**DESIGN:**

High-Level Overview – Section 10:

Algorithms:

Dijkstra’s algorithm was invented in 1956 by Edsger Dijkstra. He created it when he was trying to calculate the shortest route from Rotterdam to Groningen. The algorithm is an optimisation algorithm, which means that it identifies the best solution among all of the possible solutions for a problem. It is used to find the shortest path from one vertex to another vertex. If you were given a weighted graph which was small, you may be able to calculate the shortest route between the vertices easily. However, Dijkstra’s algorithm can calculate the shortest route on a large graph, while only knowing what vertices are connected to each other (without even knowing what the graph looks like). This is very useful, as the computer is able to calculate the shortest route much quicker than a human could, and it is mathematically proven to be the shortest route, as, unlike the travelling salesperson problem, the problem is tractable. This means that it has a polynomial or less time complexity, and therefore, it can be solved within a reasonable amount of time. Dijkstra’s algorithm is not a heuristic algorithm; it provides the optimal solution using calculations. This is very useful: Dijkstra’s algorithm can therefore be used for a range of problems, including social networking applications (friends lists and people you may know), robotic paths (drones and robots), and navigation services in maps.

(Information above was gathered from <https://isaaccomputerscience.org/concepts/dsa_search_dijkstra?examBoard=all&stage=all> and <https://www.geeksforgeeks.org/applications-of-dijkstras-shortest-path-algorithm/> )

The pseudocode for Dijkstra’s algorithm is shown below:

start\_node <-USERINPUT

end\_node <-USERINPUT

adjacency\_dict <-USERINPUT

nodes\_list <- list(adjacency\_dict.keys())

num\_of\_nodes <- len(nodes\_list)

path\_taken <- {}

distance = {}

visited = {}

FOR node in (adjacency\_dict):

distance[node] <- infinity

visited[node] <- FALSE

distance[start\_node] <- 0

visited[start\_node] <- TRUE

#SELECT THE CLOSEST, UNVISITED NODE:

FOR i in range (num\_of\_nodes):

if i = 0:

closest\_node <- start\_node

else:

list\_of\_unvisited <- []

FOR node in (nodes\_list):

if visited[node] = False:

list\_of\_unvisited.append(node)

distances\_unvisited <- []

FOR unvisited\_node in (list\_of\_unvisited):

distances\_unvisited[unvisited\_node] <- distances[unvisited\_node]

closest\_node <- min in (distances\_unvisited)

current\_node <- closest\_node

adjacent\_nodes <- adjacency\_dict[current\_node]

add\_on\_distance <- distance[current\_node]

keys\_list <- list(adjacent\_nodes.keys())

#CALCULATE DISTANCE TO ALL OTHER NODES:

FOR keys in (keys\_list):

new\_distance <-add\_on\_distance + adjacent\_nodes[key]

#UPDATE THE DISTANCES YOU ALREADY HAVE IF THE NEW DISTANCE IS SMALLER:

if new\_distance < distance[key] AND visited[key] = False:

distance[key] <- new\_distance

path\_taken[key] = current\_node

#MARK NODE AS VISITED AND REPEAT UNTIL ALL NODES ARE VISITED:

visited[current\_node] = TRUE

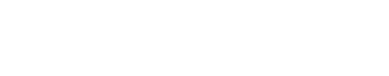
For my project, the user will provide the start node and end node (start location and desired location) and the adjacency list will already be coded. Then, Dijkstra’s algorithm will be applied to give the shortest route between these two nodes. For the step-free journey planner, the main difference will be the adjacency list that is used. Therefore, inheritance and classes can be used in order to inherit attributes and methods (and override them when necessary).

Client-Server Model:

The program will use CGI (Common Gateway Interface). CGI involves requests coming into a web server, the server processing this request (by running CGI script), and then the CGI script sending a response back to the server, which is relayed back to the client. The program should re-direct to the “cgi-bin” folder when forms have been submitted. Then, the python files within the “cgi-bin” folder are run and they output the relevant information in HTML. All of the HTML files will use the same CSS styling so that the website is not difficult to navigate or overwhelming. I chose to use CGI because adding services to the server is very easy, since you only have to add the CGI script into the “cgi-bin” and ensure that the correct CGI script is run for each request. All of the HTML files, the CSS file, and the python webserver are stored in one folder. Within this folder, there is the “cgi-bin” folder containing the CGI scripts, as shown below:

Graphical user interface, text, application

Description automatically generated



Application Programming Interfaces:

APIs are ways that two computers can communicate and share information. They can be very useful in order to get live updates and data. The program will require the use of APIs in order to show information such as status updates for the lines, lift updates, and arrival times at stations. The APIs are all from the TfL website: <https://api-portal.tfl.gov.uk/apis> . The TfL Unified API has lots of information about the tube, buses, trains, and more in London. Since I will need lots of different information, the CGI scripts will be different for the status updates, lifts, and arrival times. Therefore, there will be multiple endpoints used.

The Journey Planner:

The journey planner will use Dijkstra’s algorithm. The user inputs their start station and the station they would like to go to. They also input whether or not they want it to be a step free journey, if it is a weekend, and if it is rush hour (to account for extra time taken on the tube). With this information, Dijkstra’s algorithm is run. This means that the route from the start station to every other station is calculated, and this is then done to the closest node to the start station, and so on and so forth. This means that the best route will be calculated since the algorithm stores the shortest time taken to get to a station by comparing this time to other times, and updating it if required. It also stores the route you need to take in order to get to that node, for example, if the shortest way to get to node B is from node A, then this distance is recorded and then there is also a record of how you got to B (equivalent to “go from A to get to B the quickest). Then, the route is determined by using a recursive algorithm which looks at the record of how to get to the end node. This means that the stored values about how to get to the end node is traced back to the start node. For example, if the network looks like this:



And the start node is A, and the end node is C, then the recursive algorithm looks through the record of how to get to C. For example, if this is the record of how to get to each node:

“Getting to node B: through A

Getting to node C: through B”

(This is an abstracted way of how to store how to get to each node)

Then the recursive algorithm looks at how to get to C firstly, and if it is the start node, then this is the base case. If it is not the start node (which it is not in this example), then the recursive algorithm calls itself on the node used to get there (“B” in this example) and this is the general case. This continues until the start node is found (i.e., the call for “B” will cause the function to stop calling itself since node “A” is how you get to node “B”). Each node that is called is stored, so that the route for this example would be “C”, “B”, “A”, which is the end station to the start station. This would then need to be reversed and outputted for the user.

If the user decides to use the step free journey planner, then the principles are the same, but the only difference is which adjacency dictionary is used as the step free journey planner has different routes that can be taken, and some stations cannot be visited since there is no step free access there, for example:

Map

Description automatically generatedValid Normal Journey Planner Routes:

Valid Step-Free Journey Planner Routes:

A picture containing map

Description automatically generated

Therefore, the adjacency dictionaries used are different.

Diagram

Description automatically generatedDescription Of Algorithms – Section 11:

As discussed earlier, the tube journey planner uses graph theory and Dijkstra’s shortest path algorithm. The steps of this algorithm are illustrated in this flowchart:

This algorithm requires an adjacency list in order to show the connections (edges) between the stations (vertices). In terms of what programming concepts will be used, the algorithm uses iteration in order to go through every node. It uses another for loop in order to iterate through all other nodes to see if there is an arc between them. Therefore, at its core, the algorithm uses a nested for loop. This means that it has a time complexity of O(n2), where n is the number of vertices in the graph/elements in the adjacency list.

In one of the steps, the algorithm requires you to make a note of the node that was used to get to the new one. Therefore, once the algorithm has been run, there should be some sort of way to figure out how to get to the end node from the start node. This could be done using a list: where the index of the values can represent the node you are arriving at, and the element can represent the node you used to get there. Alternatively, this could be done using a dictionary, where the keys are the station you are arriving at, and the value is the node you have used to reach there. For this, you would have to keep going back in the path until you find the start node. Therefore, I would use recursion in order to determine what the path is from the start node to the end node.

The first prototype which has the implementation of Dijkstra’s algorithm was shown earlier in section 4. The first prototype did not work. Dijkstra’s algorithm was used to find the route between two nodes in a graph, however, in the tube journey planner problem, you need to work out if the user would need to change line or stay on the same line, and if they change line, you have to account for this with an added time of 5 minutes. The first prototype did not work since the algorithm could not detect if the user would change lines to get to the desired node. This was clear as, when you enter station 1 as the start node and station 2 as the end node, you did not get the same route as if you entered station 2 as the start node and station 1 as the end node. However, Dijkstra’s algorithm is based on mathematical calculations, therefore the solution is always the same, no matter the start and end node order. There were a few problems with the first prototype which caused the algorithm to not work:

* The algorithm checks if the user would need to change lines. This should be incorporated in the graph somehow in order to be accurate.
* The algorithm relies on a 2D array to search if there has been a line change, when this could be incorporated in a much easier way which the computer can understand.
* At the moment, the program was sort of guessing if the user would need to change lines because the program tries to determine which line was used, but if multiple lines can be used, then the program fails.
* There is no way of determining what line was used before, since there is no data structure holding this information. If there was a list or dictionary, then maybe this would have been possible, but it would still have resulted in an amount of inaccuracy in the route.

In order to fix these problems, the adjacency dictionary was changed so that there were substations for each station which show the connections between different stations and lines. This means that the program doesn’t need to try to find which lines can be used and if it matches the line used previously. After updating the code to accommodate this change, the algorithm was reduced back to the original 4 steps, instead of an extra step in which the lines are checked to see if there was a changeover. This resulted in Dijkstra’s algorithm to work, regardless of the order in which you input two stations.

Description Of Data Structures – Section 12:

Adjacency list:

A key data structure required for using Dijkstra’s algorithm is an adjacency list. An adjacency list is a way of representing a graph. The graph is the network of tube stations and the connections between them. An adjacency matrix represents the arcs and nodes in the graph in a way in which the computer can understand. I decided to use a dictionary data structure in order to represent the adjacency list (called “stations\_adjacency\_dict”). The dictionary keys are the stations, and the values are represented as another dictionary. This “nested dictionary” has keys which are the stations you are travelling to, and the values are numbers which represent the amount of time taken to get to that station. I chose the dictionary data structure as it is an easy way to represent key-value pairs, and the key-value pairs I needed to represent were stations and distances/timings.

For my first prototype, I used an adjacency dictionary in which the keys were the stations you are at. However, when coding up the change-over times when you change lines, I experienced difficulty as it was hard to determine what line had been used before. For this reason, I changed my adjacency dictionary so that the keys were “substations” in the sense that there was a key for each line at the station. This change allowed the journey planner to work because before, it would give different routes even if you use the same stations but switch the start and end stations (i.e. Paddington to Marble Arch would give a different route compared to Marble Arch to Paddington, but this should not happen). After making the change, the journey planner was working as the changeover times would be accounted for successfully.

Output of the Route:

The output of the route should be a list of all of the stations that have been visited in order to reach the end station from the start station. I created the list called “path\_list” for this. I used a list for this as it is an effective way of representing elements in a specific order and outputting them in this order is easy to do.

In the “dijkstra” function, you need to record how you get to each node, i.e, which node was used to traverse to the new node. For example, the below image shows a graph.



Text

Description automatically generated with medium confidenceIf you are going from “A” to “C”, you will need to go to “A”, then “B”, then “C”. Therefore, the node “B” must be recorded somewhere, in order to show how to reach “C”. This was recorded using the “path\_taken” dictionary data structure. In the iteration of the “dijkstra” function, you choose the node which you are starting at and then traverse to all the nodes you can possibly traverse and update distances if they are shorter. If a distance needs updating, then the following code is run:

The third line of code ensures that the path taken to get to the new node has been recorded. The key in the “path\_taken” dictionary is the station you are traversing to, and the “current\_node” variable is the node you are traversing from.

After all of the iteration in the “dijkstra” function, the “path\_list” list is created, initially containing only the end node. Then, it is updated when it is the returned value from the “recursive\_find\_path” function. This function uses recursion to create a list of all of the nodes used to get to the end node from the start node.

Graphical user interface, text, application, letter

Description automatically generated

The function stops calling itself when the start node is reached in the “path\_dict” data structure (base case). Otherwise, the function calls itself in order to determine how to get to the start node from the end node. After this, the “path\_list” is returned and then reversed (as it starts at the end node and ends at the start node). Now, “path\_list” includes all of the stations (including the start and end stations) used to reach the end station from the start station. For example, using the graph from earlier in this section and going from “A” to “C”, “path\_list” would be: [“A”, “B”, “C”], and this is ready to be outputted as the route.

Distances and visited\_nodes:

Dijkstra’s algorithm requires a data structure for recording the distances to nodes from the start node. For this data structure, I decided to use a list. The index position of each element refers to the node you are travelling to. For example, in a graph containing the 3 nodes [“A”, “B”, “C”] and “A” as the start node, distances[0] would be 0, distances[1] would be the distance from “A” to “B”, and distances[2] would be the distance from “A” to “C”. This list is used to check if there is a better route, since its values are compared to new distance values which could result in a better route. It is also used towards the end when the distance/time taken for the journey is outputted. Additionally, Dijkstra’s algorithm requires you to choose the closest, unvisited node. Therefore, there must be some way of checking which noes have been visited, and which nodes have not been visited. I decided to use a dictionary data structure for visited\_nodes, since the key can be the station, and the value can be Boolean (True or False). If it is False, then the node has not been visited yet and can be chosen as the next node.

Objects, Methods, and Attributes:

For the journey planner, there are two options for the user. They can choose to have a step-free route, or a normal route. If they choose the normal route, they can get off at any station, start at any station, get on any line, and get on at any platform. However, if they choose the step-free route, they can only get off at stations which accommodate for step-free access. For example, they may not be able to get off of the tube at a station with no step free access and won’t be able to change lines there, and they may only be able to use the tube in certain directions (North, East, South, or West). Therefore, the key difference between the normal journey planner and the step-free journey planner is the graph. This means that the adjacency dictionaries for both of these will be different, but Dijkstra’s algorithm will be able to work for both of them. After realising this, I decided that I would use inheritance in my project. Inheritance is a technique which allows methods and attributes to be used in one class (child class) from another class (parent class). In this situation, the parent class is “JourneyPlanner” and the child class is “StepFreeJourneyPlanner”. This inheritance relationship is shown in this inheritance diagram and the Unified Modelling Language Class Diagram for these classes is also shown:

Diagram

Description automatically generatedDiagram, letter

Description automatically generated

In the child class, the attributes are all overwritten in the constructor method, and the only other method is the “main()” method. This “main()” method calls methods from the parent class as there are some differences between the order in which code is run for the normal journey planner and the step-free journey planner. The two tables below show more in-depth information about the attributes and methods of the classes:

Attributes:

|  |  |  |
| --- | --- | --- |
| Class Name | Attribute | Function |
| JourneyPlanner, StepFreeJourneyPlanner | station1 | String which stores the start station that the user entered in the form. Value is retrieved from the form after it has been submitted. |
| station2 | String which stores the end station that the user entered in the form. Value is retrieved from the form after it has been submitted. |
| rush\_hour | String which stores a “Yes” or “No” value which the user entered in the form. Represents if the journey is during rush\_hour or not so that the times can be adjusted to simulate this. |
| weekend | String which stores a “Yes” or “No” value which the user entered in the form. Represents if the journey is on a weekend or not so that the times can be adjusted to simulate this. |
| stations\_adjacency\_dict | Dictionary which stores the connections between nodes. Connections are shown by a “nested dictionary”. The key of this is a station, and the value is another dictionary. This nested dictionary has a key which is the node you are travelling to and the value is the time taken to reach there. |
| stations\_list | List which stores all of the stations that are key values in the stations\_adjacency\_dict. Used to access the values in the stations\_adjacency\_dict. |
| num\_of\_stations | Integer which stores the number of items in the stations\_list. Used in order to iterate through all of the stations in the stations\_adjacency\_dict. |
| stepfree | Boolean value which represents whether or not the user has asked for a step free route or not in the form. |

Methods:

|  |  |  |
| --- | --- | --- |
| Class Name | Method | Function |
| JourneyPlanner | \_\_init\_\_ (constructor method) | Takes all of the values from the user and makes them attributes in the class. |
| correct\_stations | Takes the end and start stations that the user entered and changes these values if the stations have different names in the dictionary. For example, the station “Victoria” has to be changed to “VictoriaStation” so that it does not get confused with “VictoriaLine” later in the program. |
| dijkstra | Performs Dijkstra’s algorithm on the stations given by the user. Finds the best route from the start station to the end station. |
| linear\_search | Takes a target and a list and finds the index position of the item in the list. Used to find index position of the stations in the stations\_list. |
| recursive\_find\_path | Takes a dictionary containing keys which are stations and the values are the station used to get there. Uses recursion to output the route from the end node to the start node. |
| correct\_lines | Changes the strings that are used in the strings which are outputted in the route, for example “PaddingtonStation CircleLine” changes to “Paddington (Circle Line)”. |
| main | Calls various methods in the class in order to change station names, perform Dijkstra’s algorithm, create a route list, and output the information required by the user. |
| output\_everything | Outputs the time taken for the journey and the route that the user should take. |
| StepFreeJourneyPlanner | \_\_init\_\_ (constructor method) | Takes all of the values from the user and makes them attributes in the class. |
| main | Calls various methods in the class in order to change station names, perform Dijkstra’s algorithm, create a route list, and output the information required by the user. |

For the TfL API aspect of the project, there will be 3 python files. These are for either the status updates of a line, lift disruptions at a station, and arrivals at a station. They will all use the same subroutines for accessing data, so I created another file which has 1 class and 2 methods. This means that I can import this file into the other 3 files in order to re-use code. Here is a UML diagram of the Diagram

Description automatically generatedclass in the new file:

The dashed lines show that the file containing the class called “Tube\_API” is imported into the 3 files at the bottom.

Another reason why I chose to use objects and object-oriented programming (OOP) is because it reduces the amount of duplicate code. This is good as it means that the code is more efficient and easier to understand.

Design Of User Interface – Section 13:

The results from the questionnaire and interview both suggested that the current systems have overwhelming and confusing user interfaces. For this reason, my project must be easily accessible by the user. This means that there should be a navbar (to easily access the different webpages), a footer, a header, and CSS styling which is not too overpowering on the user’s eyes. The colours on the website that I have decided to use are white, black, and grey (except for images). This is because the end user in the interview had told me that other user interfaces on current systems are hard to use as they are so colourful. The colours on my website can be used to distinguish different sections of webpages, as shown in the initial designs for the user interface below (labels are in blue):







