DATA STRUCTURES AND ALGORITHMS – GROUP COURSEWORK

Bethelham Demissie w1722493
Ifeoma Okafor w1941705
Yusuf Jimoh w1884567

Table of Contents

Introduction

The group coursework project required the development of a C# application capable of finding the fastest walking route between any two tube stations within zone 1 of the TfL tube network system. The project was divided into three roles, Yusuf was responsible for version one, Ifeoma for version two, and Bethelham for testing and benchmarking. The project involved creating two versions of the application, one hand-coded and the other utilising any library available in the .NET framework. The report includes a comprehensive analysis of the design of both versions, design and UML diagrams for each, implementation, testing, algorithm analysis, benchmarking analysis, and comparison. The report demonstrates effective teamwork and collaboration in producing high-quality software applications that efficiently find the fastest walking routes. The report concludes with a summary of the project and includes a video showcasing the functionality of both versions of the application, with everyone collaboratively pushing to the GitHub program as well as the Google Doc report. There are no known bugs in the systems.

Software Life Cycle

The group will follow the Waterfall Model for this project, dividing it into five stages: requirements gathering and feasibility, design, implementation, testing, and maintenance. They will develop a C# application to find the fastest walking route between any two tube stations within zone 1 of the TfL tube network system using Dijkstra's shortest path algorithm. UML diagrams will be provided for both hand-coded and .NET framework versions. The group aims to produce high-quality software that fulfils the project requirements and is efficient in handling delays and line closures. The applications will be tested, and algorithm and benchmarking analyses will be conducted. The final stage involves maintenance to meet users' needs and fix bugs.

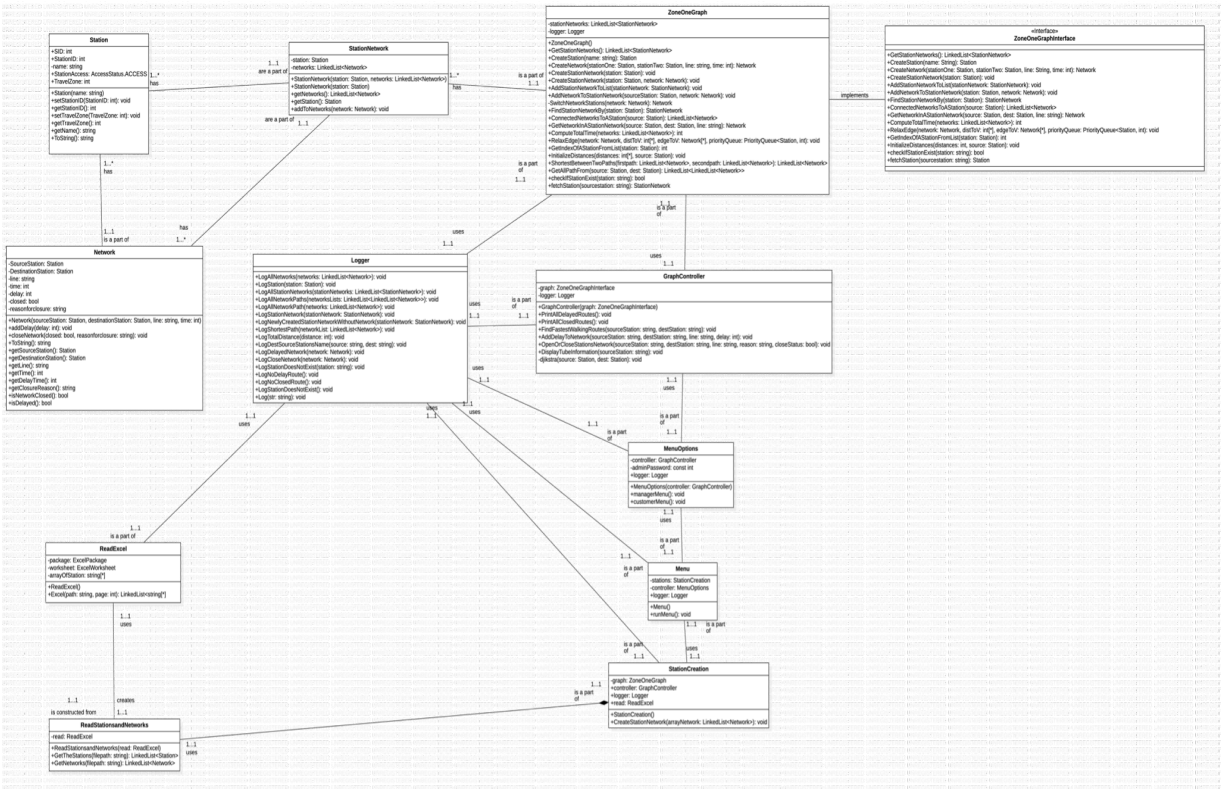
Requirements

The requirements of the application is be implemented in C#. This application is designed to read data from an Excel sheet, identify the meaning of each cell value, and return the shortest walking distance between two train stations provided by the user. It will read an excel sheet that will contain the data about stations are their walking distance and interpret it accurately to generate the desired output. The program should be able to detect errors in the input data and provide appropriate error messages to the user when they enter the wrong station. The shortest walking distance calculation should be based on a well-defined dijkstra shortest path algorithm that takes into account various factors, such as station locations, train line connections, delays, closures and best walking paths between stations. The program should also be able to display the calculated shortest walking distance in a clear and understandable format.

Design

In this segment of the report, we will discuss the design of the two versions of implementation for the application - one being the hand-coded version and other being the .NET framework version. UML diagrams will be provided for both versions. Additionally, a brief description will be given on the data structures utilised to emulate Dijkstra's shortest path algorithm and how they will be implemented using the classes depicted in the diagrams. The group will ensure that the implementation of the algorithm can handle delays and line closures that may occur.

Version 2 UML Diagram



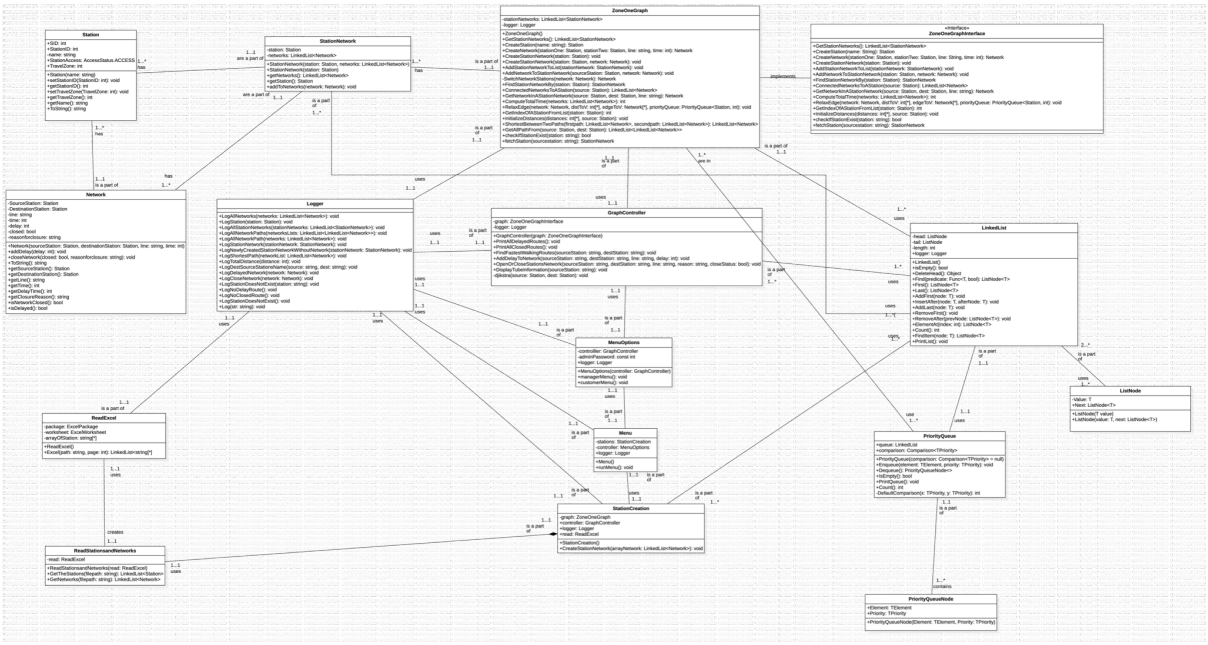
Classes

- Station
- Network
- StationNetwork
- ZoneOneGraph
- GraphController
- ZoneOneGraphInterface
- MenuOption
- StationCreationLogger
- ReadExcel
- ReadStationsandNetworks

Relationships & Multiplicities

1. Network **has** Station : one to many
2. StationNetwork **has** Station: one to many
3. StationNetwork **has** Network: one to many
4. ZoneOneGraph **has** StationNetwork: one to many
5. ZoneOneGraph **uses** Logger: one to one
6. ZoneOneGraph **implements** ZoneOneGraphInterface: one to one
7. GraphController **uses** ZoneOneGraph: one to on
8. GraphController **uses** Logger: one to one
9. MenuOption **uses** GraphController: one to one
10. MenuOption **is a part of** Menu:one to one
11. StationCreation **uses** Menu: one to one
12. ReadExcel **uses** Logger: one to one
13. StationCreation **uses** ReadStationsandNetworks: one to one
14. ReadStationsandNetworks **uses** ReadExcel: one to one
15. Menu **uses** Logger: one to one
16. MenuOption **uses** Logger: one to one
17. StationCreation **uses** Logger: one to one

Version 3 UML Diagram



Classes

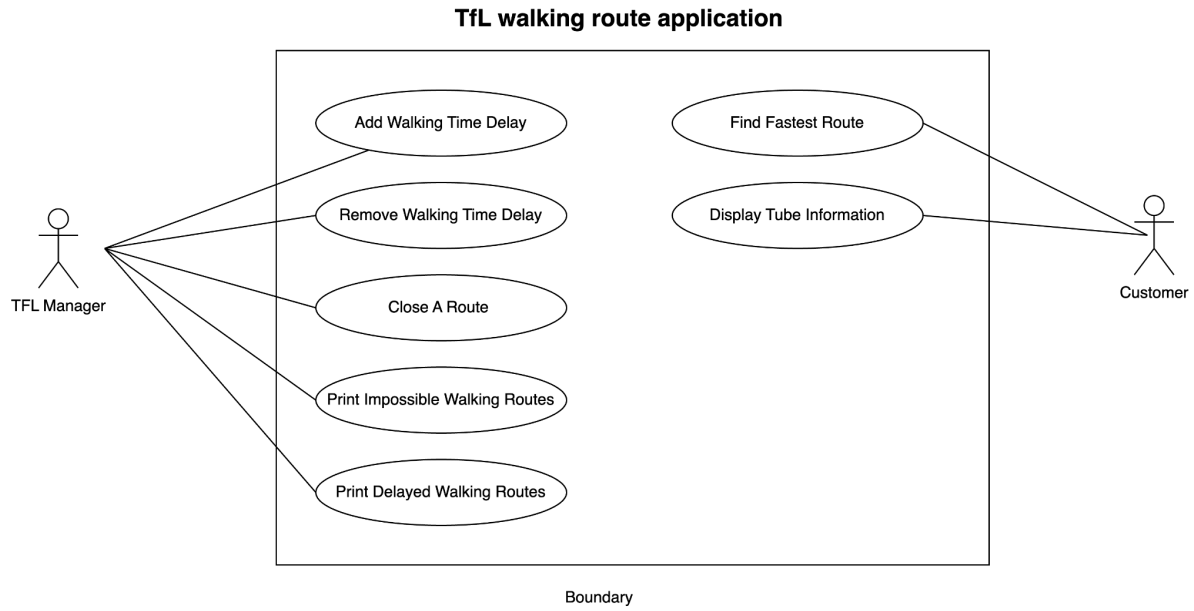
- Station
- Network
- StationNetwork
- ZoneOneGraph
- GraphController
- ZoneOneGraphInterface
- MenuOption
- StationCreationLogger
- ReadExcel
- ReadStationsandNetworks
- LinkedList
- ListNode
- PriorityQueue
- PriorityQueueNode

Relationships & Multiplicities

1. Network **has** Station : one to many
2. StationNetwork **has** Station: one to many
3. StationNetwork **has** Network: one to many
4. ZoneOneGraph **has** StationNetwork: one to many
5. ZoneOneGraph **uses** Logger: one to one
6. ZoneOneGraph **implements** ZoneOneGraphInterface: one to one
7. GraphController **uses** ZoneOneGraph: one to on
8. GraphController **uses** Logger: one to one

9. MenuOption **uses** GraphController: one to one
10. MenuOption **is a part of** Menu:one to one
11. StationCreation **uses** Menu: one to one
12. ReadExcel **uses** Logger: one to one
13. StationCreation **uses** ReadStationsandNetworks: one to one
14. ReadStationsandNetworks **uses** ReadExcel: one to one
15. Menu **uses** Logger: one to one
16. MenuOption **uses** Logger: one to one
17. StationCreation **uses** Logger: one to one
18. LinkedList **uses** ListNode: one to many
19. ZoneOneGraph **uses** LinkedList: one to many
20. PriorityQueue **uses** LinkedList: one to one
21. PriorityQueue **contains** PriorityQueueNode: one to many
22. StationCreation **uses** LinkedList: one to many
23. StationNetwork **uses** LinkedList: one to one
24. GraphController **uses** LinkedList: one to one

Use Case Diagram



Actors

- TFL Manager
- Customer

Use Case to Actor

TFL Manager

- Add walking time delay
- Remove walking time delay
- Close a route

- Print impossible walking routes
- Print delayed walking routes

Customer

- Find Fastest Route
- Display Tube Information

Actor to Actor

- There is no actor to actor relationship

Implementations

In terms of implementation, after designing both versions of the code, the group collectively decided to start by implementing the .NET version to get a better understanding of how Dijkstra's algorithm would practically work. To implement the application, we needed to read the Excel sheet using packages such as EPPlus which will allow us to access and manipulate the cells and data within the sheet showing all the stations walking distance between two given stations. We then parse the data to identify the train stations and their networks/connections.

These are the classes used in the application to enable the program to find the shortest path between two train stations:

Station - The Station class, which represents a train station in the transportation system. It has several properties such as a unique ID number, a name, a tube line, an access method, and a travel zone.

Network - The Network class represents a transportation network between two Station objects. It stores information about the line, time, delay, closure status and reason for closure. It has methods to add delay and close the network, as well as getters for all of its properties. The class uses two Station objects to represent the source and destination of the network. Overall, the Network class allows for the representation and manipulation of a transportation network in a simple and organised way.

StationNetwork - This code defines the StationNetwork class that represents a network of stations. It contains a Station object and a LinkedList of Network objects. The constructor takes a Station object and a LinkedList of Network objects as arguments and initialises the instance variables. There is also a second constructor that takes only a Station object and initialises the network's LinkedList to an empty list. The class provides methods to get the LinkedList of networks, get the Station object, and add a Network object to the networks LinkedList.

ZoneOneGraph - The ZoneOneGraph class represents the graph, and it contains methods for creating stations and networks, adding them to the graph, and retrieving information about them. The class also includes methods for finding connected networks to a station, finding a specific network between two stations, computing the total time of a set of networks, and

relaxing edges in the graph for pathfinding. The graph is implemented using a linked list data structure, with each node representing a station and a list of networks connected to it. The code also includes logging functionality to keep track of the state of the graph during execution.

GraphController - The GraphController class contains several methods to perform different operations on the tube network graph. The class implements methods for finding walking routes between stations, adding delays to connections, and opening/closing stations. The implementation uses a ZoneOneGraphInterface, which provides a way to retrieve the graph of stations and their connections. The class also uses a PriorityQueue and Djikstra's algorithm to find the shortest path between two stations.

Menu - The Menu class is responsible for displaying a console menu to the user and handling their input. It contains an instance of the StationCreation and MenuOptions classes, as well as a Logger instance to log messages to the console. The runMenu() method displays a menu to the user with options for either the customer or admin menu. The user's input is read and processed accordingly with the help of the MenuOptions class. The logger is used to display messages to the console, such as the header for the menu and error messages for invalid user input.

Logger - The logger class is a class that facilitates logging in the application. It provides methods to log different types of messages and prints them out in a readable format. The logger class helps to keep the code organised and tidy by separating the logging functionality from the rest of the application logic.

MenuOptions - The MenuOptions class has two methods, managerMenu and customerMenu, for different types of users. The managerMenu method requires a password for access and allows the manager to perform various actions such as adding or removing delays, indicating if a route is possible, printing out lists of impossible and delayed routes, and opening or closing stations. The customerMenu method does not require a password and allows customers to find the fastest walking route between two stations or display tube information. Both methods use a Logger object to log messages, and the class has a GraphController object for performing the actual operations. There are some input validations to ensure the correct data types are used.

ReadExcel - The ReadExcel class reads data from an Excel file using the Microsoft.Office.Interop.Excel and EPPluslibraries. It has a constructor that initialises a linked list of arrays of strings and an array of strings. The class has a public method called "Excel" that takes a file path and an integer page number as arguments and returns a linked list of arrays of strings. The method opens the Excel file using a file stream and the EPPlus library, then reads the specified page of the file into a worksheet. It then loops through each row of the worksheet, reads the values of the first four cells of the row into an array of strings, and adds the array to the linked list.

ReadStationsandNetworks - The ReadStationsandNetworks class is responsible for reading data from an Excel file using the ReadExcel class and converting it into a linked list of Station or Network objects. The GetTheStations method reads data from the Excel file, creates a

linked list of string arrays representing the data, and then converts each string array into a Station object and adds it to a new linked list of Station objects. The GetNetworks method performs a similar task, but instead creates a linked list of Network objects. It checks that the fourth element of each string array can be parsed as an integer before creating a new Network object, using the first and second elements of the array as the source and destination stations, and the third element as the line name.

StationCreation - The StationCreation class is responsible for creating this station network by parsing the data from two Excel files: Zone-1-walkingdistance.xlsx and StationsExcel.xlsx. Once the network is created, it is passed to a GraphController object, which is responsible for implementing graph algorithms on the network, such as finding the shortest path between two stations.

External Links

Youtube Video - https://youtu.be/2BGhAaHLl9k

GitHub Repository (evidence of group working) - https://github.com/yusuphjoluwasen/FastestWalkingRouteWithDjikstraAssignment/tree/menu-class
This is a repository that was made private so that other students could not access our work while each member works on the code separately.

Path for Text Output file in program directory - /FastestWalkingRouteWithDjikstraAssignment/ConsoleApp3/bin/Debug/net6.0

Analysis

Algorithms Used

The main algorithm used in the implementation of this coursework was Dijkstra's Shortest Path Algorithm. Dijkstra's shortest path is a widely used graph theory algorithm that finds the shortest path between two given nodes in a graph, one node being the source and the other node being the destination node (Chen, 2003). It works by assigning a label to each vertex in the graph that determines the minimal length from the starting point to other vertices.

Dijkstra's algorithm uses Breadth-First Search (BFS) to move from a node to its neighbouring nodes. In each iteration, the algorithm considers the vertex with the smallest label and all its unvisited neighbours. By using BFS, the algorithm ensures that it explores all possible paths and finds the shortest path to the destination node. Dijkstra's algorithm is a well-known and effective algorithm for finding the shortest path in a graph, and it has been widely used in many applications such as network routing and GPS systems such as Google Maps (R. Lanning, K. Harrell and Wang, 2014). Therefore, it was a suitable choice for finding the shortest walking path between two TfL stations, as it works well with geographical nodes and vertices.

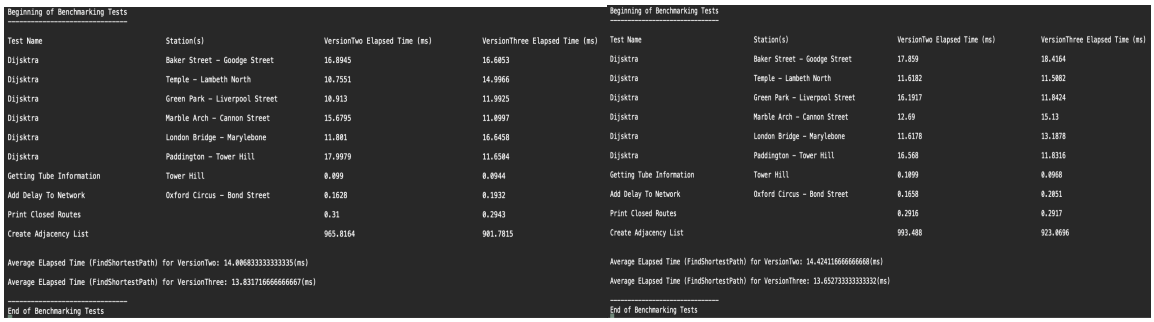
Benchmarking Analysis

Benchmarking tests are designed to test the performance of the applications when a series of different walking route enquiry tests are run on each of them. The goal is to measure and compare the time it takes each application to execute the same set of tests and identify which one performs better.
We conducted two tests on two separate computers, each consisting of two runs to validate the execution time. To obtain the average execution time for each computer, the program was run five times and the mean time was calculated.

The screenshot and table below show the result of the benchmark analysis.

(PERSONAL COMPUTER 1 TESTS)

BENCHMARKING TEST SCREENSHOT (PERSONAL COMPUTER 1 TESTS)



```
Beginning of Benchmarking Tests
-------------------------------
Test Name                       Station(s)                      VersionTwo Elapsed Time (ms)    VersionThree Elapsed Time (ms)
Dijsktra                        Baker Street - Goodge Street    16.8945                         16.6053
Dijsktra                        Temple - Lambeth North          10.7551                         14.9966
Dijsktra                        Green Park - Liverpool Street   10.913                          11.9925
Dijsktra                        Marble Arch - Cannon Street     15.6795                         11.0997
Dijsktra                        London Bridge - Marylebone      11.801                          16.6458
Dijsktra                        Paddington - Tower Hill         17.9979                         11.6504
Getting Tube Information        Tower Hill                      0.099                           0.0944
Add Delay To Network            Oxford Circus - Bond Street     0.1628                          0.1932
Print Closed Routes                                             0.31                            0.2943
Create Adjacency List                                           965.8164                        901.7815

Average ELapsed Time (FindShortestPath) for VersionTwo: 14.006833333333335(ms)
Average ELapsed Time (FindShortestPath) for VersionThree: 13.831716666666667(ms)
-------------------------------
End of Benchmarking Tests
```

```
Beginning of Benchmarking Tests
-------------------------------
Test Name                       Station(s)                      VersionTwo Elapsed Time (ms)    VersionThree Elapsed Time (ms)
Dijsktra                        Baker Street - Goodge Street    17.859                          18.4164
Dijsktra                        Temple - Lambeth North          11.6182                         11.5082
Dijsktra                        Green Park - Liverpool Street   16.1917                         11.8424
Dijsktra                        Marble Arch - Cannon Street     12.69                           15:13
Dijsktra                        London Bridge - Marylebone      11.6178                         13.1878
Dijsktra                        Paddington - Tower Hill         16.568                          11.8316
Getting Tube Information        Tower Hill                      0.1099                          0.0968
Add Delay To Network            Oxford Circus - Bond Street     0.1658                          0.2051
Print Closed Routes                                             0.2916                          0.2917
Create Adjacency List                                           993.488                         923.0696

Average ELapsed Time (FindShortestPath) for VersionTwo: 14.424116666666668(ms)
Average ELapsed Time (FindShortestPath) for VersionThree: 13.652733333333332(ms)
-------------------------------
End of Benchmarking Tests
```

BENCHMARKING TESTS TABLE (PERSONAL COMPUTER 1 FIRST TESTS)

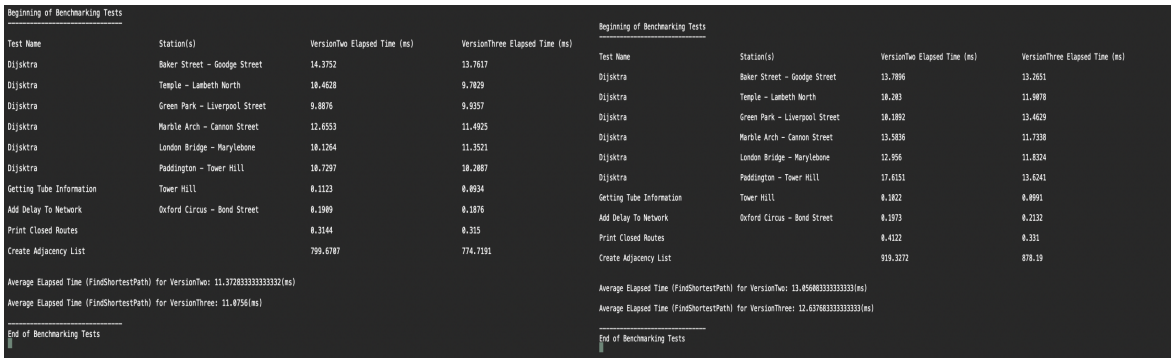
Test Name	Station(s)	VersionTwo Elapsed Time(ms)	VersionThree Elapsed Time(ms)
Djikstra	BakerStreet - Goodge Street	16.8945	16.6053
Djikstra	Temple - Lambeth North	10.7551	14.9966
Djikstra	Green Park - Liverpool Street	10.913	11.9925
Djikstra	Marble Arch - Cannon Street	15.6795	11.0997
Djikstra	London Bridge - Marylebone	11.801	16.6458
Djikstra	Paddington - Tower Hill	17.9979	11.6504
Getting Tube Information	Tower Hill	0.099	0.0944
Add Delay to Network	Oxford Circus - Bond Street	0.1628	0.1932
Print Closed Routes		0.31	0.2943
Create Adjacency List		965.8164	901.7815
Average Elapsed Time (Find Shortest Path) For Version Two		14.0068333ms	
Average Elapsed		13.6527333ms	

Time (Find Shortest Path) For Version Three			

BENCHMARKING TESTS TABLE (PERSONAL COMPUTER 1 SECOND TESTS)

Test Name	Station(s)	VersionTwo Elapsed Time(ms)	VersionThree Elapsed Time(ms)
Djikstra	BakerStreet - Goodge Street	17.895	18.4164
Djikstra	Temple - Lambeth North	11.6182	11.5082
Djikstra	Green Park - Liverpool Street	16.1917	11.8424
Djikstra	Marble Arch - Cannon Street	12.69	15.13
Djikstra	London Bridge - Marylebone	11.6178	13.1878
Djikstra	Paddington - Tower Hill	16.568	11.8316
Getting Tube Information	Tower Hill	0.1099	0.0968
Add Delay to Network	Oxford Circus - Bond Street	0.1658	0.2051
Print Closed Routes		0.2916	0.2917
Create Adjacency List		993.488	923.0696
Average Elapsed Time (Find Shortest Path) For Version Two		14.4241166ms	
Average Elapsed Time (Find Shortest Path) For Version Three		13.8317166ms	

BENCHMARKING TEST SCREENSHOT (PERSONAL COMPUTER 2 TESTS)



```
Beginning of Benchmarking Tests
-------------------------------
Test Name                 Station(s)                          VersionTwo Elapsed Time (ms)   VersionThree Elapsed Time (ms)
Dijsktra                  Baker Street - Goodge Street        14.3752                        13.7617
Dijsktra                  Temple - Lambeth North              10.4628                        9.7029
Dijsktra                  Green Park - Liverpool Street       9.8876                         9.9357
Dijsktra                  Marble Arch - Cannon Street         12.6553                        11.4925
Dijsktra                  London Bridge - Marylebone          10.1264                        11.3521
Dijsktra                  Paddington - Tower Hill             10.7297                        10.2087
Getting Tube Information  Tower Hill                          0.1123                         0.0934
Add Delay To Network      Oxford Circus - Bond Street         0.1909                         0.1876
Print Closed Routes                                           0.3144                         0.315
Create Adjacency List                                         799.6707                       774.7191

Average ELapsed Time (FindShortestPath) for VersionTwo: 11.372833333333332(ms)
Average ELapsed Time (FindShortestPath) for VersionThree: 11.0756(ms)

-------------------------------
End of Benchmarking Tests
```

```
Beginning of Benchmarking Tests
-------------------------------
Test Name                 Station(s)                          VersionTwo Elapsed Time (ms)   VersionThree Elapsed Time (ms)
Dijsktra                  Baker Street - Goodge Street        13.7896                        13.2651
Dijsktra                  Temple - Lambeth North              10.203                         11.9078
Dijsktra                  Green Park - Liverpool Street       10.1892                        13.4629
Dijsktra                  Marble Arch - Cannon Street         13.5836                        11.7338
Dijsktra                  London Bridge - Marylebone          12.956                         11.8324
Dijsktra                  Paddington - Tower Hill             17.6151                        13.6241
Getting Tube Information  Tower Hill                          0.1022                         0.0991
Add Delay To Network      Oxford Circus - Bond Street         0.1973                         0.2132
Print Closed Routes                                           0.4122                         0.331
Create Adjacency List                                         919.3272                       878.19

Average ELapsed Time (FindShortestPath) for VersionTwo: 13.056803333333333(ms)
Average ELapsed Time (FindShortestPath) for VersionThree: 12.637683333333333(ms)

-------------------------------
End of Benchmarking Tests
```

BENCHMARKING TESTS TABLE (PERSONAL COMPUTER 2 FIRST TESTS)

Test Name	Station(s)	VersionTwo Elapsed Time(ms)	VersionThree Elapsed Time(ms)
Dijkstra	Baker Street - Goodge Street	14.3752	13.7617
Dijkstra	Temple - Lambeth North	10.4628	9.7029
Dijkstra	Green Park - Liverpool Street	9.8876	9.9357
Dijkstra	Marble Arch - Cannon Street	12.6553	11.4925
Dijkstra	London Bridge - Marylebone	10.1264	11.3521
Dijkstra	Paddington - Tower Hill	10.7297	10.2087
Getting Tube Information	Tower Hill	0.1123	0.0934
Add Delay to Network	Oxford Circus - Bond Street	0.1909	0.1876
Print Closed Routes		0.3144	0.315
Create Adjacency List		799.6707	774.7191
Average Elapsed Time (Find Shortest Path) For Version Two	11.372833333333332(ms)		
Average Elapsed Time (Find Shortest Path) For Version Three	11.0756(ms)		

BENCHMARKING TESTS TABLE (PERSONAL COMPUTER 2 SECOND TESTS)

Test Name	Station(s)	VersionTwo Elapsed Time(ms)	VersionThree Elapsed Time(ms)
Dijkstra	Baker Street - Goodge Street	13.7896	13.2651
Dijkstra	Temple - Lambeth North	10.203	11.9078
Dijkstra	Green Park - Liverpool Street	10.1892	13.4629
Dijkstra	Marble Arch - Cannon Street	13.5836	11.7338
Dijkstra	London Bridge - Marylebone	12.956	11.8324
Dijkstra	Paddington - Tower Hill	17.6151	13.6241
Getting Tube Information	Tower Hill	0.1022	0.0991
Add Delay to Network	Oxford Circus - Bond Street	0.1973	0.2132
Print Closed Routes		0.4122	0.331
Create Adjacency List		919.3272	878.19
Average Elapsed Time (Find Shortest Path) For Version Two	13.056083333333333(ms)		
Average Elapsed Time (Find Shortest Path) For Version Three	12.637683333333333(ms)		

Dijkstra Complexity Analysis and Comparison to Theoretical Analysis

The time complexity of Dijkstra's algorithm is O((V + E) log V), where V is the number of vertices and E is the number of edges in the graph. This is because the algorithm needs to visit every vertex and edge once, and the priority queue used to implement the algorithm has a (log V) time complexity for inserting and removing elements.

Now, let's look at the execution time of the code. The average execution time of the FindFastestWalkingRoute function in version 2 is 13 milliseconds, while the average execution time in version 3 is 12 milliseconds.

To analyse this execution time, we need to calculate the number of vertices and edges in the graph. From the excel sheet, we know that there are 67 stations and 105 networks - a corresponding 105 networks is created to make the graph undirected, which means there are 67 vertices and 210 edges in the graph.

Using this information, we calculate the theoretical worst-case execution time of the algorithm:

O((V + E) log V) = O((67 + 210) log 67) = O(277 * 4.2) = O(1163.4)

This means the time complexity in the worst case is approximately O(1163.4), which means that the algorithm should take at most 1.1634 seconds to run.

Comparing this to the average running times of the two versions, we see that they are both significantly faster than the theoretical complexity. With an average running time of 13ms and 12ms, respectively they are both well under 1 second.

There are several reasons why the execution time is faster than the worst-case scenario:

1. The implementation of the algorithm using an adjacency list heavily optimises the code to reduce the time per operation as much as possible.

2. The priority queue used in the implementation of the algorithm is optimised for performance, which reduces the time complexity of the algorithm.

3. The input data is well-structured, which reduces the amount of pre-processing required and improves the overall efficiency of the algorithm.

Code Readability and Maintainability

In terms of code readability and maintainability, there was a notable difference between the two versions of Dijkstra's algorithm that were analysed. One version utilised the C# library of linked list, list node, priority queue, and priority queue node, while the other version had to create a hand-coded implementation of these data structures.

The version that utilised the C# library of data structures was significantly more readable and maintainable than the version that had to create the data structures from scratch. This is because the C# library provided well-documented and well-established implementations of these data structures, which are widely used in programming and are therefore well-understood by many developers. This made it much easier to understand and modify the code, as well as to debug any issues that arose.

In contrast, the hand-coded version of the data structures was more difficult to understand and maintain. Since these data structures were not part of the core language and were not widely used, they were less familiar to developers and required more effort to understand. This increased the likelihood of errors or bugs in the code, which would in turn make the code more difficult to maintain.

Furthermore, the hand-coded version of the data structures was less efficient than the C# library implementations. The C# library is optimised for performance and has been tested extensively, which means that it is likely to be faster and more memory-efficient than the

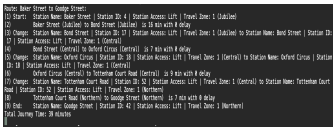
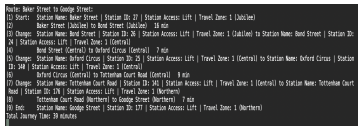
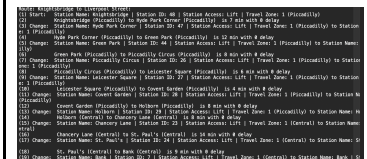
hand-coded version. This would make the code easier to run on large inputs and would improve its overall performance.

Overall, the version of the Dijkstra's algorithm that utilised the C# library of data structures was more readable, maintainable, and more efficient than the version that had to create these data structures from scratch. This demonstrates the importance of leveraging well-established and well-documented libraries when developing software, as this can greatly improve the maintainability and performance of the code.

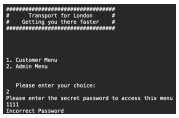
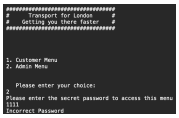
In conclusion, the execution time of Dijkstra's algorithm implemented is faster than the worst-case time complexity of the algorithm due to the small size and sparsity of the graph, as well as optimizations in the priority queue implementation and the well-structured input data.

Consistency Analysis

The purpose of the consistency test is to ensure that both versions of the application, the hand-coded version and .NET version, produce the same shortest route between two stations. This test is important to verify that the output produced by the programs is consistent and correct, and that the shortest path found by both programs is valid and comparable to other possible routes between the tested stations. Through this analysis, we will be able to identify and fix any differences or errors that may exist between the two programs, ensuring that the final form of the program is accurate and reliable.

Method	Stations	Evidence - V2	Evidence - V3	Results
FindFastestWalkingRoutes	Baker Street to Goodge Street			Passed
FindFastestWalkingRoutes for stations far away from each other	Knightsbridge to Liverpool Street			Passed
FindFastestWalkingRoutes for stations of medium distance from each other	Bond Street to Blackfriars			Passed
FindFastestWalkingRoutes for stations close to each other	Green Park to Goodge Street			Passed
DisplayTubeInformation	Tower Hill			Passed

Application Testing

Type of Testing	Description	Input	Evidence V1	Evidence V2	Result (respectively)
Security	Test admin password if correct password is entered				success, success
	Test admin password if incorrect password is entered				success, success

Comparison of Versions

Both version two and version three of the project aim to implement Dijkstra's shortest path algorithm to find the shortest path between two nodes in a graph. However, version two uses hand-coded data structures and algorithms, while version three utilises software available as part of the .NET Framework Libraries.

In version two, the adjacency list is implemented using a hand-coded LinkedList, ListNode, PriorityQueueNode, and PriorityQueue. These data structures are implemented in the code using custom classes and methods. The implementation is relatively complex and requires a deep understanding of data structures and algorithms.

In contrast, version three replaces these custom data structures and algorithms with software available in the .NET Framework Libraries. Specifically, the adjacency matrix is implemented using a built-in LinkedList and PriorityQueue data structures.

The implementation of version three is slightly simpler than version two. It requires less code and is easier to understand and modify. Additionally, because the code uses built-in libraries, it is more standardised and less prone to errors.

The performance of version three is also better than version two. This is because built-in data structures and algorithms are usually optimised for performance and have been extensively tested and refined.

Conclusion

In conclusion, both versions of the project achieve the goal of implementing Dijkstra's shortest path algorithm. However, version three, which uses software available in the .NET Framework Libraries, is a better implementation overall. The hand-coded data structures and algorithms used in version two are complex and prone to errors. Additionally, they are not as optimised for performance as built-in libraries. Version three's implementation is simpler, more standardised, and more likely to perform better.

To sum up, we have concluded from extensive testing and analysis of the two programs that we would recommend using version three's implementation of Dijkstra's shortest path algorithm in future projects.

Reference list

1. Chen, J.-C. (2003). Dijkstra's Shortest Path Algorithm. [online] Available at: https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=2dd9468be1a0ee733a3fb11008d4b2b22dba9e70 [Accessed 11 Apr. 2023].

2. Hester, D. (n.d.). Category Zone 1 Stations - Randomness Guide to London. [online] london.randomness.org.uk. Available at: https://london.randomness.org.uk/wiki.cgi?Category_Zone_1_Stations [Accessed 13 Apr. 2023].

3. R. Lanning, D., K. Harrell, G. and Wang, J. (2014). Dijkstra's algorithm and Google maps. [online] ResearchGate. Available at: https://www.researchgate.net/publication/296639227_Dijkstra%27s_algorithm_and_Google_maps.

4. TFL (2018). Public TfL Data. [online] walking.data.tfl.gov.uk. Available at: https://walking.data.tfl.gov.uk/ [Accessed 7 Apr. 2023].

Code Library Reference

System.IO: This namespace is part of the .NET Framework Class Library and provides classes and types for working with files and directories, including reading and writing files, manipulating paths, and creating and deleting directories. It is found in the ReadExcel class. We use the enums FileAccess.Open and FileAccess.Read to open and read data from the file

OfficeOpenXml: This namespace is a NuGet package included in our project that provides a way to work with Excel files using the OpenXML file format. It allows reading and writing data, formatting, and other information in Excel files without requiring Excel to be installed on the machine. It is an open-source library for reading and writing Excel files in the Office Open XML format (XLSX). It is found in the ReadExcel class. Specifically, it is using the ExcelPackage and ExcelWorksheet classes from the EPPlus library and would need to be installed prior to running the code.