster - Waterloo
Canning Town - West Ham

Following immediate neighbouring stations could be closed from Central line:
Marble Arch - Lancaster Gate
Stratford - Mile End
Grange Hill - Hainault
```

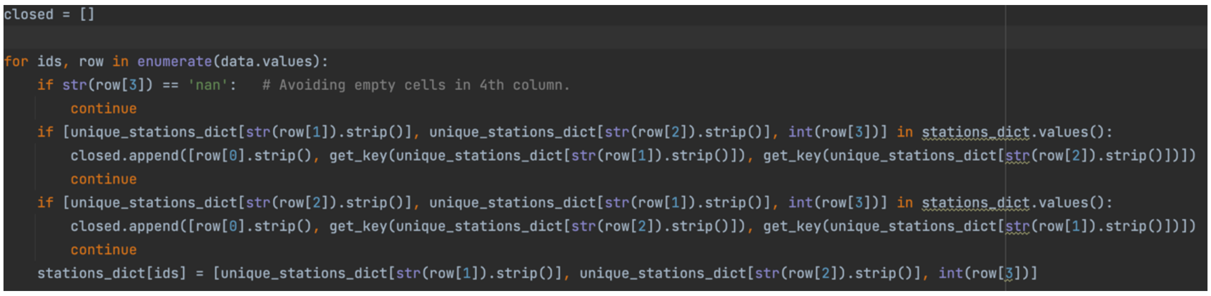
Figure 7:Example of the output for Task 2a, closed immediate neighbour station with their lines(All the lines provided on source code).

Additionally, as a sub-feature the program for Task 2 can provide the maximum of how many closures are possible (Figure 10). The software measures using the graph that is created with Kruskal’s Algorithm by adding into list called [closed] (Figure 8 and Figure 9).



```
            else:
                for row in data.values:
                    if str(row[3]) == 'nan':
                        continue
                    if [get_key(u), get_key(v)] == [row[1].strip(), row[2].strip()]:
                        closed.append([row[0].strip(), get_key(u), get_key(v)])
```

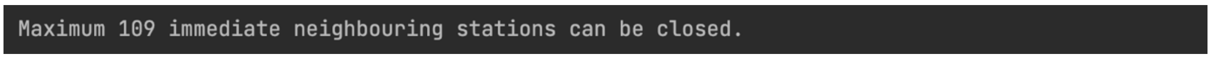
Figure 8: Appending the data that creates cycle on Graph.



```
closed = []

for ids, row in enumerate(data.values):
    if str(row[3]) == 'nan':   # Avoiding empty cells in 4th column.
        continue
    if [unique_stations_dict[str(row[1]).strip()], unique_stations_dict[str(row[2]).strip()], int(row[3])] in stations_dict.values():
        closed.append([row[0].strip(), get_key(unique_stations_dict[str(row[1]).strip()]), get_key(unique_stations_dict[str(row[2]).strip()])])
        continue
    if [unique_stations_dict[str(row[2]).strip()], unique_stations_dict[str(row[1]).strip()], int(row[3])] in stations_dict.values():
        closed.append([row[0].strip(), get_key(unique_stations_dict[str(row[2]).strip()]), get_key(unique_stations_dict[str(row[1]).strip()])])
        continue
    stations_dict[ids] = [unique_stations_dict[str(row[1]).strip()], unique_stations_dict[str(row[2]).strip()], int(row[3])]
```

Figure 9 :Appending the values that has been skipped because of the repeated data.

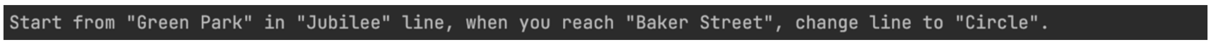


```
Maximum 109 immediate neighbouring stations can be closed.
```

Figure 10:Output of the closed immediate neighbour stations.(Task 2)

Task 2 also provides the histogram after the closures, that has been measured on Task 2 (Figure 15).

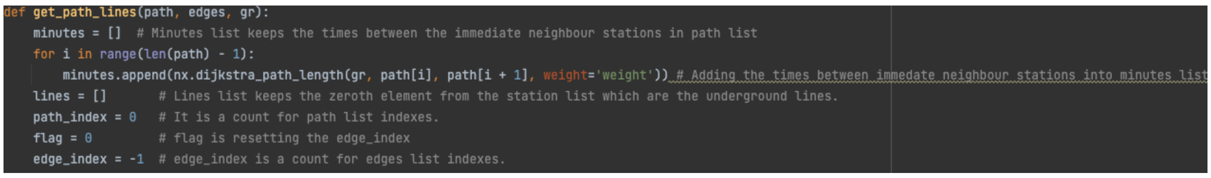
In Task 3, we have created our own idea that is related to London Underground Data which requires a different Algorithm from those used for Task 1 and Task 2. On this task the program is using Linear Search Algorithm. Where the program displays to the user, which station they need to change to which line. When there is more than one line to the same route, the user needs to know which line should be taken (Figure 11).



```
Start from "Green Park" in "Jubilee" line, when you reach "Baker Street", change line to "Circle".
```

Figure 11: Task 3 example output.

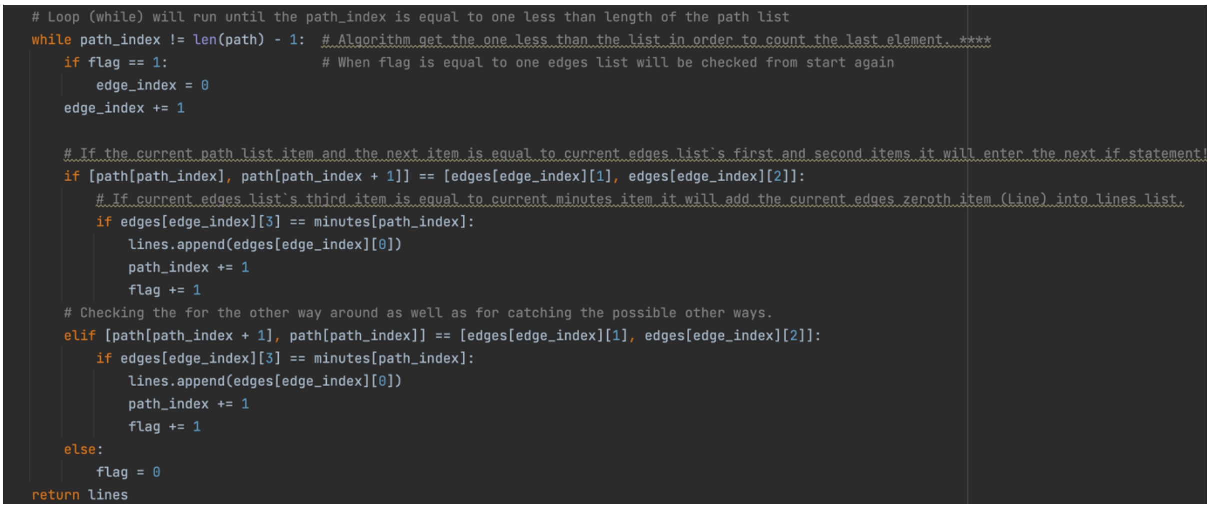
In order to provide this feature to the user, first the function called “get_path_lines” gets the lines of each immediate neighbour stations in the path (Figure 12).



```python
def get_path_lines(path, edges, gr):
    minutes = []  # Minutes list keeps the times between the immediate neighbour stations in path list
    for i in range(len(path) - 1):
        minutes.append(nx.dijkstra_path_length(gr, path[i], path[i + 1], weight='weight')) # Adding the times between immedate neighbour stations into minutes list
    lines = []       # Lines list keeps the zeroth element from the station list which are the underground lines.
    path_index = 0   # It is a count for path list indexes.
    flag = 0         # flag is resetting the edge_index
    edge_index = -1  # edge_index is a count for edges list indexes.
```

Figure 12: Creating a list called minutes: That restores the times between the immediate stations in path.

The linear search algorithm uses a while loop to iterate through the list to find the target item, it does this by searching through the list from the start until it finds a match to the target element. While loop will run until the “path_index” is equal to one less than total items of the path list. It will find and add lines of each immediate neighbour stations in the path list to the “lines” list (Figure 13).



```python
    # Loop (while) will run until the path_index is equal to one less than length of the path list
    while path_index != len(path) - 1:  # Algorithm get the one less than the list in order to count the last element. ****
        if flag == 1:                   # When flag is equal to one edges list will be checked from start again
            edge_index = 0
        edge_index += 1

        # If the current path list item and the next item is equal to current edges list`s first and second items it will enter the next if statement!
        if [path[path_index], path[path_index + 1]] == [edges[edge_index][1], edges[edge_index][2]]:
            # If current edges list`s third item is equal to current minutes item it will add the current edges zeroth item (Line) into lines list.
            if edges[edge_index][3] == minutes[path_index]:
                lines.append(edges[edge_index][0])
                path_index += 1
                flag += 1
        # Checking the for the other way around as well as for catching the possible other ways.
        elif [path[path_index + 1], path[path_index]] == [edges[edge_index][1], edges[edge_index][2]]:
            if edges[edge_index][3] == minutes[path_index]:
                lines.append(edges[edge_index][0])
                path_index += 1
                flag += 1
        else:
            flag = 0
    return lines
```

Figure 13: Fetching the lines of each immediate neighbour stations in path

Once all the required conditions turned True the program will display the output as above (Figure 11).

For Task 1b and Task 2b, the histograms for all the possible journeys with total taken times which is Task 1b (Figure 14) and all the possible journeys after as many as possible closures which is Task 2b (Figure 15) have been created by using “matplotlib.pyplot” and “pandas” libraries. “Matplatlib.pyplot” is used in order to display the histogram with the taken data sets. The data sets have been taken with the library called “Pandas”.

5) ***OUTCOMES OF TASK 1, 2, AND 3:***

On Task 1, we have been going through with all the shortest path algorithm, such as Dijkstra's Algorithm and Kruskal's Algorithm etc., while we were working on those algorithms, we have learned how to use the minimum spanning tree algorithms, and the results of those. Also, according to Task 1b and Task 2b, we have learned how to create a histogram, and analyse the data. In order to create those histograms, we discovered the extension of "pyplot" of library called "matplotlib".

To be able to understand the difference between Task 1a and Task 2a, we need to compare the (Figure 5) and (Figure 6) [Please see above!]. On the output of the Task 1a, the user can see the shortest path of the interstation by using Dijkstra's Algorithm. The total taken time on this path is 10 minutes (Figure 5). According to specifications of the Task 2a, once the closures applied, total taken time has increases to 13 minutes (Figure 6). [Given stations are just for an additional test.] In the analysis we made by looking at these outputs, we observed that the Task 2 found longer paths than the Task 1. As we used the datasets in histograms as well, we are expecting to see the difference more clearly on the histograms [Please see the (Figure 14) and (Figure 15)].

Normally number of all the possible journeys has to be 72900, where this amount has been found with the square of number of all stations (which is 270^2). We find that there is 270 stations at the total by appending the entire column, where stations stands in provided data sheet to a list, and then taking the unique version of that list. After ignoring the repeated stations from the provided data sheet (which has some errors), that number decreased to 67077 in Task 1 (Figure 5). Once the closure has been done, number of the all the possible journeys become 60750 in Task 2 (Figure 6). In the second task, number of the possible journeys approximately 7000 data less. All these outputs obtained proves the paragraph mentioned above.
At the same time, Task 2 provides the closed immediate neighbour stations for each line as above (Figure 7).

On the third task, we inherit the functions from Task 1 into Task 3, therefore we did the same thing for Task 2. The linear search algorithm still uses the networkx library in order to find the path. Once the shortest path has been found, linear search algorithm checks the lines for each immediate neighbour stations on the path in order to provide the required output.

Table 5: Outcome of the Tasks

TASK 1	TASK 2	TASK 3
*Shortest Path Algorithms	*Spanning Tree Algorithms	*Searching Algorithms
*How to use Dijkstra's Algorithm	*How to use Kruskal's Algorithm	*How to use Linear Search Algorithm with Dijkstra's
*How to implement data into algorithm and using provided data	*How to analyse the data on histogram	*Creating and implementing the new functions
*"pyplot" as an extension of Matplotlib	-	-

If we compare the histograms, the first graph (Figure 14) which provides Quickest Journey Times, rises with a faster acceleration compared to the second graph (Figure 15). This means that while all possible journeys can be made, the Journey frequency obtained increased more steeply in certain minutes compared to the second graph. From here, we can deduce that there is much more path to go in these minutes. In the Task 2 Histogram (Figure 15), after the possible lines are closed, we can see that the travel frequency increases in the higher time periods obtained in the graph. As if we check at the end of the Task 2 Histogram (Figure 15), we have more data comparing to Task 1 Histogram (Figure 14). That task teach us how to do the data analyses and how to read the graph / chart.

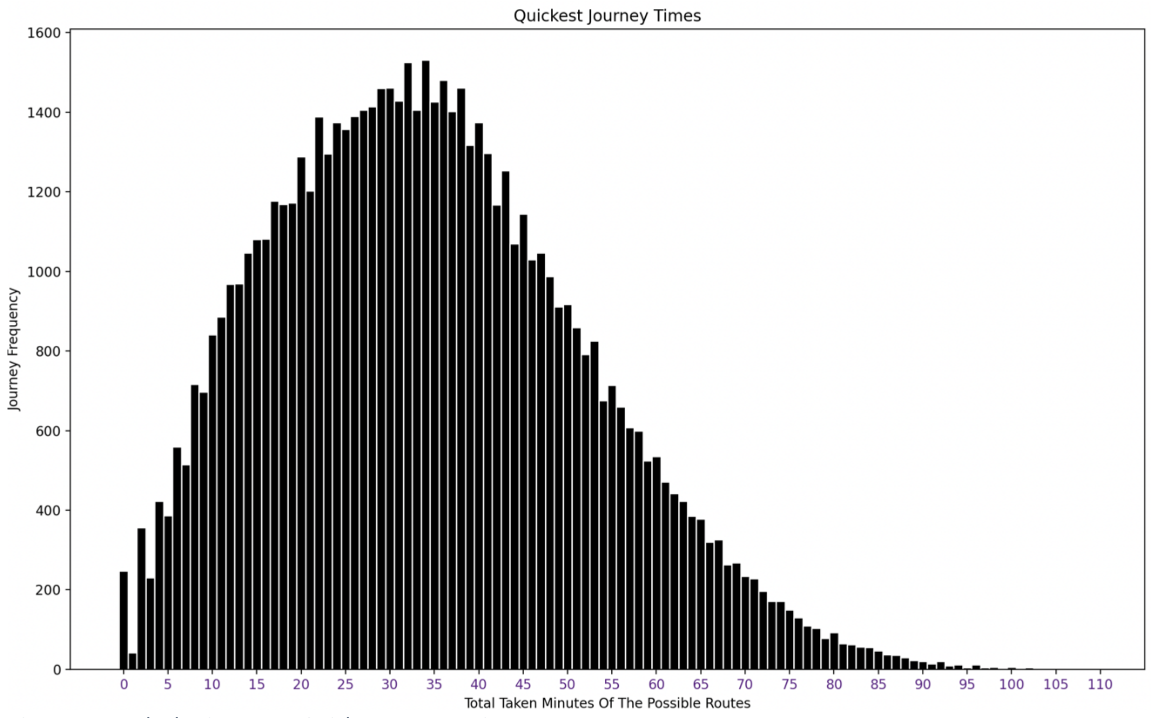


Figure 14: Task 1b Histogram-Quickest Journey Times

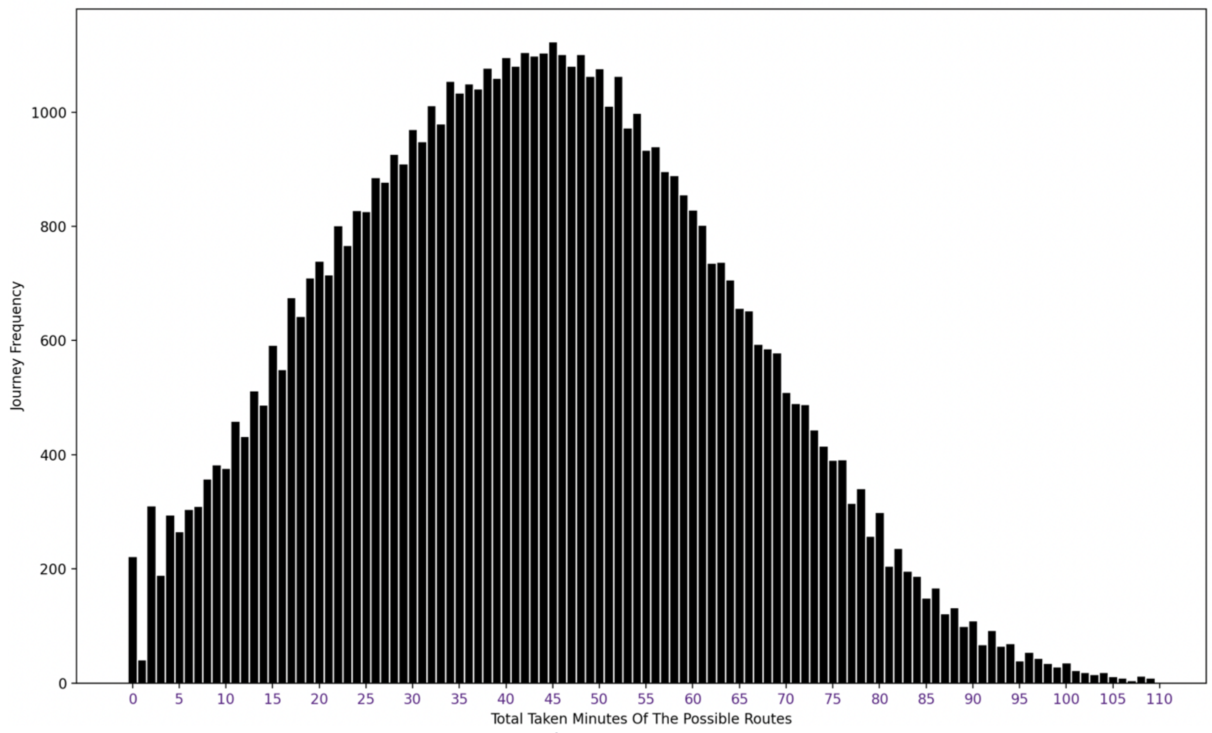


Figure 15:Task 2b Histogram - Quickest Journey Times after closures

6) ***CONCLUSION AND CRITICAL DISCUSSION ON THE LIMITATION OF THE WORK DONE:***

For Task 1a, we decided to use Dijkstra's algorithm in order to get an accurately planned shortest route and how long would that route take based on the given London Underground data. One and most considerable limitation on Dijkstra's algorithm is, it does a blind search by consuming a lot of time waste of necessary resources. As an alternative usable algorithm, for example if we consider the A* minimum spanning tree algorithm, it will work without wasting this time. At the same time, it has different limitations too. Such as, if the branching factor is not finite and every action does not have fixed cost, the algorithm is not complete.

For Task 1b, the histogram without the closures, we have been required to find the journey times of every possible route that can be formed and reflect that to a histogram. In our algorithm we used "for loops" which are inside of each other in order to find every possible routes. After a consultation of our thoughts, the best way we saw to close as many immediate neighbouring stations (edges) as possible was to turn the data we have, to a minimum spanning tree. Which we found on Kruskal's algorithm what exactly we were looking for and so we decided to use it for Task 2a. We believe we could do that with adding some conditions by using Dijkstra's algorithm, but we also believe that it would not be as accurate as the way with Kruskal's algorithm.

After all this work is completed, we could re-use our function that we created for Task 1b to get a new histogram (for Task2b) with the spanning tree we have. And the reason we did that way is to prevent unnecessarily repeated codes.

For Task 3, we created our own task which was to display the train line that the user has to take to travel, and at which station they should change to use the line. To complete this task, linear search algorithm was used. The system now outputs the station the user should change lines from, and what line to use. However, the limitation for this task was that when there are multiple lines a user can take that all have the same path, the system only outputs one of the lines instead of giving the user an option to choose between them. For example, from Baker Street to Euston Square station the user can either take Circle, Hammersmith & City or Metropolitan line but the program only shows one of these lines to the user instead of giving an option to choose from. This is a limitation, because the user can take the other lines since they all follow the same path based on the example given. As an alternative solution to this problem, we could fetch all the possible lines that could be taken and show as option where necessary however, the system would not be as efficient than how it is now.

7) ***WEEKLY LOG OF PROGRESS, INDVIDUAL CONTRUBITION TOWARD THE FINAL OUTOCME BY THE EACH TEAM MEMBER:***

Table 6: Progress as a Group

WEEK	PROGRESS AS A GROUP
24/OCT	Requirements were reviewed, and the search for the most suitable algorithm began. Each member searched different algorithms.
31/OCT	Group decided the which Algorithms we are going to use and, research has begun about how to implement them into our program. Each member worked on different data structures.
7/NOV	The task of the algorithms was created and started to be developed with given data sheet
14/NOV	The program was ready and start to work on data analyses for histogram.
21/NOV	The written algorithms were tested, and report started to be written.

Table 7: Individual Research Process

PROGRESS INDIVIDUALLY				
WEEK	Chisel Chavush	Baris Celiktutan	Arda Sahan	Ozan Gokberk
24/OCT	A* Algorithm	Dijkstra's Algorithm	Bellman-Ford Algorithm	Johnson's Algorithm
31/OCT	Array	Stack	Queue	Linked list
7/NOV	Task 1a	Task 1b / Task 2b	Task 2a	Task 3
14/NOV	Testing Algorithm	Testing Algorithm	Testing Algorithm	Testing Algorithm
21/NOV	Worked on report	Worked on report	Worked on report	Worked on report

Table 8:Allocation of Marks

NAME	Allocation of marks agreed by team (0-100%)
CHISEL CHAVUSH (001131628)	%100
BARIS CELIKTUTAN (001120868)	%100
ARDA SAHAN (001150659)	%100
OZAN GOKBERK (001151346)	%100

BARIS CELIKTUTAN

I think most of the design and implementation meet with the requirements. Additionally, we tried to impellent few more sub-features into Algorithm which is not in the specifications. I believe we have the useful functions that make the code clear and understandable, in order to explain what the code does, we have been using the comment for ourselves and for the people who is looking into our Program. While we were working on the Program, we learned much better how to use the different datasets, how they can be implemented in different ways into algorithms.

CHISEL CHAVUSH

From my point of view our Algorithm can be improved much more, During the first week of the project we were thinking which Algorithm type suits to our program best and trying the choose the best one, but once we start to go through with the requirements the algorithm that we were planning to use is totally changed. So, we understand for the future before we choose any method, we need to see customer requirements and according to that start to work on. During this project, I've well learned how to read the data sets from the histograms or in any other charts. Also, I believe it's increased my data analyses skill. Therefore, when we were working on the Algorithm comparing the first week and the last week of the project, we increased our teamworking skills, that how we can divide works between each group member and I can see that we learn to take responsibility for any project.

ARDA SAHAN

Working together on this group project was very fun but at the same time very challenging. It has allowed me to work with multiple algorithms to solve a realistic problem, something I use every day the London Underground. I am now familiar with how to apply these algorithms to solve problems similar to this one. furthermore, i have now learnt how to target items in an Excel spreadsheet using a python library, openpyxl. Overall, I am happy with the outcome and believe as a group we managed to create a solution which meets with all the objectives.

OZAN GOKBERK

During this project I learnt a lot. I learnt how to use Dijkstra's algorithm to find the shortest path. Additionally, I learnt how to traverse an Excel spreadsheet using python openpyxl. I was quite confused about Kruskal's Algorithm but managed to grasp the concept in the end. Linear search wasn't anything new to me since I used it for last year’s coursework. Overall, I am happy with how the coursework turned out and how our team chemistry naturally flowed.

References:

Hagberg , A., Schult, D. and Swart, P. (2008) *NetworkX, API documentation*. Available at: https://networkx.org/documentation/networkx-0.37/ (Accessed: October 24, 2022).

Kang, H.I., Lee, B. and Kim, K. (2009) *Path planning algorithm using the particle swarm ... - IEEE xplore, Path Planning Algorithm Using the Particle Swarm Optimization and the Improved Dijkstra Algorithm*. Available at: https://ieeexplore.ieee.org/abstract/document/4756927 (Accessed: October 27, 2022).

Kruskal's algorithm (2020) *Programiz*. Programiz. Available at: https://www.programiz.com/dsa/kruskal-algorithm (Accessed: November 6, 2022).

Storer, J.A. (2013) *Introduction to data structures and algorithms, Google Books*. Birkhauser. Available at: https://books.google.co.uk/books?id=K4vfBwAAQBAJ&dq=data+structures+and+algorithms&lr= (Accessed: November 3, 2022).

Verma, A. (2022) *Python: Get key from value in dictionary, GeeksforGeeks*. Available at: https://www.geeksforgeeks.org/python-get-key-from-value-in-dictionary/ (Accessed: November 13, 2022).